Principles of Software Engineering

DCAP305

Edited by:
Deepak Mehta

LOVELY
PROFESSIONAL
UNIVERSITY

LOVELY PROFESSIONAL UNIVERSITY

PRINCIPLES OF SOFTWARE ENGINEERING

Edited By

Deepak Mehta

Printed by
FRANK BROTHERS & CO. (PUBLISHERS) LIMITED
B-41, Sector-4, NOIDA - 201301, Gautam Budh Nagar
for
Lovely Professional University
Phagwara

SYLLABUS

Principles of Software Engineering

Objectives:

- To enable the student to understand software development process.
- To enable the student to learn various software engineering approaches.
- To enable the student to implement various software testing techniques.
- To enable the student to prepare software requirement specification documents.
- To enable the student to learn various verification and validation techniques.
- To enable the Student to practice software engineering concepts using UML.

COURSE CONTENTS:

S. No.	Topics
1.	**Introduction:** Concept of Software Engineering. Software Engineering Challenges & Approach.
2.	**Software Processes & models:** Processes and Models, Characteristics of Software Model, Waterfall, Prototype, Iterative, Time Boxing. Comparison.
3.	**Software Requirements:** Problem Analysis, DataFlow, Object Oriented Modelling, Prototyping. **Software Requirement Specification Document: SRS,** Characteristics, Components, Specification Language, Structure of Document.
4.	**Introduction to Validation, Metrics:** Function Point & Quality Metrics. Software Architecture: Architecture Views, Architecture Styles:Client/Server, Shared Data.
5.	**Software Project Planning:** Process Planning, Effort Estimation, COCOMO Model, Project Scheduling and Staffing. Intro to Software Configuration Management: Quality Plan, Risk Management, Project Monitoring.
6.	**Functional Design:** Principles, Abstraction, Modularity, Top Down, Bottom Up Approach. Coupling, Cohesion. Structure Charts, Data Flow Diagrams, Design Heuristics.
7.	**Intro to Verification:** Meaning, Metrics: Network, Stability, Information Flow.
8.	**Detailed Design:** Process Design Language. Logic/Algorithm Design. Verification of Logic/Algorithm Design. Metrics: Cyclomatic Complexity, Data Bindings, Cohesion Metric.
9.	**Coding:** Common Errors, Structured Programming, Programming Practices, Coding standards. Coding Process: Incremental, Test Driven, Pair Programming. Refactoring: Meaning and Example. Verification, Metrics: Size & Complexity
10.	**Testing:** Fundamentals, Error, Fault, Failure, Test Oracles, Test Cases & Criteria. Black Box: Equivalence Class Partitioning, Boundary Value Analysis. White Box: Control Flow Based, Data Flow Based Testing Process: Levels of Testing, Test Plan, Test Case Specifications, Execution and Analysis. Logging and Tracking.Metrics: Failure Data and Parameter Estimation.

CONTENT

Unit 1: Introduction to Software Engineering

CONTENTS

Notes

Objectives

After studying this unit, you will be able to:

- Understand the concepts of software engineering
- Discuss the evolution and impact of software engineering
- Explain the evolution of an art to an engineering discipline
- Define the software characteristics
- Discuss the changing nature of software
- Explain software components
- Discuss the software applications
- Define software myths
- Discuss the term software a crisis on the horizon
- Explain the terminologies in software engineering
- Understand the role of management in software development

Introduction

In all industrialized countries; and increasingly in just beginning countries, computer systems are economically critical. Ever more products incorporate computers in some form. Educational, administrative and health care systems are maintained by computer systems. The software in these systems represents a large and increasing proportion of the total system costs. The effective functioning of modern economic and political systems therefore depends on our ability to produce software in a cost-effective way.

Software engineering is disturbed with the theories, methods and tools which are needed to expand the software for these computers. Generally, the software systems which must be urbanized are large and complex, systems. They are also nonrepresentational in that they do not have any physical form. Software engineering is therefore different from other engineering disciplines. It is not constrained by materials governed by physical laws or by manufacturing processes.

Software engineers model parts of the real world in software. These models are large, abstract and complex so they must be made visible in documents such as system designs, user manuals, and so on. Producing these documents is as much part of the software engineering process as programming. As the real world which is modelled changes, so too must the software. Therefore, software engineering is also concerned with evolving these models to meet changing needs and requirements.

The idea of software engineering was planned in the late 1960s at a conference held to argue what was then called the 'software crisis'. This software crisis resulted directly from the introduction of third-generation computer hardware. These machines were instructions of magnitude more powerful than second-generation machines. Their power made hitherto unrealizable applications a feasible proposition. The performance of these applications required large software systems to be built.

Early experience in building large software systems shows that breathing methods of software development were not acceptable. Techniques applicable to small systems could not be scaled up. Major projects were sometimes years late. They cost much more than initially predicted, were undependable, difficult to preserve and performed poorly. Software development was in crisis. Hardware costs were plummeting while software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

After almost 30 years of development, we have made enormous progress in software engineering. Productivity improvements are hard to quantify but there is no doubt that our ability to produce efficient and dependable software has markedly improved. We have a much better understanding of the activities involved in software development. We have developed methods of software specification, design and implementation. New notations and tools reduce the effort required to produce large and complex systems. .

Nevertheless, many large software projects are still late and over-budget. Software is delivered and installed which does not meet the real needs of the customers buying that software. New technologies resulting from the convergence of computers and communication systems place new demands on software engineers. Software engineering has come far in its short lifetime; it still has far to go.

1.1 Concepts in Software Engineering

The goal of research in this area is to give a solid foundation for development and deployment of software for systems with stringent requirements for security, safety, functional correctness, and efficiency. Principal focuses are general programming methodology and program examination techniques. Generic programming is mainly an activity of "lifting" of specific computer code to a more widely useful level, while maintaining high principles of efficiency and other requisite properties. This process is aided by theoretical classification of software components according to thoroughly specified requirements. Results include the C++ Standard Template Library (STL), which is based on joint research with colleagues in industry; new generic sorting and searching algorithms; and new algorithm concept taxonomies. In the program analysis area, new techniques are being developed for different software tasks, including testing, understanding, and verification of object-oriented software. Results include new analyses that have been applied to testing of polymorphism in Java applications, and to testing of recovery code in highly reliable Web service applications. Future directions include greater integration of generic programming and program analysis methodology, with increased emphasis on proof-based approaches. The long range goal is to meet new challenges that arise from distributed software components, embedded system software updates, Web services, and other software for modern, pervasive computing.

There are two planned meanings of the word "Concepts" in our research area's name: the common one, and the more exact, technical sense of "concept" as a set of abstraction, such as abstract data types or algorithms, whose partisanship is defined by a list of requirements. In this technical intelligence, concepts are the main abstraction and organization instrument in generic programming. The key operation on concepts is refinement: incrementally addition requirements to a concept description, thereby plummeting the number of abstractions it contains but at the same time enabling more efficient algorithms or more refined analysis to be applied to the abstractions that remain. Repeated refinement with different choices of additional requirements results in a concept taxonomy or hierarchy, such as the one for STL.

Notes

Did u know? Programming languages started to appear in the 1950s and this was also another major step in abstraction and also assumed as the base of the software engineering.

1.2 Evolution and Impact

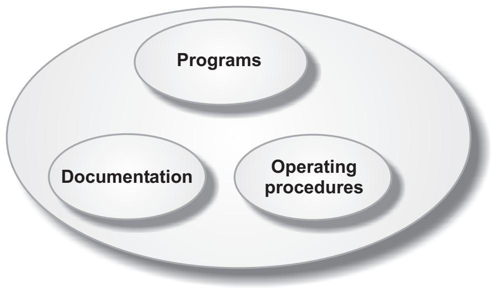
We temporarily review how the software engineering discipline has gradually come into existence, preliminary with a modest beginning about five decades ago. We also point out that in malice of all shortcoming of this regulation; it is the best gamble to apply for a solution to the software crisis.

1.2.1 Program versus Software

Software is in excess of programs. It consists of programs; documentation of any surface of the program and the procedures used to setup and operate the software system. The components of the software systems are shown in Figure. 1.1.

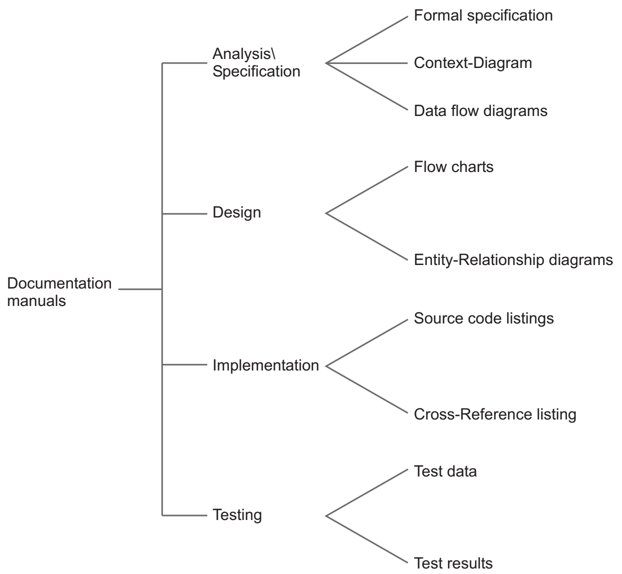
Software = Programs + Documentation + Operating procedures

Figure 1.1: Components of Software



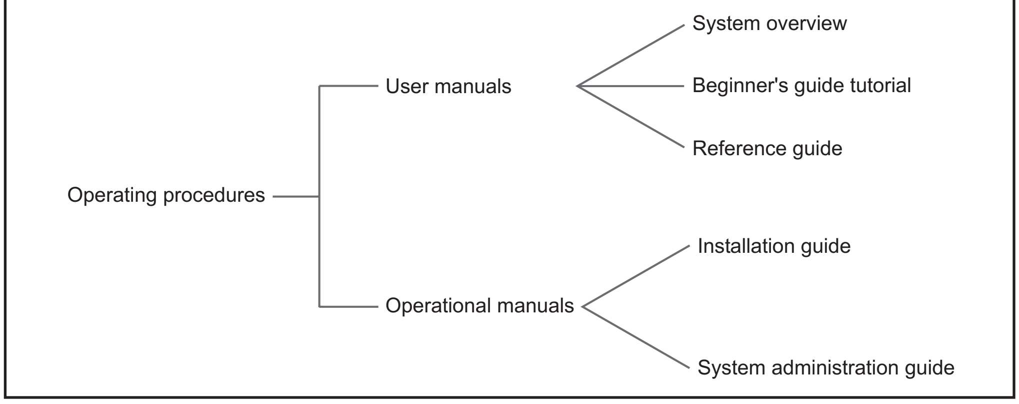
Any program is a subset of software and it becomes software only if documentation and functioning. Procedure manuals are ready. Program is a grouping of source code and object code. Documentation consists of different types of manuals as shown in Figure. 1.2.

Figure 1.2: List of Documentation Manuals



Operating procedures consist of instructions to setup and use the software system instructions on how to react to system failure. List of operating procedure manuals/documents is given in Figure 1.3.

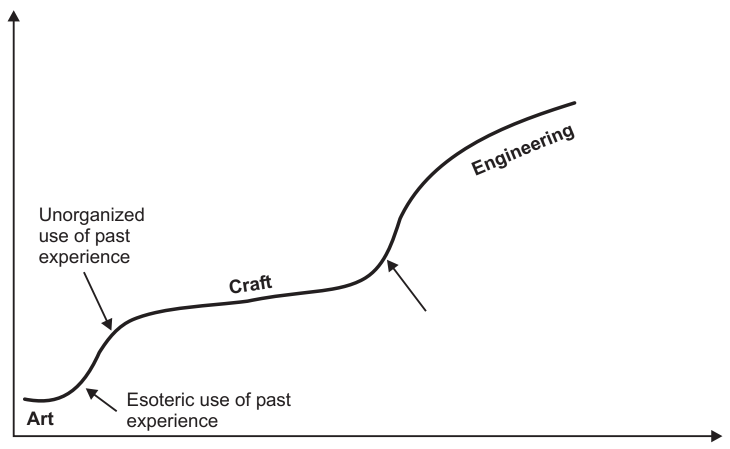
Figure 1.3: List of Operating Procedures Manuals



1.3 Evolution of an Art to an Engineering Discipline

Software engineering main beliefs have evolved over the past fifty years with contributions from frequent researchers and software professionals. The early programmers used an investigative programming style. In an investigative programming style, every programmer himself evolves his own software development techniques exclusively guided by his perception, experience, whims, and fancies. The investigative style of writing programs is also the one that is usually adopted by all students who do not have experience to software engineering we can believe the investigative program development style as an art-since, art is mostly guided by perception. There are many stories about programmers in the past who were like proficient artists and could write good programs based on some esoteric. In modern software industry, programmers rarely make use of such an esoteric knowledge. If we analyze the evolution of software development style over the past fifty we would see that it has changed from an esoteric art form to craft form, and has emerged as an engineering discipline. As a matter of fact, this growth pattern is not different from that of any other engineering discipline. Irrespective of the area, evolution of technology has followed strikingly similar patterns. A schematic representation of this technology development pattern is shown in Figure 1.4.

Figure 1.4: Technology Development with Time



Notes

Example: Let us consider the development of iron-making technology. In antique times, very few people knew about iron making. Those who knew about it kept it a closely-guarded secret. This esoteric knowledge used to get transferred from generation to generation as a family secret. Slowly, technology graduated from art to craft form, where tradesmen shared their knowledge with their apprentices and the knowledge pool continued to grow. Much later, through a systematic organization of knowledge, and incorporation of scientific basis, modern steel-making technology has evolved.

Critics point out that many of the methodologies and guidelines provided by the software engineering discipline lack scientific basis, are subjective, and are often inadequate. Yet, there is no denying the fact that adopting software engineering techniques facilitates development of high quality, software in a cost-effective and efficient manner. Software engineering practices have proven to be indispensable to the development of large software products-though exploratory styles can often be successfully used to develop small programs.

1.4 Software Characteristics

The software has a very particular characteristic e.g., “it does not exhaust”. Its performance and nature is quite diverse than other products of human life. A comparison with one such case, i.e., constructing a bridge in comparison with writing a program is given in Table 1.1. Both activities require different processes and have different characteristics.

Table 1.1: A Compression Between Constructing a Bridge and Writing a Program

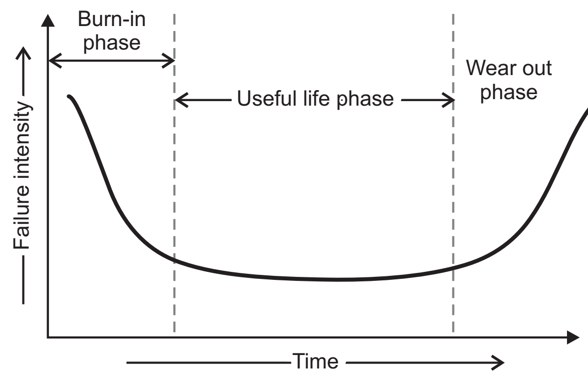
Sr. No.	Constructing a bridge	Writing a program
1.	The problem is well understood.	Only some parts of the problem are understood, others are not.
2.	There are many existing bridges.	Every program is different and designed for special applications.
3.	The requirements for a bridge typically do not change much during construction.	Requirements typically change during all phases of development.
4.	The strength and stability of a bridge can be calculated with reasonable precision.	Not possible to calculate correctness of a program with existing methods.
5.	When a bridge collapses, there is a detailed investigation and report.	When a program fails, the reasons are often unavailable or even deliberately concealed.
6.	Engineers have been constructing bridges for thousands of years.	Developers have been writing programs for 50 years or so.
7.	Materials (wood, stone, iron, steel) and techniques (making joints in wood, carving stone, casting iron) change slowly.	Hardware and software changes rapidly.

Some of the important characteristics are discussed:

1.4.1 Software does not Wear out

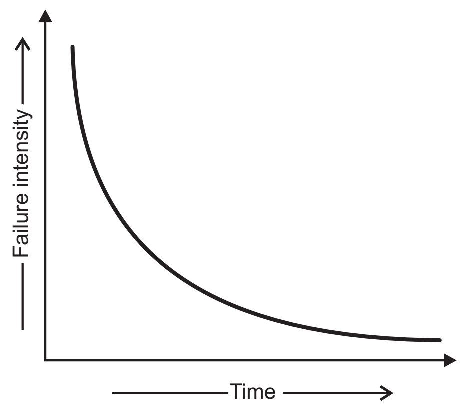
There is a renowned "bath tub curve" in dependability studies for hardware products. The curve is given in Figure 1.5. The shape of the curve is like "bath tub"; and is known as bath tub curve.

Figure 1.5: Bath Tub Curve



There are three phases for the life of a hardware product. First phase is burn-in phase, where failure intensity is high. It is expected to test the product in the industry before delivery. Outstanding to testing and fixing faults, failure strength will come down initial and may become stable after positive time. The second phase is the useful life phase where failure strength is just about constant and is called useful life of a product. After few years, again failure intensity will increase due to wearing out of components. This phase is called wear out phase. We do not have this phase for the software as it does not wear out. The curve for software is given in Figure 1.6.

Figure 1.6: Software Curve



Important point is software becomes dependable overtime rather than wearing out. It becomes outdated, if the environment for which it was urbanized, changes. Hence software may be retired due to environmental changes, new requirements, new prospect, etc.

1.4.2 Software is not Manufactured

The life of software is from concept exploration to the retirement of the software product. It is one time development effort and continuous maintenance effort in order to keep it operational. However, making 1000 copies is not an issue and it does not involve any cost. In case of hardware product, every product costs us due to raw material and other processing expenses. We do not have assembly line in software development. Hence it is not manufactured in the classical sense.

1.4.3 Reusability of Components

If we have to manufacture a TV, we may purchase picture tube from one vendor, cabinet from another, design card from third and other electronic components from fourth vendor. We will assemble every part and test the product thoroughly to produce a good quality TV. We may be required to manufacture only a few components or no component at all. We purchase every unit and component from the market and produce the finished product. We may have standard quality guidelines and effective processes to produce a good quality product. In software, every project is a new project. We start from the scratch and design every unit of the software product. Huge effort is required to develop software which further increases the cost of the software product. However, effort has been made to design standard components that may be used in new projects. Software reusability has introduced another area and is known as component based software engineering. Hence developers can concentrate on truly innovative elements of design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In software, there is only a humble beginning like graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.

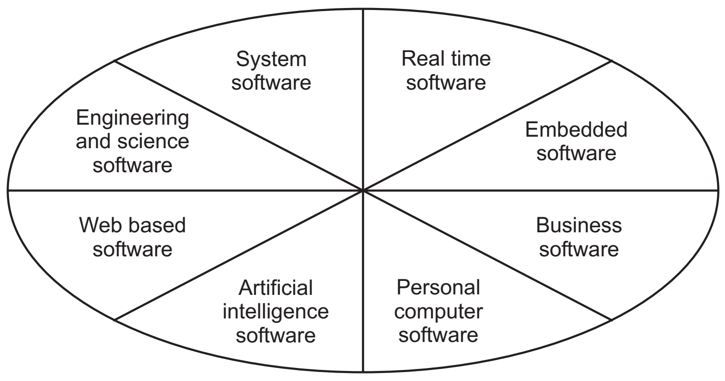
1.4.4 Software is Flexible

We all feel that software is flexible. A program can be urbanized to do almost anything. Sometimes, this feature may be the best and may help us to accommodate any kind of change. However, most of the times, this “almost anything” characteristic has made software development difficult to plan, monitor and control. This unpredictability is the basis of what has been referred to for the past 30 years as the “Software Crisis”.

1.5 The Changing Nature of Software

Software has become integral part of most of the fields of human life. We name a field and we find the usage of software in that field. Software applications are grouped in to eight areas for expediency as shown in Figure. 1.7.

Figure 1.7: Software Application


System software
Real time software
Engineering and science software
Embedded software
Web based software
Business software
Artificial intelligence software
Personal computer software

1.5.1 System Software

Infrastructure software comes under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a set of programs to provide service to other programs.

1.5.2 Real Time Software

This software is used to monitor, control and analyze real world proceedings as they occur. An example may be software required for weather forecasting. Such software will gather and process the position of temperature, humidity and other environmental parameters to forecast the weather.

1.5.3 Embedded Software

This type of software is located in "Read-Only-Memory (ROM)"of the product and controls the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The entrenched software handles hardware components and is also termed as intelligent software.

1.5.4 Business Software

This is the largest application area. The software designed to procedure business applications is called business software. Business software could be payroll, file monitoring system, employee management, and account management. It may also be a data warehousing tool which helps us to take decisions based on obtainable data. Management information system, enterprise resource planning (ERP) and such other software are accepted examples of business software.

1.5.5 Personal Computer Software

The software used in personal computers is roofed in this category. Examples are word processors, computer graphics, multimedia and animating, Business software, Embedded software, Real time software, System software, Engineering and technical software, Web based software Artificial intelligence software, Personal computer software tools, database management, computer games etc. This is a very upcoming area and many large organizations are concentrating their attempt here due to great customer base.

1.5.6 Artificial Intelligence Software

Artificial Intelligence software makes use of no numerical algorithms to resolve multifaceted problems that are not amenable to computation or straight forward analysis [PRESOI]. Examples are expert systems, artificial neural network, signal processing software etc.

1.5.7 Web Based Software

The software connected to web applications comes under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

1.5.8 Engineering and Scientific Software

Technical and engineering application software is grouped in this category. Enormous computing is usually required to process data. Examples are CAD/CAM package, SPSS, MATLAB, Engineering Pro, Circuit analyzers etc. The expectations from software are increasing in modern civilization. Software of any of the above groups has a specialized role to play. Customers and development organizations desire more features which may not be always possible to provide. Another trend has emerged to provide source code to the customers and organizations so that they can make modifications for their needs. This trend is particularly visible in infrastructure software like data bases, operating systems, compilers etc. Software where source codes are available, are known as open source. Organizations can develop software applications around such source codes. Some of the examples of "open source software" are LINUX, MySQL, PHP, open office, Apache web server etc. Open source software has risen to great prominence. We may say that these are the programs whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program without paying royalties to original developers. Whether open

Notes

source software is better than proprietary software? Answer is not easy. Both schools of thought are in the market. However, popularity of many open source software give confidence to every user. They may also help us to develop small business applications at low cost.

1.6 Software Components

As an engineering regulation evolves a compilation of standard design mechanism is shaped. Standard screws and off-the –shelf included circuits are only two of thousands of standard mechanism that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design (i.e., the parts of the design that represent something new). In the hardware world, component reuse is a natural part of the engineering process. In the software world it is something that has yet to be achieved on a broad scale.

Reusability is an important characteristic of a high quality software component. A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well-defined algorithms in an effective manner, but had a limited domain of application. Today we have extended our view of reuse components encapsulate both data and the processing that is applied to the data enabling the software engineer to create new application from reusable parts. For example today's interactive interfaces are built using reusable components that enable the creation of graphics windows pull-down menus and a wide variety of interaction mechanism. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software components are built using a programming language that has a limited vocabulary an explicitly defined grammar and well formed rules of syntax and semantics. At the lowest level the language mirrors the instruction set of the hardware. At mid-level programming languages such as Ada 95, C or Smalltalk are used to create a procedural description of the program. A highest level the language uses graphical icons or other symbology to represent the requirements for a solution. Executable instructions are automatically generated.

Machine level language symbolic representation of the CPU instruction set. When a good software developer produces a maintainable well documented program machine level language can made extremely efficient use of memory and "optimize" program execution speed. When a program is poorly designed and has little documentation machine language is a nightmare.

Mid-level languages allow the software developer and the program to be machine-independent. When a more sophisticated translator is used, the vocabulary, grammar, syntax and semantics of a mid-level language can be such more sophisticated that machine-level languages. In fact mid-level language compilers ad interpreters produce machine-level language as output.

Although hundreds of programming languages are in use today fewer than ten mid-level programming languages are widely used in the industry. Languages such as COBOL and FORTRAN remain in widespread use more than 30 years after their introduction. More modern programming languages such as Ada95, C, C++, Eiffel, Java and Smalltalk have each gained an enthusiastic following.

Machine code assembly languages and mid-level programming languages are often referred to as the first three generation of computer languages. With any of these languages the programmer must be concerned both with the specification of the information structure and the control of the program itself. Hence languages in the first three generation are termed procedural languages.

Fourth generation languages also called nonprocedural languages move the software developer even further from the computer hardware. Rather than requiring the developer to specify procedural detail, the nonprocedural language implies a program by "specifying the desired result rather than specifying action required to achieve that result". Support software translates the specification of result into a machine executable program.

1.7 Software Applications

Software may be practical in any state of affairs for which a pre-specified set of technical steps (i.e., an algorithm) has been defined (notable exceptions to this rule are specialist systems and artificial neural network software). Information satisfied and determinacy are significant factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information .For example many business applications make use of highly structured input data and produce formatted "reports" Software that controls an automated machine (e.g., numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

1.7.1 Information Determinacy

It refers to the inevitability of the order and timing of information. An engineering analysis program accepts data that have a predefined order executes the analysis algorithm without break and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system on the other hand accepts inputs that have varied content and arbitrary timing executes algorithms that can be interrupted by external conditions and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate. It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

1.7.2 System Software

System software is a collection of programs written to service other programs. Some system software (e.g., compiler s editors and file management utilities) processes complex but determinate information structures. Other systems application (e.g., operating system components driver's telecommunications processors) process largely indeterminate data. In either case the systems software area is branded by heavy interaction with computer hardware heavy usage by multiple users; simultaneous operation that requires scheduling resource sharing and sophisticated process management; complex data structures and multiple external interfaces.

1.7.3 Real-Time Software

Programs that monitor/analyze/ control genuine world proceedings as they occur are called concurrent software. Elements of concurrent software comprise a data assembly component that collects and formats information from external surroundings an analysis component that transforms information as necessary by the application a control / output constituent that responds to the external environment so that concurrent answer (typically ranging from 1 millisecond to 1 minute) can be maintained. It should be noted that the term "real-time" differs from "interactive" or timesharing". A real-time system must respond within strict time constraints. The response time of an interactive (or time-sharing) system can normally be exceeded without disastrous results.

1.7.4 Business Software

Business information dispensation is the major single software application area. Separate "systems" (e.g., payroll accounts receivable/payable inventory, etc.,) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area reorganize existing data in a way that facilitates business operation or management decision making. In addition to conventional data dispensation applications, business software applications also encompass interactive and client/server computing (e.g., point-of scale transaction processing).

1.7.5 Embedded Software

Intelligent products have become ordinary in nearly every consumer and industrial market. Embedded software resides in read only memory and is accustomed to control products and systems for the consumer and industrial markets. Embedded software can perform very imperfect and mysterious functions (e.g., digital functions in an automobile such as fuel control, dashboard displays, braking systems, etc.).

1.7.6 Personal Computer Software

The personal computer software market has burgeoned over the past decade. Word processing, spreadsheets, computer graphics, multimedia entertainment, database management personal and business financial applications and external network or database right of entry are only a small number of hundreds of application.

1.7.7 Artificial Intelligence Software

Artificial Intelligence (AI) software makes use of non numerical algorithms to resolve multifaceted problems that are not agreeable to calculation or straight forward analysis. An active AI area is expert systems also called knowledge-based systems. However other application areas for AI software are pattern acknowledgment (image and voice) theorem proving and game playing. In recent years a new branch of AI software called artificial neural networks, has evolved. A neural network simulates the structure of brain processes (the functions of the biological neuron) and may in the end lead to a new class of software that can be familiar with complex patterns and learn from past knowledge.

Did u know? The first theory about software was proposed by Alan Turing in his 1935 essay Computable numbers with an application to the Entscheidungs problem (Decision problem). The term "software" was first used in print by John W. Tukey in 1958.

Self Assessment Questions

1. Generic programming is largely an activity of "lifting" of specific computer code to a more widely useful level, while maintaining high standards of efficiency and other required properties.

   (*a*) True (*b*) False

2. The components of the software systems are:

   (*a*) Maintaining (*b*) Development

   (*c*) Coding (*d*) Programs

3. The life of software is from concept exploration to the retirement of the ....

   (*a*) software applications (*b*) software components

   (*c*) software product (*d*) software development

4. Huge effort is required to develop software which further increases the cost of the ....

   (a) business software (*b*) real-time software

   (*c*) software product (*d*) system software

5. .... is a collection of programs written to service other programs.

   (*a*) Information determinacy (*b*) System software

   (*c*) Real-time software (*d*) Business software

1.8 Software Myths

There are number of myths connected with software growth neighbourhood. Some of them really affect the way, in which software development should take place. In this, we list few myths, and discuss their applicability to standard software development [PIER99, LEVE95].

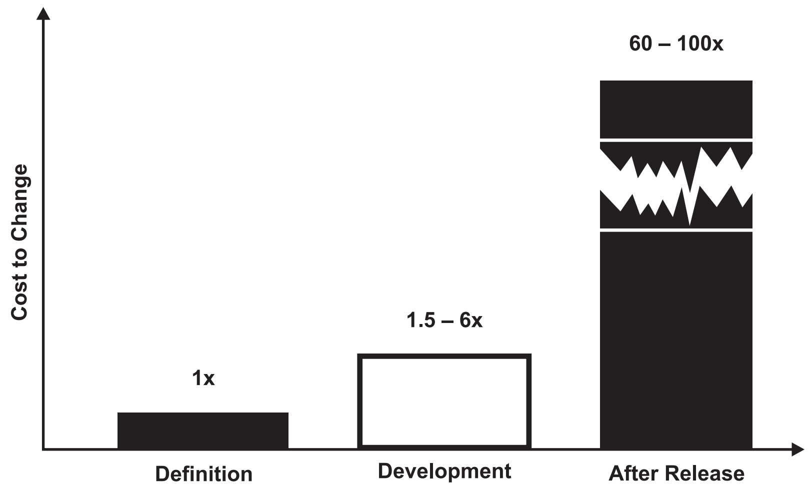
- *Software is easy to change*: It is true that source code files are easy to edit, but that is quite dissimilar than saying that software is easy to change. This is misleading exactly because source code is so easy to alter. But making changes without introducing errors is extremely difficult, predominantly in organizations with poor process prime of life. Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.
- *Computers provide greater reliability than the devices they replace*: It is true that software does not fail in the conventional sense. There are no restrictions to how many times a given piece of code can be executed before it "wears out". In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have been computerized. Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.
- *Testing software or 'proving' software correct can remove all the errors*: Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.
- *Reusing software increases safety*: This myth is particularly troubling because of the false sense of security that code re-uses can create. Code re-use is a very powerful tool that can yield dramatic improvement in development efficiency, but it still requires analysis to determine its suitability and testing to determine if it works.
- *Software can work right the first time*: If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote us a price. If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineers are often asked to do precisely this sort of work, and they often accept the job.
- *Software can be designed thoroughly enough to avoid most integration problems*: There is an old saying among software designers: "Too bad, there is no complier for specifications" This points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these. Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.
- *Software with more features is better software*: This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.
- *Addition of more software engineers will make up the delay*: This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

- *Aim is to develop working programs*: The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community. This list is endless. These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. In addition, following are the contributing factors:
  - (a) Change in ratio of hardware to software costs
  - (b) Increasing importance of maintenance
  - (c) Advances in software techniques
  - (d) Increased demand for software
  - (e) Demand for larger and more complex software systems.

1.9 Software a Crisis on the Horizon

Lots of industry observers have branded the exertion connected with software development as a "crisis" Yet what we really have may be something somewhat dissimilar. The word "crisis is defined in Webster's Dictionary as a crossroads in the course of no matter which: important or crucial time stage or event" Yet for software there has been no "defining moment" no "decisive time" only slow evolutionary change. In the software industry we have had a "crisis" that has been with us for close to 30 years and that is a contradiction in terms. Anyone who looks up the word "crisis" in the dictionary will find another definition: "the turning point in the course of a disease when it becomes clear whether the patient will live or die". This definition may give us a clue about the real nature of the problems that have plagued software development. We have yet to reach the stage of crisis in computer software. What we really have is a chronic affliction. The word "affliction is defined as anything causing pain or distress" But it is the definition of the adjective "chronic" that is the key to our argument: "lasting a long time or recurring often; continuing indefinitely". It is far more accurate to describe what we have endured for the past three decades as a chronic affliction rather that a crisis. There are no miracle cures, but there are many ways that we can reduce that pain as we strive to discover a cure (See in Figure 1.8).

Figure 1.8: The Impact of Change



Whether we call it a software crisis or a software affliction the term alludes to asset or problems that are encountered in the development of computer software. The problems are not limited to software that "does not function properly" Rather, the affliction encompasses problems associated with how we develop software how we maintain a growing volume of existing software and how we can expect to keep pace with a growing demand for more software. Although reference to a crisis or even an affliction can be criticized for being melodramatic the phrases do serve a useful purpose by denoting real problems that are encountered in all area of software development.

Caution Changing a system's software configuration is not a trivial event, particularly in the case of a system used, for emergency notification. A critical component could fail to operate due to incorrect programming.

1.10 Terminologies in Software Engineering

Some terminologies are discussed in this part which is commonly used in the field of Software Engineering.

1.10.1 Product and Process

What is delivered to the customer is called a product. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.

The software products are divided in two categories:

(i) Generic products

(ii) Customized products.

Generic products are urbanized for anonymous customers. The aim is generally the entire world and many copies are expected to be sold. Infrastructure software like functioning systems, compilers, analyses, word processors, CASE tools etc. are covered in this category. The customized products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the developments projects (say about 80%) come under this category.

Until the 1980s, the vast majority of software systems which were sold were bespoke specially designed systems which ran on great computers. They were costly because all the development cost had to be met by a single client.

Since the development of personal computers, this situation has completely changed. The personal computer market is totally dominated by software products produced by companies such as Microsoft. These account for the vast majority of software sales. These are usually relatively cheap because their development cost is spread across hundreds or thousands of different customers.

However, there is still a large market for specially designed systems. Hardware control always requires some kind of special-purpose system. As computers are embedded in more and more devices, there is an increasing demand for software controllers. Consequently, most software development effort is still probably devoted to producing bespoke systems rather than shrink-wrapped software products.

1.10.2 Software Product Attributes

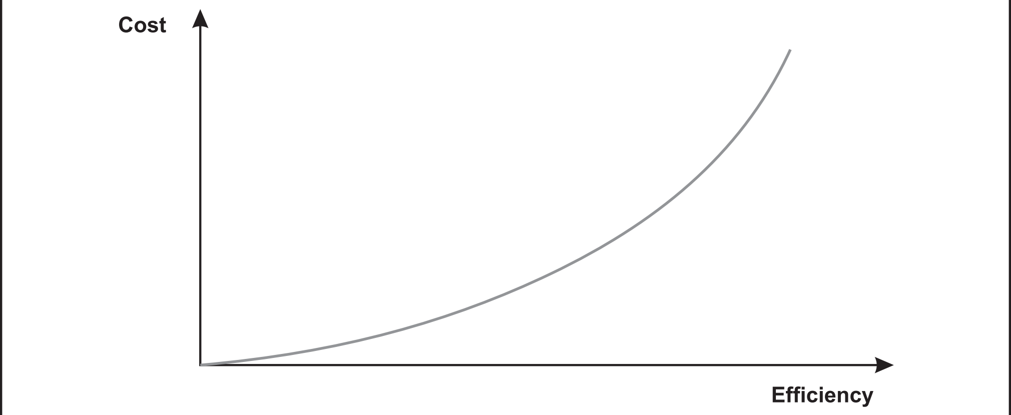
In the vein of all engineering, software engineering is not almost producing products but involves producing products in a gainful way. Agreed limitless possessions, the mainstream of software problems can almost certainly be solved. The challenge for software engineers is to produce high-quality software with a finite amount of resources and to a predicted schedule.

The attributes of a software product are the characteristics displayed by the product once it is installed and put into use. These are not the services provided by the product. Rather, they are concerned with the product's dynamic behaviour and the use made of the product. Examples of these attributes are therefore efficiency, reliability, maintainability, robustness, portability, and so on.

The relative importance of these characteristics obviously varies from system to system. However, optimizing all of these attributes is difficult as some are exclusive. For example, providing a better user interface may reduce system efficiency. All are subject to the law of diminishing returns. The relationship between cost and improvements in each of these attributes is not a linear one. Small improvements in any of the attributes can be expensive. A great deal of effort must be devoted to optimizing a particular attribute.

As an example of this, Figure 1.9 shows how costs may rise exponentially as efficiency improvements are required.

Figure 1.9: Costs vs. Efficiency



For some kinds of system, such as avionic systems, efficiency is a prime consideration. The software may have to run on a computer where weight and size considerations restrict the power of the hardware which can be used. It may have to run in a relatively small memory with no associated disks. It may be necessary to optimize efficiency at the expense of the other system attributes. Where this is necessary, the design decisions for efficiency should be explicit and the consequences of these decisions carefully analyzed.

1.10.3 Process and its Approach

The software process is the set of activities and associated results which produce a software product. These activities are mostly carried out by software engineers. CASE (computer-aided software engineering) tools may be used to help with some procedure activities.

There are four fundamental process performances which are common to all software processes. The activities are:

- *Software Specification*: The functionality of the software and constraints on its operation must be defined.
- *Software Development*: The software to meet the specification must be produced.
- *Software Validation*: The software must be validated to ensure that it does what the customer wants.
- *Software Evolution*: The software must evolve to meet changing customer needs.

Notes

There is no such obsession as a right or a wrong software process. Dissimilar software processes crumble these performances in different ways. The timing of the performance varies as does the results of each activity different organizations use-different processes to produce the same type of product. Different types of product may be produced by an organization using different processes. However, some processes are more suitable than others for some types of application. If the wrong process is used, this will probably reduce the quality or the usefulness of the software product to be developed.

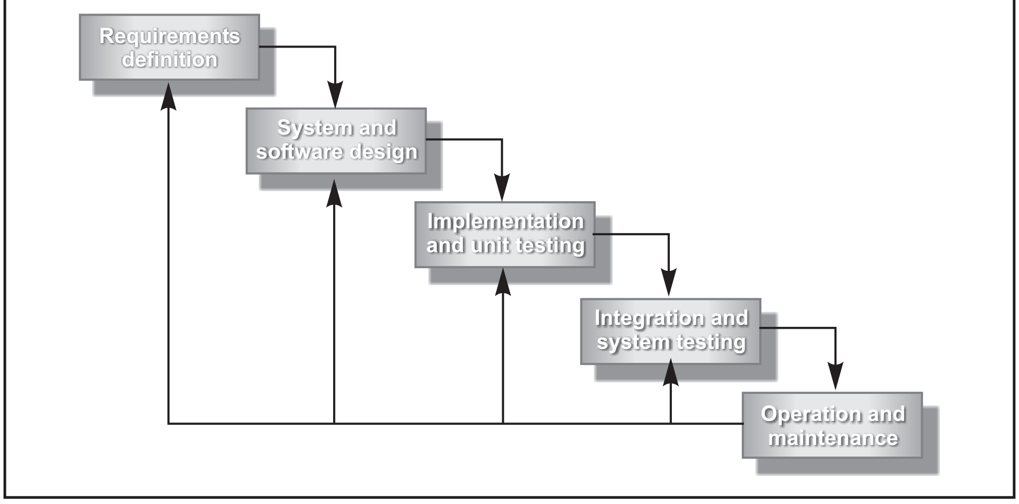
Because there is a variety of different process models used, it is impossible to produce reliable figures for cost distribution across these activities. However, we do know that modifying software usually takes up more than 60% of total software costs. This percentage is increasing as more and more software is produced and has to be maintained. Designing software for change is therefore essential. Software processes (like most business processes) are complex and involve a very large number of activities. Like products, processes also have attributes or characteristics it is not possible to optimize all process attributes simultaneously. For example: plea, if a rapid development process is required then it may be necessary to reduce the process visibility making a process visible means producing documents at regular intervals. This will slow down the process.

Detailed software process models are still the subject of research but it is now clear that there are a number of different general models or paradigms of software development:

The waterfall Approach

The first explicit model of the software growth process was unoriginal from other engineering processes. This was passionately accepted by software project management. It offered a means of making the expansion process more visible. Because of the cascade from one phase to another, this model is known as the 'waterfall model' (Figure 1.10).development goes on to the following stage.

Figure 1.10: The Software Life Cycle


Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

There are frequent variations of this process model (which is sometimes called the software life cycle). The principal stages of the model map onto the primary development activities:

- *Requirements analysis and definition*: The system's services, constraints and goals are established by consultation with system users. They are then defined in a manner which is understandable by both users and development staff.

- *System and software design*: The systems design process partitions the requirements to either hardware or software systems. It establishes overall system structural design. Software design involves representing the software system functions in a form- that may be transformed into one or more executable programs.
- *Implementation and unit testing*: during this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
- *Integration and system testing*: The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
- *Operation and maintenance*: Normally (although not necessarily) this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

Actually, these stages extend beyond and feed information to each other. During design, problems with requirements are identified; during coding, design problems are found and so on. The software process is not a simple linear model but involves a sequence of iterations of the development activities.

During the final life cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered. Program and design errors emerge and the need for new functionality is identified. Modifications become necessary for the software to remain useful.

(1) *Evolutionary Development*

This moves towards interleaves the performance of specification, growth and validation. An initial system is quickly developed from very abstract specifications. This is then refined with customer input to produce a system which satisfies the customer's needs. The system may then be delivered. Alternatively, it may be re-implemented using a more structured approach to produce a more robust and maintainable system.

(2) *Formal Transformation*

This approach is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods, to a program. These transformation are correctness preserving. This means that you can be sure that the developed program meets its specification.

(3) *System assembly From Reusable Components*

This technique assumes that parts of the system already exist. The system development process focuses on integrating these parts rather than developing them from scratch.

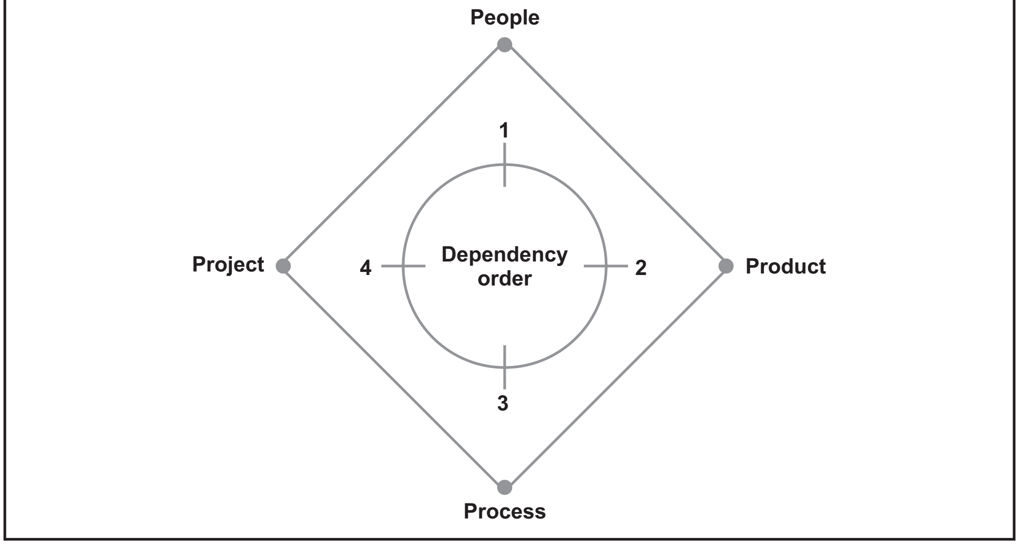
The first two of these approaches, namely the waterfall approach and evolutionary development, are now widely used for practical systems development. Some systems have been built using correctness-preserving transformations but this is still an experimental.

1.11 Role of Management in Software Development

The management of software development is heavily dependent on four factors: People, Product, Process, and Project. Order of dependency is as shown in Figure. 1.11.

Notes

Figure 1.11: Role of Management in Software Development



Software development is a people centric activity. Hence, success of the project is on the shoulders of the people who are involved in the development.

1.11.1 The People

Software development requires good managers. The managers, who can understand the psychology of people and provide good leadership. A good manager cannot ensure the success of the project, but can increase the probability of success. The areas to be given priority are: proper selection, training, compensation, career development, work culture etc. Managers face challenges. It requires mental toughness to endure inner pain. We need to plan for the best, be prepared for the worst, expect surprises, but continue to move forward anyway. Charles Maurice once rightly said "I am more afraid of an army of one hundred sheep led by a lion than an army of one hundred lions led by a sheep". Hence, manager selection is most crucial and critical after having a good manager, project is in safe hands. It is the responsibility of a manager to manage, motivate, encourage, guide and control the people of his/her team.

1.11.2 The Product

What do we want to deliver to the customer? Obviously a product a solution to his/her problems. Hence, objectives and scope of work should be defined clearly to understand the requirements. Alternate solutions should be discussed. It may help the managers to select a "best" approach within constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces etc. Without well defined requirements, it may be impossible to define reasonable estimates of the cost, development time and schedule for the project.

1.11.3 The Project

A proper planning is required to monitor the status of development and to control the complexity. Most of the projects are coming late with cost overruns of more than 100%. In order to manage a successful project, we must understand what can go wrong and how to do it right. We should define concrete requirements (although very difficult) and freeze these requirements. Changes should not be incorporated to avoid software surprises. Software surprises are always risky and we should minimize them. We should have a planning mechanism to give warning before the occurrence of any surprise. All four factors (People, Product, Process and Project) are important for the success of the project. Their relative importance helps us to organize development activities

Notes

in more scientific and professional way. Software Engineering has become very important discipline of study, practice and research. All are working hard to minimize the problems and to meet the objective of developing good quality maintainable software that is delivered on time, within budget, and also satisfies the requirements. With all cries and dissatisfaction, discipline is improving and maturing day by day. New solutions are being provided in the niche areas and encouraging results are being observed. We do feel that within couple of years, situation is bound to improve and software engineering shall be a stable and mature discipline.

Case Study

Galaxy Mining Company Ltd.

A mining company named Galaxy Mining Company Ltd. (GMC) has mines located at various places in India. It has about fifty different mine sites spread across eight states. The company employs a large number of mines at each mine site. Mining being a risky profession, the company intends to operate a special provident fund, which would exist in addition to the standard provident fund that the miners already enjoy. The main objective of having the special provident fund (SPF) would be quickly distributing some compensation before the standard provident amount is paid. According to this scheme, each mine site would deduct SPF instalments from each miner every month and deposit the same with the Central Special Provident Fund Commissioner (CSPFC). The CSPFC will maintain all details regarding the SPF instalments collected from the miners. The GMC employed a reputed software vendor Adventure Software Inc. to undertake the task of developing the software for automating the maintenance of SPF records of all employees. The GMC realized that besides saving manpower on bookkeeping work, the software would help in speedy settlement of claim cases. The GMC indicated that the amount it can afford for this software to be developed and installed is Rs. 1 million. Adventure Software Inc. deputed their project manager to carry out the feasibility study. The project manager discussed the matter with the top managers of GMC to get an overview of the project. He also this cussed the issues involved with the several field PF officers at various mine sites to determine the exact details of the project. The project manager identified two broad approaches to solve the problem. One was to have a central database which could be accessed and updated via as satellite connection to various mine sites. The other approach was to have local databases at each mine site and to update the central database periodically through a dial-up connection. These periodic updates could be done on a daily or hourly basis depending on the delay acceptable to GMC in invoking various functions of the software. The project manager found that the second approach was very affordable and more fault-tolerant as the local mine sites could still operate even when the communication link to the central database temporarily failed. The project manager quickly analyzed the database functionalities required, the user-interface issues, and the software handling communication with the mine sites. He arrived at a cost to develop from the analysis. He found that the solution involving maintenance of local databases at the mine sites and periodic updating of a central database was financially and technically feasible. The project manager discussed his solution with the GMC management and found that the solution was acceptable to them as well.

Activities Under Taken During Case Study

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of two distinct activities, namely Requirements gathering and analysis, and Requirements specification. The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to

Contd...

clearly understand the customer requirements so that incompleteness and inconsistencies are removed. The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the non functional requirements, and the goals of implementation.

Questions

1. What approach is used in this case study?
2. Describe water fall method and explain that approach satisfy this case study?

Self Assessment Questions

6. CASE Tool is ……….
   (*a*) Computer Aided Software Engineering
   (*b*) Component Aided Software Engineering
   (*c*) Constructive Aided Software Engineering
   (*d*) All of these.
7. Software consists of …………
   (*a*) Set of instructions + operating procedures
   (*b*) Programs + documentation + operating procedures
   (*c*) Programs + hardware manuals
   (*d*) Set of programs
8. Which is the most important feature of spiral model?
   (*a*) Quality management (*b*) Risk management
   (*c*) Performance management (*d*) Efficiency management
9. Which phase is not available in software life cycle?
   (*a*) Coding (*b*) Testing
   (*c*) Maintenance (*d*) Abstraction
10. Which is not a step of requirement engineering?
   (*a*) Requirements elicitation (*b*) Requirements analysis
   (*c*) Requirements design (*d*) Requirements documentation

11. The model in which the requirements are implemented by category is…………….
    (*a*) evolutionary development model (*b*) waterfall model
    (*c*) prototyping (*d*) iterative enhancement model
12. For a well understood data processing application it is best to use……….
    (*a*) waterfall model (*b*) prototyping model
    (*c*) evolutionary model (*d*) spiral model
13. The level at which the software uses scarce resources is:
    (*a*) Reliability (*b*) Efficiency
    (*c*) Portability (*d*) All of these.
14. If every requirement can be checked by a cost-effective process, then the SRS is……….
    (*a*) verifiable (*b*) traceable
    (*c*) modifiable (*d*) complete
15. Modifying the software to match changes in the ever changing environment is called:
    (*a*) adaptive maintenance (*b*) corrective maintenance
    (*c*) perfective maintenance (*d*) preventive maintenance

1.12 Summary

- Software engineers are model parts of the real world in software. These models are large, abstract and complex so they must be made visible in documents such as system designs, user manuals, and so on.
- Product is delivered to the customer. It may include source code, specification document, manuals, documentation etc. Basically, it is nothing but a set of deliverables only.
- Generic products are developed for anonymous customers. The target is generally the entire world and many copies are expected to be sold. Infrastructure software like operating systems, compilers, analyses, word processors.
- Customized products are developed for particular customers. The specific product is designed and developed as per customer requirements. Most of the developments projects (say about 80%) come under this category.
- Formal transformation approach is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods, to a program.

1.13 Keywords

Evolution Development Model: The evolutionary development model of the software process treats specification, development and validation as concurrent activities.

Software Engineering: Software engineering is concerned with methods, tools, and techniques for developing and managing the process of creating and evolving software products.

Software Process: The software process consists of activities which are involved in developing software products. Basic activities are software specification, development, validation and evolution.

Software Products: Software products consist of developed programs and associated documentation. Essential product attributes are maintainability, dependability, efficiency and usability.

Waterfall Model: The waterfall model of the software process considers each process activity as a separate and discrete phase.

Lab Exercise

1. Create a flowchart for the working of Waterfall model.
2. Write the steps for software planning.

1.14 Review Questions

1. List the reasons for the "software crisis"? Why are CASE tools not normally able to control it.
2. "The software crisis is aggravated by the progress in hardware technology?" Explain with examples.
3. What is the significance of software crisis in reference to software engineering discipline?
4. How are software myths affecting software process? Explain with the help of examples.
5. State the difference between program and software. Why have documents and documentation become very important?
6. What is software engineering? Is it an art, craft or a science? Discuss.
7. What is the aim of software engineering? What does the discipline of software engineering discuss?
8. Define the term "Software Engineering". Explain the major differences between software engineering and other traditional engineering disciplines.
9. What is software process? Why is it difficult to improve it?
10. Describe the characteristics of software contrasting it with the characteristics of hardware.

Answers for Self Assessment Questions

1.	(*a*)	2.	(*d*)	3.	(*c*)	4.	(*c*)	5.	(*b*)
6.	(*a*)	7.	(*b*)	8.	(*b*)	9.	(*d*)	10.	(*c*)
11.	(*a*)	12.	(*a*)	13.	(*b*)	14.	(*a*)	15.	(*a*)

1.15 Further Readings

Books

Humphrey W.S., "Managing the Software Process", by Addison-Wesley Pub. Co.
Schach, Stephen, "Software Engineering", Vanderbilt University

Online link

http://books.google.co.in/books?id=JiIiAQAAIAAJ&q=introduction+of+software+engineering&dq=introduction+of+software+engineering&hl=en&sa=X&ei=C5oHUMHNNITZrQeWy5DxAg&.

Deepak Mehta, Lovely Professional University

Unit 2: Software Processes and Models

CONTENTS

Sarabjit Kumar, Lovely Professional University

Objectives

After studying this unit, you will be able to:

- Explain the processes and models
- Define the characteristics of software model
- Discuss the waterfall model
- Explain the prototype
- Discuss the iterative
- Explain the time boxing
- Discuss the comparison

Introduction

The software process is a significant factor for delivering excellence software systems, as it aims to manage and change the user need into a software product that meets this need. In this context, software process means the set of activities required to produce a software system, executed by a group of people organized according to a given organizational structure and counting on the support of techno-conceptual tools. Software process modelling describes the creation of software development process models. A software process model is an abstract representation of the architecture, design or definition of the software process each representation describes, at different detail levels, an organization of the elements of a finished, ongoing or proposed process and it provides a definition of the process to be used for evaluation and improvement. A process model can be an aliased, validated and simulated, if executable. The process models are used mainly for software process control (evaluation and improvement) in an organization, but they can be also used for experimenting with software process theory and to ease process automation.

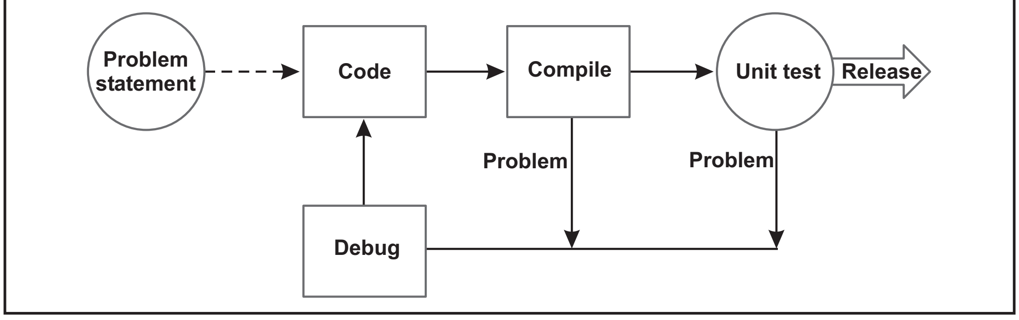
2.1 Processes and Models

On the contrary to software life cycle models, software process models often stand for a networked Sequence of activities, objects, transformations, and events that exemplify strategies for accomplishing software development. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Software process networks can be viewed as representing multiple interconnected Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. On-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard. Wino grad and others have referred to these units uncooperative work between people and computers as "structured discourses of work while task chains have become popularized under the name of "workflow" Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order. For example, a task chain for the activity of object-oriented software design might include the following task actions:

- Develop an informal narrative specification of the system.
- Identify the objects and their attributes.
- Identify the operations on the objects.
- Identify the interfaces between objects, attributes, or operations.
- Implement the operations.

Circumstances or idiosyncratic clearly, this sequence of actions could entail multiple iterations and non-procedural primitive action invocations in the course of incrementally progressing toward an object-oriented software design. Task chains join or split into other task chains resulting in an overall production network or web. The production web represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems. The production lattice therefore structures how software system is developed, used, and maintained. However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible foul-ups that can emerge in the real world of software development.

Figure 2.1: Processes and Models



Therefore, any software manufacture web will in some way understand only an approximate or incomplete description of software development. Articulation work is a kind of surprising task that is performed when a planned task chain is inadequate or breaks down. It is work that represents an open-ended non-deterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain thus, descriptive task chains are employed to characterize the observed course of events and situations that emerge when people try to follow a planned task sequence. Articulation work in the context of software evolution includes actions people take that entail either their accommodation to the contingent or anomalous behaviour of a software system, or negotiation with others who may be able to affect a system modification or otherwise alter current circumstances this notion of articulation work has also been referred to as software process dynamism.

Task Design a structure for software project model.

2.2 Characteristics of Software Model

Software disaster leads to the term "Software Engineering." Software engineering is the limb of software that deals with development of well urbanized software that satisfies all the users' requirements and ensures that the software is provided on time and within financial plan the software is provided along with documentation that gives explanation to the user on how to use the software. Software engineering can be defined as: "Software engineering is the branch of engineering in which the software is developed in a manner that it will have all the properties like maintenance, extensibility and is within time and budget and also satisfies all the requirements given by user." Producing software does not mean producing just software but it means to develop the software in a cost effective manner. Characteristics of well engineered software are:

Efficiency: Software is said to be well-organized if it uses the available resources in the most efficient manner. The software should be able to offer a quick response in the least processing time using the resources at minimum level. Resources refer to the memory and processor utilization. The software should efficiently perform what the user demanded and give appropriate response in each case i.e. the output given is accurate or not.

Maintainability: This characteristic of the software is important for both the software engineer and the user. If the change is to be required in the software then the change leads to the change in the software so that it performs in accordance with the user requirement. The software engineer has to respond very fast if there is any change in the user requirements. Changes should be performed like this that it will not affect the overall integrity of the software.

On-time: The software should be developed on-time. If the software is urbanized late then it is of no use. A good engineer always develops the software on-time.

Within Budget: Some of the software gets swarming. Overrun does not mean that the cost of the software exceeds the limit given by user. But, it means that the software cost is out of control. So, the software should be developed in such a manner that it will not overrun and the software being developed is within budget.

Functionality: The software system is developed in a manner that it performs the entire task perfectly for which it is developed. The software should respond correctly as the user wants.

Adaptability: The software should be adaptable. Software is desired to be adaptable all the changes efficiently. The software should easily adopt all the changes in the software with no change in the efficiency of the software.

Dependability: It is the ability of the software that should not cause any physical or economic damage in the event of system failure. It includes a range of characteristics like: Reusability, security, and safety.

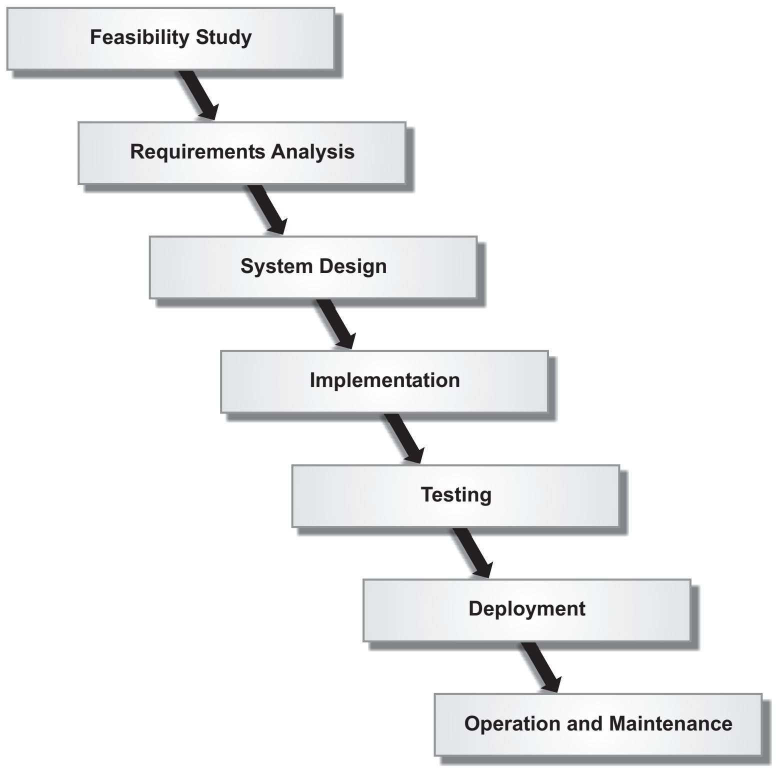
Usability: Software becomes usable if it does not call for extra efforts to be learned. Usability increases with good documentation provided along with the software. In software operations a lot depends on the quality of user manual. If software satisfies all the above characteristics then it is said to be good software or the software is well engineered.

Did u know? The first known presentation describing use of similar phases in software engineering was held by Herbert D. Benington at Symposium on advanced programming methods for digital computers on 29 June 1956. This presentation was about the development of software for SAGE.

2.3 Waterfall Model

The Waterfall model is one of the most used models of 70's. It was proposed as an alternative to Build and Fix software development method in which code was written and debugged. System was not formally designed and there was no way to check the quality criteria. Different phases of Waterfall model are shown in Figure 2.2. Given below are a brief description of different phases of Waterfall model. Feasibility study explores system requirements to determine project feasibility. All projects are feasible given unlimited resources and infinite time.

Figure 2.2: The Waterfall Model



Feasibility can be categorized into

- Economic feasibility
- Technical feasibility
- Operational feasibility
- Schedule feasibility
- Legal and contractual feasibility
- Political feasibility

Economic feasibility is also called cost-benefit analysis and focuses on determining.

The Project Costs and Benefits: Reimbursement and costs can be both touchable as well as 10 Software Engineering insubstantial. Touchable costs and benefits are the ones which can be easily measured whereas intangible ones cannot be easily measured. Examples of tangible benefits are reduced errors, improved planning and control, reduced costs etc. Insubstantial benefits include timely information, better resource control, improved information processing, better assets utilization and many more. Similarly tangible costs include cost of hardware and software required in the project, training costs, operational costs and labor costs whereas intangible costs include loss of customer goodwill, decreased operation efficiency and many more. Economic feasibility uses the concept of time-value of money (TVM) which compares the present cash outlays to future expected returns. Technical feasibility focuses on organization's ability to construct the proposed system in terms of hardware, software, operating environments, project size, complexity, experience of the organization in handling the similar types of projects and the risk analysis. Operational feasibility deals with assessing the degree to which a proposed system solve business problems. Similarly schedule feasibility ensures that the project will be completed well in time. Legal and contractual feasibility relates to issue like intellectual property rights, copyright laws, labor laws and different trade regulations. Political feasibility finally evaluates how the key stakeholders within the organization view the proposed system.

Requirements Analysis: This phase focuses on understanding the problem domain and representing the requirements in a form which are understandable by all the stakeholders of the project i.e. analyst, user, programmer, tester etc. The output of this stage is a document called Requirements Specification Document (RSD) or Software Requirements Specification (SRS). All the successive stages of software life cycle are dependent on this stage as SRS produced here is used in all the other stages of the software lifecycle.

System Design: This phase translates the SRS into the design document which depicts the overall modular structure of the program and the interaction between these modules. This phase focuses on the high level design and low level design of the software. High level design describes the main components of software and their externals and internals. Low level design focuses on transforming the high level design into a more detailed level in terms of an algorithms used, data structures used etc.

Implementation: This phase transforms the low level design part of software design description into a working software product by writing the code.

Testing Phase: It is responsible for testing the code written during implementation phase. This phase can be broadly divided into unit testing (tests individual modules), integration testing (tests groups of interrelated modules) and system testing (testing of system as a whole). Unit testing verifies the code against the component's high level and low level design. It also ensures that all the statements in the code are executed at least once and branches are executed in all directions.

Additionally it also checks the correctness of the logic. Integration testing tests the inter modular interfaces and ensures that the module drivers are functionally complete and are of acceptable quality. System testing validates the product and verifies that the final product is ready to be delivered to the customers. Additionally several tests like volume tests, stress tests, performance tests etc., are also done at the system testing level.

Deployment Phase: It makes the system operational through installation of system and also focuses on training of user.

Operations and Maintenance: This phase resolves the software .errors, failures etc. enhances the requirements if required and modifies the functionality to meet the customer demands. This is something which continues throughout the use of product by the customer.

2.3.1 Advantages and Disadvantages

Advantages and disadvantages of the Waterfall model are listed as:

Advantages

- Easy to understand even by non-technical persons i.e. customers.

- Each phase has well defined inputs and outputs e.g. input to system stage is Requirement Specification Document (RSD) and output is the design document.
- Easy to use as software development proceeds.
- Each stage has well defined deliverables or milestones.
- Helps the project manager in proper planning of the project.

Disadvantages

- Idealized, does not match reality well.
- Does not reflect iterative nature of exploratory development.
- Unrealistic to expect accurate requirements so early in project.
- Software is delivered late in project, delays discovery of serious errors.
- Difficult to integrate risk management.
- Difficult and expensive to make changes to documents, swimming upstream
- Significant administrative overhead, costly for small teams and projects.

Self Assessment Questions

1. Software is said to be .... if it uses the available resources in the most efficient manner.

   (*a*) efficiency (*b*) maintainability

   (*c*) functionality (*d*) adaptability

2. .... is the ability of the software that should not cause any physical or economic damage in the event of system failure.

   (*a*) Functionality (*b*) Adaptability

   (*c*) Dependability (*d*) Usability

3. The Waterfall model is one of the most used models of ....

   (*a*) 1964 (*b*) 1965

   (*c*) 1968 (*d*) 1970

4. .... phase transforms the low level design part of software design description into a working software product by writing the code.

   (*a*) Requirements analysis (*b*) System design

   (*c*) Implementation (*d*) Testing phase

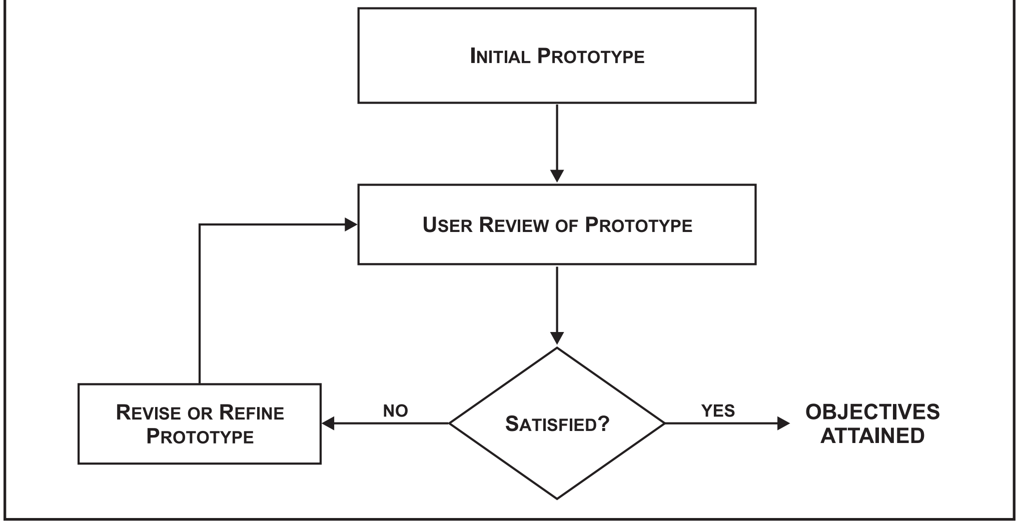
5. Testing phase makes the system operational through installation of system and also focuses on training of user.

   (*a*) True (*b*) False

2.4 Prototype Model

Prototypes are 'instruments' used within the software development process and different kinds of prototypes are employed to achieve different goals. The 'product' prototype has been variously defined within the prototyping literature and an early definition is that of Neumann and Jenkins who considers an information systems prototype to be a system that captures the essential features of a later system is the most appropriate definition of a prototype. A prototype system, intentionally incomplete, is to be modified, supplemented, or supplanted.

Figure 2.3: The Prototyping Process



The Prototyping Model was developed on the assumption that it is often difficult to know all of your requirements at the beginning of a project. Typically, users know many of the objectives that they wish to address with a system, but they do not know all the nuances of the data, nor do they know the details of the system features and capabilities. The Prototyping Model allows for these conditions, and offers a development approach that yields results without first requiring all information up-front. When using the Prototyping Model, the developer builds a simplified version of the proposed system and presents it to the customer for consideration as part of the development process. The customer in turn provides feedback to the developer, who goes back to refine the system requirements to incorporate the additional information. Often, the prototype code is thrown away and entirely new programs are developed once requirements are identified

There are a few different approaches that may be followed when using the Prototyping Model.

- Creation of the major user interfaces without any substantive coding in the background in order to give the users a "feel" for what the system will look like,
- Development of an abbreviated version of the system that performs a limited subset of functions; development of a paper system (depicting proposed screens, reports relationships etc.), or
- Use of an existing system or system components to demonstrate some functions that will be included in the developed system.

Prototyping is comprised of the following steps:

Requirements Definition/Collection: Similar to the Conceptualization phase of the Waterfall Model, but not as comprehensive. The information collected is usually limited to a subset of the complete system requirements.

Design: Once the initial layer of requirements information is collected, or new information is gathered, it is rapidly integrated into a new or existing design so that it may be folded into the prototype.

Prototype Creation/Modification: The information from the design is rapidly rolled into prototype. This may mean the creation/modification of paper information, new coding, or modifications to existing coding.

Assessment: The prototype is presented to the customer for review. Comments and suggestions are collected from the customer.

Prototype Refinement: Information collected from the customer is digested and the prototype is refined. The developer revises the prototype to make it more effective and efficient.

System Implementation: In most cases, the system is rewritten once requirements are understood. Sometimes, the iterative process eventually produces a working system that can be the cornerstone for the fully functional system.

Caution Prototyping does not eliminate the need for front-end analysis. It cannot help if the situation is not amenable to instructional design.

2.4.1 Problems/challenges Associated with the Prototyping Model

Criticisms of the Prototyping Model generally fall into the following categories:

Prototyping Can Lead To False Expectations: Prototyping often creates a situation where the customer mistakenly believes that the system is finished" when in fact it is not. More specifically, when using the Prototyping Model, the pre-implementation versions of a system are really nothing more than one-dimensional structures. The necessary, behind the-scenes work such as database normalization, documentation, testing, and reviews for efficiency have not been done. Thus the necessary underpinnings for the system are not in place.

Prototyping Can Lead to Poorly Designed Systems: Because the primary goal of Prototyping is rapid development, the design of the system can sometimes suffer because the system is built in a series of "layers" without a global consideration of the integration of all other components. While initial software development is often built to be a throwaway," attempting to retroactively produce a solid system design can sometimes be problematic.

2.4.2 Variation of the Prototyping Model

A popular variation of the Prototyping Model is called Rapid Application Development (RAD). The RAD introduces strict time limits on each development phase and relies heavily on rapid application tools which allow for quick development.

Caution Prototyping may lead to premature commitment to a design if it is not remembered that a design is only a hypothesis.

2.5 Iterative Model

The iterative development process model counters the third and fourth limits of the waterfall model and tries to unite the reimbursement of both prototyping and the waterfall model. The basic idea is that the software should be urbanized in increments, each increment addition some practical capability to the system until the full system is implemented.

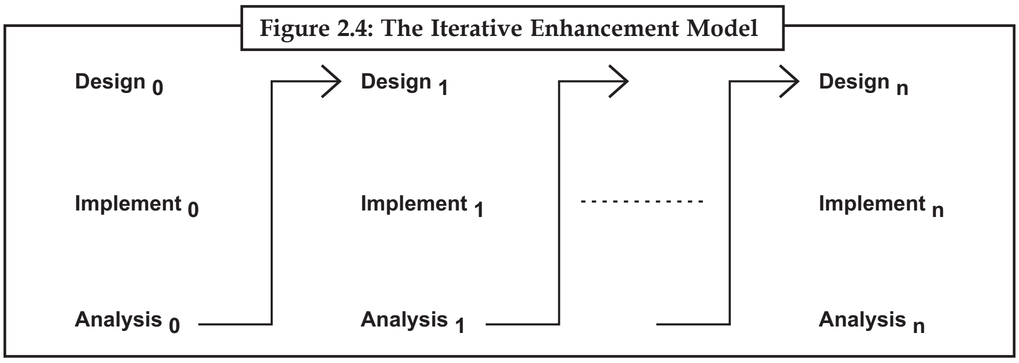
The iterative enhancement model is an example of this approach. In the first step of this model, a simple initial implementation is done for a subset of the overall problem. This subset is one that contains some of the key aspects of the problem that are easy to understand and implement and which form a useful and usable system. A project control list is created that contains, in order, all the tasks that must be performed to obtain the final implementation. This project control list gives an idea of how far along the project is at any given step from the final system.

Each step consists of removing the next task from the list, designing the implementation for the selected task, coding and testing the implementation, performing an analysis of the partial system obtained after this step, and updating the list as a result of the analysis. These three phases are called the design phase, implementation phase, and analysis phase. The process is iterated until

Notes

the project control list is empty, at which time the final implementation of the system will be available. The iterative enhancement model is shown in Figure 2.4.

Figure 2.4: The Iterative Enhancement Model



The project control list guides the iteration steps and keeps track of all tasks that must be done. Based on the analysis, one of the tasks in the list can include redesign of defective components or redesign of the entire system. However, redesign of the system will generally occur only in the initial steps. In the later steps, the design would have stabilized and there is less chance of redesign. Each entry in the list is a task that should be performed in one step of the iterative enhancement process and should be simple enough to be completely understood. Selecting tasks in this manner will minimize the chances of error and reduce the redesign work. The design and implementation phases of each step can be performed in a top-down manner or by using some other technique.

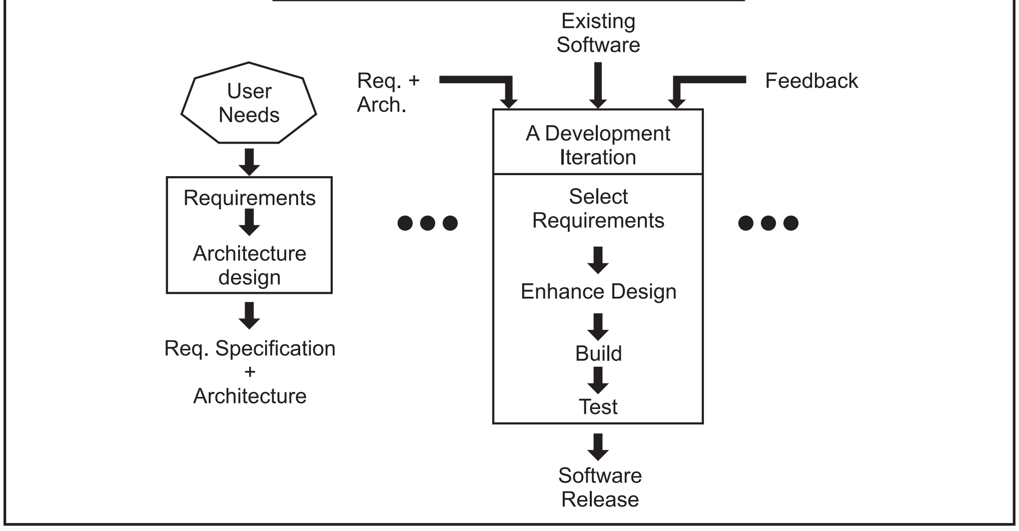
Though there are clear benefits of iterative development, particularly in allowing changing requirements, not having the all-or-nothing risk, etc., there are some costs associated with iterative development also. For example, as the requirements for future iterations are not known, the design of a system may not be too robust. Also, changes may have to be made to the existing system to accommodate requirements of the future iterations, leading to extra rework and/or discarding of work done earlier. Overall, it may not offer the best technical solution, but the benefits may outweigh the costs in many projects.

Another common approach for iterative development is to do the requirements and the architecture design in a standard waterfall or prototyping approach, but deliver the software iteratively. That is, the building of the system, which is the most time and effort-consuming task, is done iteratively, though most of the requirements are specified upfront. We can view this approach as having one iteration delivering the requirements and the architecture plan, and then further iterations delivering the software in increments. At the start of each delivery iteration, which requirements will be implemented in this release is decided, and then the design is enhanced and code developed to implement the requirements. The iteration ends with delivery of a working software system providing some value to the end user. Selecting of requirements for iteration is done primarily based on the value the requirement provides to the end users and how critical they are for supporting other requirements. This approach is shown in Figure 2.5

The advantage of this approach is that as the requirements are mostly known upfront, an overall view of the system is available and a proper architecture can be designed which can remain relatively stable. With this, hopefully rework in development iterations will diminish. At the same time, the value to the end customer is delivered iteratively so it does not have the all-or-nothing risk. Also, since the delivery is being done incrementally, planning and execution of each of iteration is done separately, feedback from iteration can be incorporated in the next iteration. Even new requirements that may get uncovered can also be incorporated. Hence, this model of iterative development also provides some of the benefits of the model discussed.

Notes

Figure 2.5: Iterative Delivery Approach



The iterative approach is becoming extremely popular, despite some difficulties in using it in this context. There are a few key reasons for its increasing popularity. First and foremost, in today's world clients do not want to invest too much without seeing returns. In the current business scenario, it is preferable to see returns continuously of the investment made. The iterative model permits this after each of iteration some working software is delivered, and the risk to the client is therefore limited. Second, as businesses are changing rapidly today, they never really know the complete requirements for the software, and there is a need to constantly add new capabilities to the software to adapt the business to changing situations. Iterative process allows this. Third, each of iteration provides a working system for feedback, which helps in developing stable requirements for the next iteration.

The four basic process areas of the iterative model are:

- The requirements phase, in which the requirements for the software are gathered and analyzed. Iteration should eventually result in a requirements phase that produces a complete and final specification of requirements.
- A design phase, in which software architecture and components to meet the requirements are designed; this may be a new design, or an extension of an earlier design.
- An Implementation phase, when the software is coded, integrated and tested.
- A review phase, in which the software is evaluated, the current requirements are reviewed, and changes and additions to requirements proposed.

The design phase develops the architecture that forms the foundation for the system and is critical to the success of the subsequent iterations. For obvious reasons, the design must facilitate change, and be robust enough to support unforeseen, future implementations. For each cycle of the model, a decision has to be made as to whether the software produced by the cycle will be discarded, or kept as a starting point for the next cycle. This approach has been referred to as incremental prototyping. However, the temptation to create a quick prototype that cannot scale-up must be resisted typically; this is not prototyping, although it could be. The cycle is complete when the requirements have been satisfied, and the release to the customer is made. On some iteration particularly early ones, a decision may have to be made to abandon the approach and begin anew.

The iterative model is not without its share of potential pitfalls. If fact, according to Kruchten, iterative development actually involves much more planning and is therefore likely to put more burden on the project manager: an overall plan has to be developed, and detailed plans will in turn be developed for each iteration". More architectural planning will also take place earlier, and as stated previously, takes on greater importance. Artifacts documents and code will have to be modified, reviewed and approved repeatedly at each revision Personnel will have to be continually reviewed as the iterative tasks cycle rapidly and overlap.

Task Prepare a flow chart for the working of iterative model.

2.5.1 Advantages of the Iterative Model

A comparison of the conventional lifecycle to the iterative one reveals that the iterative model is more flexible, as it presents more opportunities to introduce change. In the iterative model, change is an acknowledged, integral component. Rework is accepted, but should diminish rapidly; the changes should be less widespread as the architecture stabilizes and ideally, the hard issues are being resolved .The embrace of source code amendment has a profound and positive impact on software quality. When errors are found, they can be corrected at best real-time, at worst in the next iteration. Contrast this to the waterfall model, where software is often released with major defects--because it is too late in the lifecycle to rewrite, or redesign components.

Another advantage of the iterative model is that the complexity of implementing the system is never overwhelming. Because elements are designed, developed and integrated in iterations, the analysis paralysis that is common on enterprise scope projects is alleviated. In addition, the developers get a chance to grow with the project. Each of iteration can leverage the business knowledge gained on the previous and the team gets used to delivering finished software.

Using the iterative model brings more focus on the software, and less on the specifications. This is a valid trade-off, as the customer is not buying specifications. The specifications are only an attempt to communicate the business mind to the technical mind. By moving forward into an implementation sooner in the project, requirement issues are bubbled to the surface earlier. If critical errors in design or integration are to be discovered, it is better for all concerned that they be uncovered early in the project. This early feedback from the customer serves to drive the next iteration in the correct direction. The requirements can be modified without junking weeks of development, as each of iteration through the model has all of the aspects of a traditional software project, which has been completed.

Did u know? The iteration and increment model was first used by NASA's 1960s Project Mercury.

2.6 Time Boxing

Eventually boxing, as in other iterative development approaches, some software is developed and a working system is delivered after each iteration. In time boxing, each iteration is of equal duration, which is the length of the time box. In this section we discuss the various theoretical issues relating to this process model.

2.6.1 A Time box and Stages

In the time boxing process model, the essential unit of expansion is a time box, which is of permanent duration. Within this time box all activities that need to be performed to successfully release the next version are executed. Since the duration is fixed a key factor in selecting the requirements or features to be built in a time box is what can be fit into the time box.

Each time box is divided into a sequence of stages, like in the waterfall model. Each stage performs some clearly defined task of the iteration and produces a clearly defined output. The output from one stage is the only input from this stage to the next stage. Furthermore, the model requires that the duration of each stage, that is, the time it takes to complete the task of that stage, is approximately the same.

There is a dedicated team for each stage. That is, the team for a stage performs only tasks of that stage tasks for other stages are performed by their respective teams. This is quite different from many other models where the implicit assumption is that the same team (by and large) performs all the different tasks of the project or the iteration.

As pipelining is to be employed, the stages must be carefully chosen. Each stage performs some logical activity which may be communication intensive – that is, the team performing the task of that stage needs to communicate and meet regularly. However, the stage should be such that its output is all that is needed from this stage by the team performing the task of the next stage. In other words, the output should be such that it can be passed to the team for next stage, and the team needs to communicate minimally with the previous stage team for performing their task. Note that it does not mean that the team for a stage cannot seek clarifications with teams of earlier stages – all it means is that the communication needs between teams of different stages are so low that their communication has no significant effect on the work of any of the teams.

2.6.2 Pipelined Execution

With fixed duration for implementation of iteration, the model renders itself to pipelining. Each iteration can be viewed like one instruction whose implementation is divided into a succession of fixed duration stages, a stage being executed after the conclusion of the preceding stage. In general, let us consider a time box with duration T and consisting of n stages – S1, S2, Sn. As stated, each stage Si is executed by a dedicated team (similar to having dedicated hardware for executing a stage in an instruction). Let the size of the team dedicated for stage Si is RI representing the number of resources assigned to this stage.

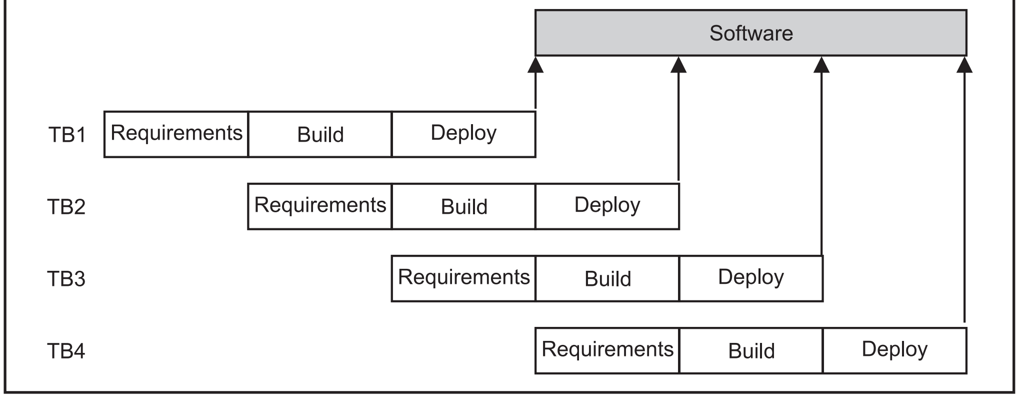
The team of each stage has T/n time available to finish their task for a time box, that is, the duration of each stage is T/n. When the team of a stage it completes the tasks for that stage for a time box k, it then passes the output of the time box to the team executing the stage i+1, and then starts executing its stage for the next time box k + 1. Using the output given by the team for Si, the team for S_{i+1} starts its activity for this time box. By the time the first time box is nearing completion, there are n – 1 different time boxes in different stages of execution. And though the first output comes after time T, each subsequent delivery happens after T/n time interval, delivering software that has been developed in time T.

As an example, consider a time box consisting of three stages: requirement specification, build, and deployment. The requirement stage is executed by its team of analysts and ends with a prioritized list of requirements to be built in this iteration. The requirements document is the main input for the build team, which develops the code for implementing these requirements, and performs the testing. The tested code is then handed over to the deployment team, which performs pre-deployment tests, and then installs the system for production use.

These three stages are such that in a typical short-cycle development, they can be of equal duration (though the effort consumed is not the same, as the manpower deployed in the different stages is different.) Also, as the boundary between these stages is somewhat soft (e.g. high level design can be made a part of the first stage or the second), the duration of the different stages can be made approximately equal by suitably distributing the activities that lie at the boundary of two adjacent stages.

With a time box of three stages, the project proceeds as follows. When the requirement team has finished requirements for timebox-1, the requirements are given to the build-team for building the software. Meanwhile, the requirement team goes on and starts preparing the requirements for time box. When the build for the timebox-1 is completed, the code is handed over to the deployment team, and the build team moves on to build code for requirements for time box and the requirements team moves on to doing requirements for timebox-3. This pipelined execution of the time boxing process is shown in Figure 2.6.

Figure 2.6: Executing the Time Boxing Process Model



With a three-stage time box, at most three iterations can be concurrently in progress. If the time box is of size T days, then the first software delivery will occur after T days. The subsequent deliveries, however, will take place after every T/3 days.

2.6.3 Time, Effort, and Team Size

It should be clear that though the duration of each of iteration has not been reduced, the delivery time to the end client (after the first iteration) reduces by a factor of n with an-stage time box. That is, the speedup (the reduction in the average completing time of iteration) is n times. We can also view it in terms of throughput the amount of output per unit time. We can clearly see that in steady state, the throughput of a project using time boxing is n times that of if serial iterations were employed. In other words, n times more functionality is being delivered per unit time. If the size of the team executing the stage Si is RI, then the effort spent in the stage Si is

$$E\ (Si) = RI * T/n$$

Note that the model only requires that the duration of the stages be approximately the same, which is T/n in this case. It does not imply that the amount of effort spent in a stage is same. The effort consumed in a stage Si also depends on RI, the size of the team for that stage. And there is no constraint from the model that the different R is should be the same.

The total effort consumed in iteration, i.e. in a time box, is

$$E(TB) = \sum_{i=1}^{n} E(S_i)$$

This effort is no different than if the iterations were executed serially – the total effort for iteration is the sum of the effort for its stages. In other words, the total effort for iteration remains the same in time boxing as in serial execution of iterations. If the same effort is spent in each of iteration, what is the cost of reducing the delivery time? The really cost of this increased throughput is in the resources used in this model. The total team size for the project, in which multiple time boxes may be running in parallel, is Let us compare the team size of a project using time boxing with another project that executes iterations serially. In a serial execution of iterations, it is implicitly assumed that the same team performs all the activities of the iteration, that is, they perform all the stages of the iteration. For sake of illustration, let us assume that the team size is fixed throughout the iteration, and that the team has R resources. So, the same R people perform the different stages – first they perform the tasks of stage 1, then of stage 2, and so on. With time boxing, there are different teams for different stages. Assuming that even with dedicated resources for a stage, the same number of resources is required for a stage as in the linear execution of stages; the team size for each stage will be R. Consequently, the total project team size when the time box has n stages is n × R. That is, the team size in time boxing is n times the size of the team in serial execution of iterations. Hence, in a sense, the time boxing provides an approach for utilizing additional manpower to reduce the delivery time. It is well known that with standard methods of executing projects, we cannot compress the cycle time of a project substantially by adding more manpower. This principle holds here also within a time box – we cannot reduce the size of a time box by more manpower. However, through the time boxing model, we have been able to use more manpower in a manner such that by parallel execution of different stages we are able to deliver software quicker.

2.6.4 Unequal Stages and Exceptions

Clearly, the reality will rarely present itself in such a clean manner such that iterations can be fit in a time box and can be broken into stages of equal duration. There will be scenarios where these requirements will not hold. What happens when such exceptions present themselves? First situation which is likely to occur is that the stages are of unequal duration. As the pipelining concepts from hardware tell us in such a situation the output is determined by the slowest stage, that is, the stage that takes the longest time. With unequal stages, each stage effectively becomes equal to the longest stage and therefore the frequency of output is once every time period of the slowest stage. Note that even with this, a considerable speedup is possible. For example, let us consider a 3-stage pipeline of the type discussed in which the different stages are 2 weeks, 4 weeks, and 3 weeks – that is, the duration of the time box is 9 weeks. In a serial iterative development, software will be delivered every 9 weeks. With time boxing, the slowest stage will determine the speed of execution, and hence the deliveries will be done every 4 weeks. This delivery time is less than half the delivery time of serial iterations. However, there is a cost if the stages are unequal. As the longest stage determines the speed, each stage effectively becomes equal to the slowest stage. In the example given, it means that the 8first and third stages will also get 4 weeks each, even though their work requires only 2 and 3 weeks. In other words, it will result in "slack time" for the teams for the first and third stage, resulting in under utilization of resources. So, the resource utilization, which is 100% when all the stages are of equal duration, will reduce resulting in underutilization of resources. Of course, this wastage can easily be reduced by reducing the size of the teams for the slower stages to a level that they take the same time as the slowest stage. Note that elongating the cycle time by reducing.

Manpower is generally possible (even though the reverse is not possible.) Another special situation can easily arise an exceptional condition arises during the execution of a stage of sometime box, due to which the stage is not able to finish in its allotted time. We do not need to worry about the nature of the exception except that the net effect of the exception is that it elongates that stage by ΔT. Clearly, if such an exception occurs, the execution of the later stages will be delayed resulting in the output being delivered late by ΔT. Similarly, due to this delay, the output of earlier stages in later time boxes cannot be consumed in time, resulting in the teams of these stages "waiting" for their output to be consumed. The net result of this is that, one delivery gets delayed by ΔT, with a corresponding slack time for each team for one time box. After that, all future deliveries will come after every T/n time units (for an -stage time box of T duration.)

Notes

2.7 Comparison Process Models

The comparison between various process models is given in Table 2.1.

Table 2.1: Comparison of Process Models

Strengths	**Weaknesses**	**Types of Projects**
WATERFALL Simple Easy to execute Intuitive and logical	All or nothing approach Requirements frozen early Disallows changes Cycle time too long May choose outdated hardware technology User feedback not allowed Encourages requirement bloating	For well-understood problems Short duration projects Automation of existing manual systems
PROTOTYPING Helps in requirements elicitation Reduces risk Leads to a better system	Front heavy process Possibly higher costs Disallows later changes	Systems with novice users when there are uncertainties in requirements
ITERATIVE ENHANCEMENT Regular/quick deliveries Reduces risk Accommodates changes Allows user feedback Allows reasonable exit points Avoids requirement bloating Prioritizes requirements	Each iteration can have planning overhead Costs may increase as work done in one iteration may have to be undone later System architecture and structure may suffer as frequent changes are made	For businesses where time is of essence Where risk of a long project cannot be taken Where requirements are not known
SPIRAL Controls project risks Very flexible Less documentation needed	No strict standards for software development No particular beginning or end of particular phase	Projects built on untested assumptions

Case Study

Pepsi-Cola Manufacturing International Ltd.

GCS Implementation

Pepsi-Cola Manufacturing International Ltd. ("PCMIL") has successfully executed a compliance initiative supported by BPM technology in response to a need for more efficient

Contd...

Notes

compliance with the corporate Global Control Standard (GCS). Their primary objective for this initiative was to streamline their preparations for SOX activities and to ensure compliance with GCS requirements. Using Interfacing's Enterprise Process Center (EPC), PCMIL effectively integrated their GCS implementation with day-to-day operations in order to benefit from a unified approach to corporate governance and process management. They achieved significant cost savings for their internal GCS audit and extremely positive overall results. "Our auditors gave us excellent feedback on the use of the EPC as part of our GCS implementation initiative. They were particularly impressed with our ability to identify controls and relate them to risks prioritized with the Risk Matrix. They have since recommended our approach to internal GCS compliance as best-practice for other PepsiCo plants." Control Officer Team, PCMIL.

Challenge Addressed

Without a formal and central documentation system, the preparation time – and, therefore, business disruption – for internal GCS audits and SOX reporting was very high, and inconsistent process knowledge was a source of insecurity as it pertained to the outcome of the audits. PCMIL also needed to address an acute lack of easy accessibility to the documentation and process flows for auditors and employees, as well as inconsistencies and gaps in their process documentation that were compounding problems associated with manual audit preparation. Although approximately 100 critical processes had been mapped with Microsoft.

Visio and role matrices had been documented in MS Excel processes were decentralized and managed manually and independently of GRC programs. PCMIL did not have a system in place to support their GCS implementation initiative, and frequent, manual process updates, as well as the audit process itself, were proving time-consuming and costly due to a lack of integration with business processes.

Process Undertaken and Roles Involved

The GCS implementation initiative was extremely successful as it received full support from top management at PCMIL, the IT department, a dedicated Project Manager, and the core team assigned to the program. It involved employees at all levels of the organization in process modelling and training, so resistance to the end-result was effectively eliminated.

PCMIL process modellers captured and documented process-related information, including documents, risks, controls, business rules, roles, and resources, that they then linked directly to processes in the EPC. They utilized the EPC's best-practice RACI (Responsible / Accountable/ Consulted/Informed) framework to define roles and responsibilities and leveraged the EPC's integrated Risk Matrix to quantify and prioritize risks in terms of impact and likelihood. With their GCS compliance requirements embedded in their processes, PCMIL used the EPC's automatic reporting capabilities to generate process documentation and audit reports.

Questions

1. How to use challenge addressed in Pepsi-Cola Manufacturing International Ltd.?

2. Explain the process undertaken and roles involved.

Self Assessment Questions

6. The software process is a critical factor for delivering quality …………………. systems

(*a*) software	(*b*) working
(*c*) model	(*d*) None of these.

Notes

7. This effort is no different than if the iterations were …………………. serially

(*a*) assumption	(*b*) working
(*c*) executed	(*d*) None of these.

8. As pipelining is to be …………………. the stages must be carefully chosen

(*a*) model	(*b*) employed
(*c*) executed	(*d*) None of these.

9. This early feedback from the customer serves to drive the next …………………. in the correct direction.

(*a*) working	(*b*) iteration
(*c*) sequence	(*d*) All of these.

10. The project control list guides the iteration steps and keeps track of …………………. that must be done.

(*a*) all tasks	(*b*) process
(*c*) requirements	(*d*) All of these.

11. A popular variation of the …………………. is called rapid application development (RAD).

(*a*) model	(*b*) prototyping model
(*c*) requirements	(*d*) process model

12. …………………. is responsible for testing the code written during implementation phase.

(*a*) Testing phase	(*b*) process testing
(*c*) Rapid testing	(*d*) None of these.

13. The information from the design is …………………. rolled into prototype.

(*a*) rapidly	(*b*) responsible
(*c*) feedback	(*d*) None of these.

14. The Prototyping Model was ………………… on the assumption that it is often difficult to know all of your requirements at the beginning of a project.

(*a*) developed	(*b*) rolled
(*c*) rapid	(*d*) All of these.

15. The iterative model is not without its share of ………………….

(*a*) developed pitfalls	(*b*) potential pitfalls
(*c*) Rapid	(*d*) All of these.

2.8 Summary

- The software process is a critical factor for delivering quality software systems, as it aims to manage and transform the user need into a software product that meets this need.
- In contrast to software life cycle models, software process models often represent a networked Sequence of activities, objects, transformations, and events.
- The Waterfall model is one of the most used models of 70's. It was proposed as an alternative to Build and Fix software development.

- Implementation phase transforms the low level design part of software design description into a working software product by writing the code.
- The Prototyping Model was developed on the assumption that it is often difficult to know all of your requirements at the beginning of a project.
- The iterative development process model counters the third and fourth limitations of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model.
- In time boxing as in other iterative development approaches, some software is developed and a working system is delivered after each of iteration.

2.9 Keywords

Comparison Testing: Test Cases results are compared with the predicted results of the Test Oracle. Test Oracle: a mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test.

Iterative Enhancement: iterative enhancement life cycle model counters the third limitation of the waterfall model and tries to combine the benefits of both prototyping and the waterfall model.

Requirement Analysis and Specification: Identifies the problems a new software system is suppose to solve, its operational capabilities, its desired performance characteristics, and the resource infrastructure needed to support system operation and maintenance.

Software Life Cycle Model: A software life cycle model (SLCM) is a representation of the major components of software development work and their interrelationships in a graphical framework that can be easily understood and communicated. Just as the WBS partitions the deliverable into its component parts so the SLCM apportions the work to be done into manageable work units.

Software Project: In Software Project, the end users and developers need to know the length, duration and cost of the project. It is a process of managing, allocating and timing resources to develop computer software that meets requirements.

Lab Exercise

1. Prepare a flow chart for waterfall model.
2. Search about the spiral model.

2.10 Review Questions

1. Explain the software processes and models?
2. What is software project?
3. Explain the concept of Waterfall Model.
4. What is pipelined execution?
5. Explain the unequal stages and exceptions.
6. Define the iterative model.
7. What are the advantage and disadvantage waterfall model?
8. Write the four basic process areas of the iterative model.
9. What is prototype model?
10. Comprise between different types of process models.

Notes

Answers for Self Assessment Questions

1.	(*a*)	2.	(*c*)	3.	(*d*)	4.	(*c*)	5.	(*b*)
6.	(*a*)	7.	(*c*)	8.	(*b*)	9.	(*d*)	10.	(*a*)
11.	(*b*)	12.	(*a*)	13.	(*a*)	14.	(*a*)	15.	(*d*)

2.11 Further Readings

Books

Process Modeling and Model Analysis, by Katalin M. Hangos

Software Engineering, by Schach

Online link

http://books.google.co.in/books?id=BZyS2VPw1wAC&pg=PA66&dq=Further+Readings+Software+Processes+%26+models:&hl=en&sa=X&ei=T5QGUIPJGMLRrQfqvZG2Bg&ved=0CDUQ6AEwAA#v=onepage&q=Further%20Readings%20Software%20Processes%20%26%20models%3A&f=false

Unit 3: Software Requirements

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the problem analysis
- Define data flow
- Discuss the object oriented modelling
- Understand representation of prototyping
- Discuss software requirement specification document (SRS)
- Discuss the characteristics and components
- Understand the specification language
- Discuss the structure of document

Introduction

Before starting to design a software product, it is extremely important to understand the precise requirements of the customer and to document them properly. In the past, many projects have suffered because the developers started implementing something 'without determining whether they were building what the customer exactly wanted. Starting development activities with improperly documented requirements is possibly the biggest mistake that one can commit during a product development. Improperly documented requirements increase the number of iterative changes required during the life cycle phases, and thereby push up the development costs tremendously. They also set the ground for bitter customer-developer disputes and protracted legal battles. Therefore, requirements analysis and specification is considered to be a very important phase of software development and has to be undertaken with utmost care. Even experienced analysts take considerable time to understand the exact requirements of the customer and to document them. They know that without a clear understanding of proper documentation of the problem, it is impossible to develop a satisfactory solution. The requirements analysis and specification phase starts once the feasibility study phase is complete and the project is found to be financially sound and technically feasible. The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organize these requirements in a specification document. This phase consists of the following two activities:

- Requirements gathering and analysis.
- Requirements specification.

To carry out the requirements gathering and analysis activity, a few members of the development team usually visit the customer site. The engineers who gather and analyze customer requirements and write the requirements specification document are known as System analysts in the software industry parlance. System analysts collect data pertaining to the product to be developed and analyze these data to conceptualize what exactly needs to be done. He then proceeds to write the Software Requirements Specification (SRS) document. The SRS document the final output of the requirements analysis and specification phase. Once the SRS document is ready, it is first reviewed internally by the project team ensure that it is understandable, consistent, unambiguous, and complete. The SRS document is then given to the customer for review. After the customer has reviewed the document and approved it, the same forms the basis for all future development activities also serves as a contract document between the customer and the development organization. With this brief introduction to the requirements analysis and specification phase, let examine the various activities performed in this phase in greater depth.

3.1 Problem Analysis

As basic aspire of problem analysis is to obtain a clear sympathetic of the needs of the clients and the users, regularly the client and the users do not understand or know all their needs, because the potential of the new system is often not fully appreciated. The analysts have to ensure that the real needs of the clients and the users are uncovered, even if they do not know them clearly. That is, the analysts are not just collecting and organizing information about the client's organization and its processes, but they also act as consultants who play an active role of helping the clients and users identify their needs. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer". For software design, partition the problem into sub problems and then try to understand each sub problem and its relationship to other sub problems in an effort to understand the total problem. That is goal is to divide the problem into manageably small pieces that can be solved separately, because the cost of solving the entire problem is more than the sum of the cost of solving all the pieces. The different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. Problem partitioning also aids design verification. The concepts of state and projection can sometimes also be used effectively in the partitioning process.

1. A state of a system represents some conditions about the system. This approach is sometimes used in real-time software or process-control software.
2. Projections, different viewpoints of the system are defined and the system is then analyzed from these different perspectives. The different "projections" obtained are combined to form the analysis for the complete system. Analyzing the system from the different perspectives is often easier, as it limits and focuses the scope of the study.

Informal Approach

1. The informal approach to analysis is one where no defined methodology is used.
2. The information about the system is obtained by interaction with the client, end users, questionnaires, study of existing documents, brainstorming etc.
3. The informal approach to analysis is used widely and can be quite useful because conceptual modelling based approaches frequently do not model all aspects of the problem and are not always well suited for all the problems.
4. As the SRS is to be validated and the feedback from the validation activity may require further analysis or specification.
5. Choosing an informal approach to analysis is not very risky the errors that may be introduced are not necessarily going to slip by the requirements phase. Hence such approaches may be the most practical approach to analysis in some situations.
6. Various fact finding methods are used to collect detailed information about every aspect of an existing system.

Shadowing

1. Shadowing is a technique in which we observe a user performing the tasks in the actual work environment and ask the user any questions related to the task.
2. We typically follow the user as the user performs tasks.
3. The information obtained by using this technique was firsthand and in context.

3.2 Data Flow

Software testing is "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items". The main goals of software testing are to reveal bugs and to ensure that the system being developed complies with the customer's requirements. To make testing effective, it is recommended that test planning/development begin at the onset of the project. Software testing techniques can be divided into two kinds: black box and white box techniques.

Black box testing is mainly a validation technique that checks to see if the product meets the customer requirements. However, white box testing is a verification technique which uses the source code to guide the selection of test data. Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. Additionally, he/she may define a variable, but not initialize it and then use that variable in a predicate.

Notes

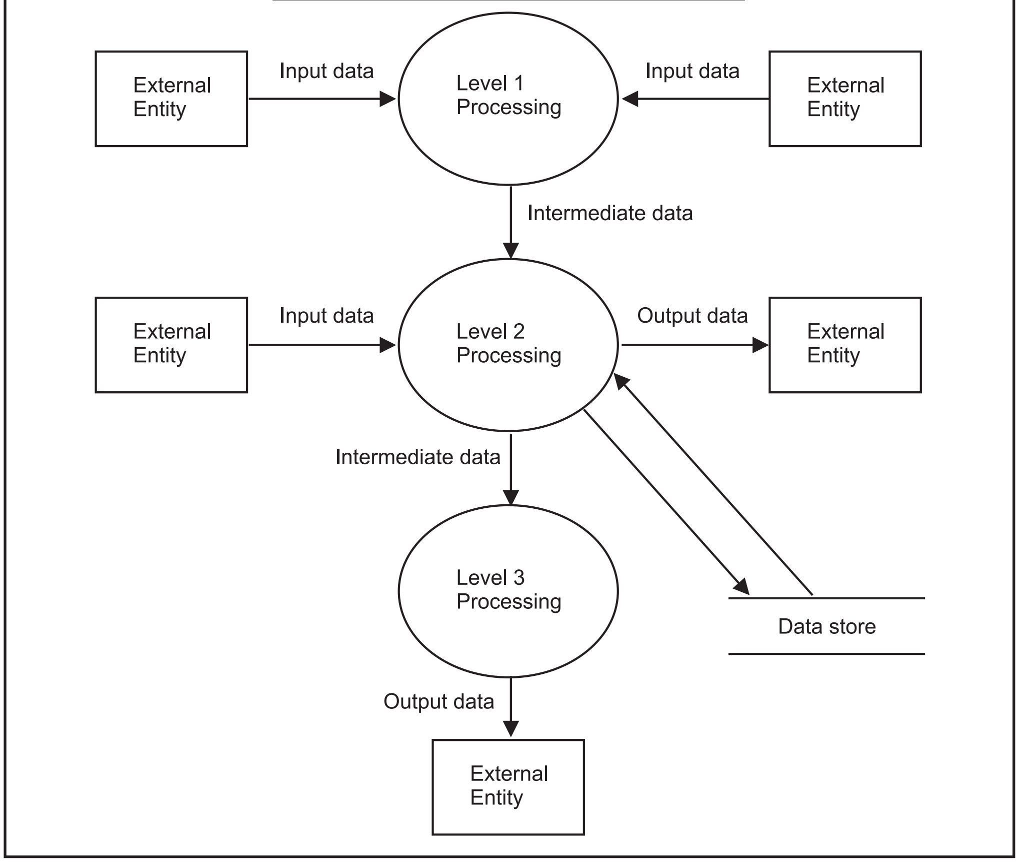
```
e.g. Int x;
if (x ==100) {};
```

In data-flow testing, the first step is to model the program as a control flow graph. This helps to identify the control flow information in the program. In step 2, the associations between the definitions and uses of the variables that is needed to be covered in a given coverage criterion is established. We have discussed the concept of dataflow testing. The next section covers the data-flow testing criteria and data-flow anomalies. A billing application is considered and the corresponding control flow graphs are presented and annotated to explain the concept of dataflow testing. Data-flow testing is a control flow testing technique which also examines the lifecycle of data variables. Use of data-flow testing leads to a richer test suite concentrating on improper use of data due to coding errors.

3.2.1 Data Flow Diagrams (DFD)

It is a graphical representation of flow of data through a system. It pictures a system as a network of functional processes. The basis of DFD is a data flow graph, which pictorially represents transformation on data as shown in Figure 3.1.

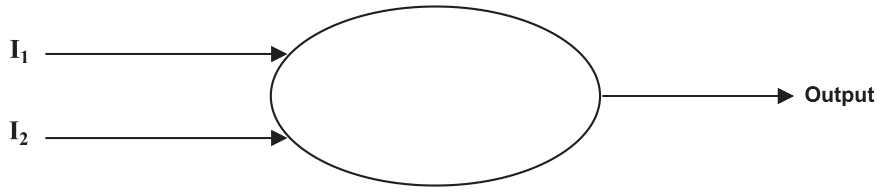
Figure 3.1: Data Flow Diagrams



In this diagram, the external entities provide input data for the processing. During the processing, some intermediate data is generated. After final processing, the final output data is generated. The data store is the repository of data.

The structured approach of system design requires extensive modelling of the system. Thus, instead of making a complete model exhibiting the functionality of system, the DFD's are created in a layered manner. At the first layer, the DFD is made at block level and in lower layers, the details are shown. Thus, level “0” DFD makes a fundamental system. (See Figure 3.2)

Figure 3.2: Depiction of Process



The DFD can represent the system at any level of abstraction. DFD of “0” level views entire software element as a single bubble with indication of only input and output data. Thus, “0” level DFD is also called as Context diagram. These symbols can be seen in Figure 3.3.

Figure 3.3: Symbols of a Data Flow Diagram

Symbol	Name	Description
⟶	Data Flow	Represents the connectivity between various processes
○	Process	Performs some processing of input data
▭	External Entity	Defines source or destination of system data. The entity which receives or supplies information.
═	Data Store	Repository of data

Rules for making DFD

The following factors should be considered while making DFDs:

1. Keep a note of all the processes and external entities. Give unique names to them. Identify the manner in which they interact with each other.
2. Do numbering of processes.
3. Avoid complex DFDs (if possible).
4. The DFD should be internally consistent.
5. Every process should have minimum of one input and one output.

The data store should contain all the data elements that flow as input and output.

Data Flow Model

The data flow model is based on the program’s control flow graph do not confuse that with the program’s data flow graph. We annotate each link with symbols (for example, d, k, u, c, p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights. The control flow graph structure is same for every variable: it is the weights that change.

Components of the model

1. To every statement there is a node, whose name is unique. Every node has at least one out link and at least one in link except for exit nodes and entry nodes.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.
3. The out link is simple statements (statements with only one out link) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement A= A + B in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.
4. Predicate nodes (if-then-else, do while, case) are weighted with the p use(s) on every out link, appropriate to that out link.
5. Every sequence of simple statements (e.g., a sequence of nodes with one in link and one out link) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data flow action for any variable.

Data Flow Testing

Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects. For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.

Motivation

It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

Data Flow Machines

There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines (MIMD).

Machine Architecture

Most computers today are von-Neumann machines. This architecture features interchangeable storage of instructions and data in the same memory units. The Von Neumann machine Architecture executes one instruction at a time in the following micro instruction sequence:

1. Fetch instruction from memory
2. Interpret instruction
3. Fetch operands
4. Process or Execute
5. Store result
6. Increment program counter
7. GOTO 1

Multi-instruction, Multi-data machines (MIMD)

These machines can fetch several instructions and objects in parallel. They can also do arithmetic and logical operations simultaneously on different data objects. The decision of how to sequence them depends on the compiler.

Data Dictionary

This is another tool of requirement analysis which reduces complexity of DFD. A data dictionary is a catalogue of all elements of a system. DFD depicts flow of data whereas data dictionary gives details of that information like attribute, type of attribute, size, names of related data items, range of values, data structure definitions etc. The name specifies the name of attribute whose value is collected. For example, fee deposit may be named as FD and course opted may be named as CO.

Related data items captures details of related attributes. Range of values store total possible values of that data. Data structure definition captures the physical structure of data items.

Some of the symbols used in data dictionary are given:

X= [a/b] x consists of either data element a or b

X=a x consists of an optional data element a

X=a+b x consists of data element a and b

X=y{a}z x consists of some occurrences of data elements at which are between y and z.

Separator

Comments

@ Identifier

( ) Options

Example: The data dictionary of payroll may include the following fields:

PAYROLL = [Payroll Details]

= @ employee name + employee + id number + employee address + Basic Salary + additional allowances

Employee name = * name of employee *

Employee id number = * Unique identification no. given to every employee*

Basic Salary = * Basic pay of employee *

Additional allowances = * the other perks on Basic pay *

Employee name = First name + Last name

Employee address = Address 1 + Address 2 + Address 3

Structured Requirements Definition

Structured Requirements definition is an approach to perform the study about the complete system and its various sub-systems, the external inputs and outputs, functionality of complete system and its various sub-systems. The following are various steps of it:

Step 1 : Make a user level data flow diagram.

This step is meant for gathering requirements with the interaction of employee. In this process, the requirements engineer interviews each individual of organisation in order to learn what each individual gets as input and produces as output. With this gathered data, create separate data flow diagram for every user.

Step 2 : Create combined user level data flow diagram.

Create an integrated data flow diagram by merging old data flow diagrams. Remove the inconsistencies if encountered, in this merging process. Finally, an integrated consistent data flow diagram is generated.

Step 3 : Generate application level data flow diagram.

Perform data analysis at system's level to define external inputs and outputs.

Step 4 : Define various functionalities.

In this step, the functionalities of various sub-systems and the complete system are defined.

Static Data-flow Testing

With static analysis, the source code is analyzed without executing. Let us consider an example of an application to calculate the bill of a cellular service customer depending upon on his/her usage. The following rules Static Data-flow testing will fail in situations where the state of a data variable cannot be determined by just analyzing the code. This is possible when the data variable is used as an index for a collection of data elements. For example, in case of arrays, the index might be generated dynamically during execution hence we cannot guarantee what the state of the array element is which is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence not completely anomalous.

Dynamic Data-flow Testing

The primary purpose of dynamic data-flow testing is to uncover possible bugs in data usage during the execution of the code. To achieve this, test cases are created which trace every definition to each of its use and every use is traced to each of its definition. Various strategies are employed for the creation of the test case. Include at least one path from the definition to every computational use; if there are definitions of the variable that are not covered then add predicate use test cases as required to cover every definition.

Static Data Flow Analysis

Static profiling is a technique that produces estimates of execution likelihoods or frequencies based on source code analysis only. It is frequently used in determining cost/benefit ratios for certain compiler optimizations. In previous work, we introduced a simple algorithm to compute execution likelihoods, based on a control flow graph and heuristic branch prediction.

Dynamic Data Flow Analysis

Dynamic data flow analysis is a method of analyzing the sequence of actions on data in a program as it is being run. Mention that there are three types of actions that can be performed on a data item, namely, define (d), reference (r) and under (u). A variable is said to be defined if a value is assigned to it; referenced if the value is fetched from the memory; and under if the value becomes unknown or inaccessible. During program execution, a variable can be in one of the following four states: state D, state R (referenced), state U, and state A .Introduced tracing the data flow anomalies through state transitions instead of sequence actions. When an action is applied on a variable, its state follows transitions according to the state transition diagram. When a variable enters the abnormal state, this indicates a data flow anomaly.

Did u know? The data-flow diagrams (DFDs) were introduced in the late 1970s and popularized for structured analysis and design.

3.3 Object Oriented Modelling

Notes

Object-oriented software engineering is the methodology. This technology uses actors, use cases, interface descriptions and problem domain objects. Relevant notations are drawn from the UML (Unified Modelling Language) the object oriented paradigm is the framework in software engineering, influencing all effort in information science. It is one of the main objectives of the software engineering. Object models are different from other modelling techniques because they have merged the concept of variables and abstract data types into an abstract variable type: an object. Objects have identity, state, and behaviour and object models are built out of systems of these objects. To make object modelling easier, there are concepts of type, inheritance, association, and possibly class. Object modelling focus on identity and behaviour is completely different from the relational model's focus on information. Ontology is well known as description of declaration and abstract way the domain information of the application, it involved with vocabulary and how to constrain the use of the data and they are used widely in the semantic web approach, which requires a significant degree of structure. In the area of ontology the concept have been supplemented above which allow expressing the similarity of concept in ontology with object (atom) in object oriented. Ontology's themselves are rising as an important tool for coping with very great, compound and various sources of information. It has also been known that ontology's are advantageous for software engineering. Ontology representations are little known outside AI research laboratories; In contrast, commercial interest has results in ideas from object oriented programming community maturing into industry standards and powerful tools for object oriented analysis design and implementation. And this maturing standards and tools can be used for ontology modelling. Ontology is formally specified models of bodies of knowledge defining concepts used to describe a domain and the relationship that hold between them.

Object Modelling

The object-oriented model is based on a compilation of objects. An object contains values stored in case in point variables within the object. Thus objects enclose objects to a randomly deep level of nesting. Attributes/properties objects will have at least one attribute. Possible slot types are primitive types (integer, Boolean, string etc.), references to other objects (modelling relationships) and sets of values of these types. An object also contains bodies of code that operate on the object. These bodies of code are called methods. Method/Operations are attached to classes or slots and contain Meta information, such as comments, constraints and default values. Relationship/relations represent the relation between objects/classes from object model (KB). Major classes of relations exist: relations combining labels (the name we tend to give to things) and concepts (the things themselves) and concepts and relations combining concepts (the part-whole relation). Objects that contain the same types of values and the same methods are grouped into classes. A class may be viewed as a type definition for objects. Analogy: the programming language concept of an abstract data type. The only way in which one object can access the data of another object is by invoking the method of that other object. This is called sending a message to the object. Internal parts of the object, the instance variables and method code, are not visible externally or some researchers called it as black box. Black Box Testing is used to test the functional correctness of the program. Black Box Testing attempts to find errors in the following categories.

1. Incorrect or Missing Function
2. Interface Error
3. Performance Error

Black Box Testing an approach to testing where the program is considered as a "Black Box", that is one cannot "see" inside of it. The program test cases are based on the system specification, and are not based on the internal workings of the program. White Box Testing is conducted in the early stage of testing process; While Black Box Testing is conducted at the later stage to verify the functional correctness of the program. Black Box Testing is also called as Behavioural Testing or Partition Testing.

Object-Oriented Analysis

The principle of object-oriented examination is to develop solution models that satisfy the customer requirement. The object oriented analysis generates model of the problem sphere of influence. It represents classes, objects and interaction between objects. We use UML (unified modeling language) to represent the analysis details. Use case diagram, class diagram, are used to model the object oriented analysis using UML.

Object-Oriented Design

The object-oriented design converts the object-oriented analysis model into a design model. This serves a draw-round for software building. Object-oriented design supports following object oriented concepts such as abstraction, information hiding, functional independence, modularity. Design is the initial step in moving towards from the problem domain to the solution domain. A detailed design include specification of all the classes with its attributes, detailed interface. The purpose of design is to specify a working solution that can be easily translated into a programming language code.

The object-oriented design is classified into:

- Architectural design
- Detailed design

Architectural design

Architectural designs divide the system into dissimilar sub systems known as packages. Then the dependence, relationship and communication between the packages are also identified. Package diagram is use to represent architectural design using UML.

Detailed design

It describes the detailed description of the classes that is all the attributes (variables and functions). The detailed class diagram represents the detailed design using UML.

Did u know? Object-modelling technique (OMT) was introduces by Rumbaugh in 1994.

Caution Be aware while changing names of object elements, the consistency of the design is must be not affected.

Self Assessment Questions

1. .... is a white box testing technique that can be used to detect improper use of data values due to coding errors.

   (*a*) Software testing (*b*) Data-flow testing

   (*c*) Static data-flow testing (*d*) Dynamic data-flow testing

2. The primary purpose of .... is to uncover possible bugs in data usage during the execution of the code.

   (*a*) software testing (*b*) data-flow testing

   (*c*) static data-flow testing (*d*) dynamic data-flow testing

3. ……………….engineering is the methodology.

   (*a*) Real-time software (*b*) Process-control software

   (*c*) Object-oriented software (*d*) None of these

4. The .... is based on the program's control flow graph do not confuse that with the program's data flow graph.

   (*a*) data flow testing (*b*) data flow model

   (*c*) data flow machines (*d*) data dictionary

5. The purpose of object oriented analysis is to develop solution models that satisfy the customer requirement.

   (*a*) True (*b*) False

3.4 Prototyping

Prototyping is a process that enables developer to create a small model of software. The standard defines prototype as a preliminary form or instance of a system that serves as a model for later stages for the final complete version of the system.

A prototype may be categorized as follows:

1. A paper prototype, which is a model depicting human machine interaction in a form that makes the user understand how such interaction, will occur.
2. A working prototype implementing a subset of complete features.
3. An existing program that performs all of the desired functions but additional features is added for improvement.

Prototype is developed so that customers, users and developers can learn more about the problem. Thus, prototype serves as a mechanism for identifying software requirements. It allows the user to explore or criticize the proposed system before developing a full scale system.

3.4.1 Types of Prototype

The prototypes are developed using the following two techniques:

Throw Away Prototype

In this technique, the prototype is discarded once its purpose is fulfilled and the final system is built from scratch. The prototype is built quickly to enable the user to rapidly interact with a working system. As the prototype has to be ultimately discarded, the attention on its speed, implementation aspects, maintainability and fault tolerance is not paid. In requirements defining phase, a less refined set of requirements are hurriedly defined and throw away prototype is constructed to determine the feasibility of requirement, validate the utility of function, uncover missing requirements, and establish utility of user interface. The duration of prototype building should be as less as possible because its advantage exists only if results from its use are available in timely fashion.

Evolutionary Prototype

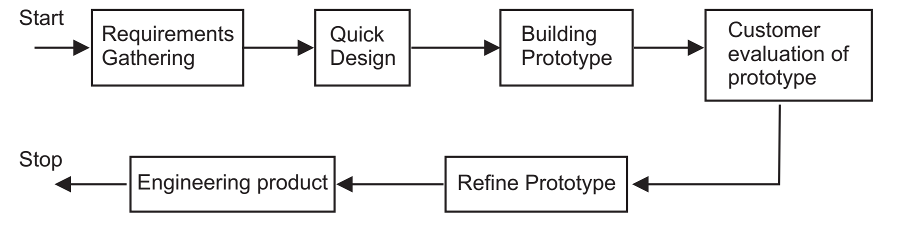
In this, the prototype is constructed to learn about the software problems and their solutions in successive steps. The prototype is initially developed to satisfy few requirements. Then, gradually, the requirements are added in the same prototype leading to the better understanding of software system. The prototype once developed is used again and again. This process is repeated till all requirements are embedded in this and the complete system is evolved.

According to SOMM the benefits of developing prototype are listed:

1. Communication gap between software developer and clients may be identified.
2. Missing user requirements may be unearthed.
3. Ambiguous user requirements may be depicted.
4. A small working system is quickly built to demonstrate the feasibility and usefulness of the application to management.
5. It serves as basis for writing the specification of the system.

The sequence of prototyping is seeing in following Figure 3.4.

Figure 3.4: Sequence of Prototyping



3.4.2 Problems of Prototyping

In some organizations, the prototyping is not as successful as anticipated. A common problem with this approach is that people expect much from insufficient effort. As the requirements are loosely defined, the prototype sometimes gives misleading results about the working of software. Prototyping can have execution inefficiencies and this may be questioned as negative aspect of prototyping. The approach of providing early feedback to user may create the impression on user and user may carry some negative biasing for the completely developed software also.

Software Metrics

Measurement is fundamental to any engineering discipline and software engineering is no exception. Software metric is a quantitative measure derived from the attribute of software development life cycle. It behaves as software measures. A software measure is a mapping from a set of objects in the software engineering world into a set of mathematical constructs such as numbers or vectors of numbers. Using the software metrics, software engineer measures software processes, and the requirements for that process. The software measures are done according to the following parameters

1. The objective of software and problems associated with current activities
2. The cost of software required for relevant planning relative to future projects
3. Testability and maintainability of various processes and products
4. Quality of software attributes like reliability, portability and maintainability, Utility of software product, User friendliness of a product.

Various characteristics of software measures define:

Objects of Measurement: They indicate the products and processes to be measured.

Source of Measurement: It indicates who will measure the software. For example, software designer, software tester and software managers.

Property of Measurement: It indicates the attribute to be measured like cost of software, reliability, maintainability, size and portability.

Context of Measurement: It indicates the environments in which context the software measurements are applied. Common software measures, There are significant numbers of software measures. The following are a few common software measures:

Size

It indicates the magnitude of software system. It is most commonly used software measure. It is indicative measure of memory requirement, maintenance effort, and development time.

LOC

It represents the number of lines of code (LOC). It is indicative measure of size oriented software measure. There is some standardization on the methodology of counting of lines. In this, the blank lines and comments are excluded. The multiple statements present in a single line are considered as a single LOC. The lines containing program header and declarations are counted.

3.4.3 Advantages of Prototyping

The advantages of prototyping outperform the problems of prototyping. Thus, overall, it is a beneficial approach to develop the prototype. The end user cannot demand fulfilling of incomplete and ambiguous software needs from the developer. One additional difficulty in adopting this approach is the large investment that exists in software system maintenance. It requires additional planning about the re-engineering of software. Because, it may be possible that by the time the prototype is build and tested, the technology of software development is changed, hence requiring a complete re-engineering of the product.

Task Prepare a flow chart to show the steps of testing software.

3.5 Software Requirement Specification Document (SRS)

Requirements engineering is the systematic use of proven principles, techniques and language tools for the cost effective analysis, documentation, and on-going evaluation of user's needs and the specification of external behaviour of a system to satisfy those user needs. It can be defined as a discipline, which addresses requirements of objects all along a system development process. The output of requirements of engineering process is Requirements Definition Description (RDD). Requirements engineering may be defined in the context of Software Engineering. It divides the Requirements Engineering into two categories. First is the requirements definition and the second is requirements management.

Requirements definition consists of the following processes:

1. Requirements gathering.
2. Requirements analysis and modelling.
3. Creation of RDD and SRS.
4. Review and validation of SRS as well as obtaining confirmation from user.

Requirements management consists of the following processes:

1. Identifying controls and tracking requirements.
2. Checking complete implementation of RDD.
3. Manage changes in requirements which are identified later.

Notes

3.5.1 Types of Requirements

There are various categories of the requirements. On the basis of their priority, the requirements are classified into the following three types:

1. Those that should be absolutely met.
2. Those are highly desirable but not necessary.
3. Those that are possible but could be eliminated.

On the basis of their functionality, the requirements are classified into the following two types:

Functional Requirements

They define the factors like, I/O formats, storage structure, computational capabilities, timing and synchronization.

Non-functional Requirements

They define the properties or qualities of a product including usability, efficiency, performance, space, reliability, portability etc.

Software Requirements Specification

Requirements specification for a software system is a complete explanation of the behavior of a system to be urbanized and may include a set of use cases that describe communications the users will have with the software. In addition it also contains non-functional requirements. Non-functional requirements impose constraints on the design or implementation (such as performance engineering requirements, quality standards, or design constraints), and are sometimes referred to as "ilities". A software requirement is a sub-field of software engineering that deals with the elicitation, analysis, specification, and validation of requirements for software. The software requirement specification document enlists all necessary requirements for project development, to derive the requirements we need to have clear and thorough understanding of the products to be developed. This is prepared after detailed communications with project team and the customer. General organization of a SRS is as follows:

Introduction

- Purpose
- Definitions
- System overview
- References

Overall Description

- Product perspective
- Product functions
- User characteristics
- Constraints, assumptions and dependencies

Specific Requirements

- External interface requirements
- Functional requirements
- Performance requirements

- Design constraints
- Logical database requirement
- Software System attributes

3.5.2 Problems in SRS

There are various features that make requirements analysis difficult. These are discussed:

1. Complete requirements are difficult to uncover. In recent trends in engineering, the processes are automated and it is practically impossible to understand the complete set of requirements during the commencement of the project itself.
2. Requirements are continuously generated. Defining the complete set of requirements in the starting is difficult. When the system is put under run, the new requirements are obtained and need to be added to the system. But, the project schedules are seldom adjusted to reflect these modifications. Otherwise, the development of software will never commence.
3. The general trends among software developer shows that they have over dependence on CASE tools. Though these tools are good helping agents, over reliance on these Requirements Engineering Tools may create false requirements. Thus, the requirements corresponding to real system should be understood and only a realistic dependence on tools should be made.
4. The software projects are generally given tight project schedules. Pressure is created from customer side to hurriedly complete the project. This normally cuts down the time of requirements analysis phase, which frequently lead to disaster(s).
5. Requirements engineering is communication intensive. Users and developers have different vocabularies, professional backgrounds and psychology. User writes specifications in natural language and developer usually demands precise and well specified requirement.
6. In present time, the software development is market driven having high commercial aspect. The software developed should be a general purpose one to satisfy anonymous customer, and then, it is customized to suit a particular application.
7. The resources may not be enough to build software that fulfils all the customer's requirements. It is left to the customer to prioritize the requirements and develop software fulfilling important requirements.

Caution The improper management to preparing the SRS can lead to confusion in the project implementation.

3.5.3 Characteristics of SRS

To properly satisfy the basic goals, an SRS should have certain properties and should contain different types of requirements. Some of the desirable characteristics

- Correct
- Complete
- Unambiguous
- Verifiable
- Consistent
- Valid

The SRS is verifiable if and only if every stated requirement is verifiable. A requirement is verifiable if there exist some cost-effective process that can check whether the final software meets that requirement. It is consistent if there is no requirement that conflicts with another. Terminology can cause inconsistencies; for example, different requirements may use different terms to refer to the same object. There may be logical or temporal conflict between requirements that causes inconsistencies. This occurs if the SRS contains two or together by any software system. For example, suppose a requirement states that an event e is to occur before another event *f*. But then another set of requirements states (directly or indirectly by transitivity) that event *f* should occur before event e. Inconsistencies in an SRS can reflect some major problems. All these characteristics, completeness is perhaps the most important and also the most difficult property to establish. One of the most common defects in requirements specification is incompleteness. Missing requirements necessitate additions and modifications to the requirements later in the development cycle, which are often expensive to incorporate. Incompleteness is also a major source of disagreement between the client and the supplier.

Correct

The specification must define the desired capability's real world operational environment, its interface to that environment and its interaction with that environment. It is the real world aspect of requirements that is the major source of difficulty in achieving specification correctness. The real world environment is not well known for new applications and for mature applications the real world keeps changing.

Complete

A complete requirements specification must precisely define all the real world situations that will be encountered and the capability's responses to them. It must not include situations that will not be encountered or unnecessary capability features.

Unambiguous

Statement of a requirement is unambiguous if it can only be interpreted one way. This perhaps, is the most difficult attribute to achieve using natural language. The use of weak phrases or poor sentence structure will open the specification statement to misunderstandings.

Verifiable

In order to be verifiable, requirement specifications at one level of abstraction must be consistent with those at another level of abstraction. Most, if not all, of these attributes are subjective and a conclusive assessment of the quality of a requirements specification requires review and analysis by technical and operational experts in the domain addressed by the requirements.

Consistent

System functions and performance level must be compatible and the required quality features (reliability, safety, security, etc.) must not contradict the utility of the system. For example, the only aircraft that is totally safe is one that cannot be started, contains no fuel or other liquids, and is securely tied down.

Valid

To validate a requirements specification all the project participants, managers, engineers and customer representatives, must be able to understand, analyze and accept or approve it. This is the primary reason that most specifications are expressed in natural language.

3.5.4 Components of SRS

Component-based software development move towards is based on the idea to expand software systems by selecting suitable off-the-shelf mechanism and then to assemble them with a well-defined software architecture. Because the new software development paradigm is much different from the traditional approach, quality assurance (QA) for component-based software development is a new topic in the software engineering community.

Standards Compliance

This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit requirements which may require logging of operations.

Hardware Limitations

The software may have to operate on some existing or predetermined hardware, thus imposing restrictions on the design. Hardware limitations can include the type of machines to be used, operating system available on the system, languages supported, and limits on primary and secondary storage. Reliability and Fault Tolerance: Fault Tolerance requirements can place a major constraint on how the system is to be designed, as they make the system more complex and expensive. Recovery requirements are often an integral part here, detailing what the system should do if some failure occurs to ensure certain properties.

3.5.5 Specification Language

A requirements requirement for an in order system is important for several reasons; it serves as a means of communication between the consumer and the systems developer. it represents in a systematic fashion the present state of the real world, its problems and its future requirements; it enables the systems developer to turn real world problems into other forms which are more manageable in terms of size, complexity, human understanding and computer process ability; it serves as the basis for the design, implementation, testing and maintenance of the target system. In order that all the objectives of a requirements specification are met, we need a powerful specification language. Information systems development can be conceived of as an engineering process. We must first of all build a model, which is a small-scaled abstract representation of the real world. All unnecessary details in the physical world which are irrelevant to the engineering process are removed from the model, or in other words, ignored during the analysis stage. When a bridge or tunnel is planned between two nations, for instance, the political issues should best be dealt with by politicians and not form part of the engineer's requirements specification. If the resulting model is still too complex, further abstractions may be necessary, until the problem is reduced to a manageable size. The model is then analyzed and manipulated until a feasible solution is found. In engineering, diagrams and mathematics are often used because they have e been found to be more suitable for manipulation than verbal descriptions. One representation may have to be transformed into another so that the most appropriate model for a given analysis can be used. Diagrams, for instance, may have to be converted into equations. Finally, if the abstract solution is accepted by the customer, a construction phase turns it into a real system. In order for a requirements specification to be useful in systems. In order for a requirements specification to be useful in systems development, seen as an engineering process, the specification language must exhibit various features, each being relevant to one of the stages. These features will be highlighted in this section. We recognize that there are authors who may object to having an engineer's view of information systems development. Understanding the impacts of change, people-oriented design and user participation throughout the development process. It would be interesting to study how our proposal for the desirable features of a specification language fits into this alternative framework.

3.5.6 Structure of Document

Design document structures so that the dissimilar parts are as self-governing as possible. This allows each part to be read as a single item and reduces problems of cross-referencing when changes have to be made. Structuring a document properly also allows readers to find in sequence more easily. As well as document components such as contents lists and indexes, well-structured documents can be skim read so that readers can quickly locate sections or sub-sections that are of most interest to them. Structure of a document should include the components the standard makes clear that these are desirable or essential features of a document but makes clear that the ways in which these components are provided depends on the designers of the documentation. Some (such as a table of contents) are clearly separate sections; other components such as navigational features will be found throughout the document. However, many organizations will use the standard as a guide and will not necessarily include all of the components in such circumstances, there are some minimal structuring guidelines that we believe should always be followed, all documents, however short, should have a cover page which identifies the project, the document, the author, the date of production, the type of document, configuration management and quality assurance information, the intended recipients of the document, and the confidentiality class of the document. It should also include information for document retrieval (an abstract or keywords) and a copyright notice. Requirements have to be specified using some specification language. Though formal notations exist for specifying specific properties of the system, natural languages are now most often used for specifying requirements. When formal languages are employed, they are often used to specify particular properties or for specific parts of the system, as part of the overall SRS. All the requirements for a system, stated using a formal notation or natural language, have to be included in a document that is clear and concise. For this, it is necessary to properly organize the requirements document. Here we discuss the organization based on the IEEE guide to software requirements specification.

The detailed requirements section describes the details of the requirements that a developer needs to know for designing and developing the system. This is typically the largest and most important part of the document. For this section, different organizations have been suggested in the standard. These requirements can be organized by the modes of operation, user class, object, feature, stimulus, or functional hierarchy one method to organize the specific requirements is to first specify the external interfaces, followed by functional requirements, performance requirements, design constraints, and system attributes. The external interface requirements section specifies all the interfaces of the software: to people, other software, hardware, and other systems. User interfaces are clearly a very important component; they specify each human interface the system plans to have, including screen formats, contents of menus, and command structure. In hardware interfaces, the logical characteristics of each interface between the software and hardware on which the software can run are specified. Essentially, any assumptions the software is making about the hardware are listed here. In software interfaces, all other software that is needed for this software to run is specified, along with the interfaces. Communication interfaces need to be specified if the software communicates with other entities in other machines.

Task Prepare a DFD for the flow of data in an organization.

Case Study

Requirements Specification to the Software Process with ACUS

Improving the software process to improve overall software development has been an on-going endeavour for both industrial practitioners and academics for many years. In software engineering, and in particular the software process area, issues relating to requirements engineering (RE) have been repeatedly cited. Requirements specification is regarded as a critical stage of software development, with the claim that software development problems could be better addressed with Bgood RE practice, While software engineering is benefiting from the development of models and standards for software process improvement and assessment such as the ISO 9001 standard for quality management systems, and the

Contd...

Software Engineering Institute's Capability Maturity Model for Software (CMM), the relatively new field of RE does not enjoy well-established, proven strategies for improving or assessing the requirements process. This leads to a strong need for assessment and measurement of effects of rigorous RE practice in software development. Many practical guides naturally focus on the RE process within the larger software development process, but deliberately present their material vaguely, reminding practitioners that good requirements engineering (process) depends on the organization, its development process, its tools and particular circumstances, or even that every project needs a different process. These sentiments offer little comfort to the practitioner. These same guides warn that revolutionary change is not practical; that instead careful evolutionary improvements are more fruitful. Practitioners are encouraged to measure results to gauge effectiveness, but topics on empirical assessment are largely left to the imagination, as an exercise for the reader. Further, the role of requirements engineering in software development has been discussed in the literature as important in planning activities such as: determining the nature of the problem, exploring solutions through feasibility studies, and ultimately deciding precisely what to build. It is notified that the opportunity for RE to improve all subsequent stages of the development life-cycle, ultimately leading to broader improvements in software quality and user satisfaction. An important note is that the RE process improvements at ACUS occurred in parallel with initiatives for improvement in other processes in the organization, such as software quality assurance, project planning and project tracking.

Questions

1. What is the reengineering process?

2. What are its advantages and disadvantages?

Self Assessment Questions

6. Software Engineering ………… provides the technical "how to" for Building the software

   (*a*) Tools (*b*) Methods

   (*c*) Data (*d*) Process

7. A good specification should be?

   (*a*) Unambiguous (*b*) Distinctly Specific

   (*c*) Functional (*d*) All of these.

8. If every requirement stated in the Software Requirement Specification (SRS) has only one interpretation, SRS is said to be:

   (*a*) Correct (*b*) unambiguous.

   (*c*) Consistent (*d*) verifiable.

9. SRS is also known as specification of

   (*a*) White box testing (*b*) Stress testing

   (*c*) Integrated testing (*d*) Black box testing

10. The feature of the object oriented paradigm which helps code reuse is:

    (*a*) Object (*b*) class

    (*c*) Inheritance (*d*) aggregation

11. The testing that focuses on the variables is called:
    (*a*) Black box testing (*b*) White box testing
    (*c*) Data variable testing (*d*) Data flow testing
12. Which is not a step of requirement engineering?
    (*a*) Requirements elicitation (*b*) Requirements analysis
    (*c*) Requirements design (*d*) Requirements documentation
13. What types of models are created during software requirements analysis?
    (*a*) Functional and behavioural (*b*) Algorithmic and data structure
    (*c*) Architectural and structural (*d*) Usability and reliability
14. If every requirement can be checked by a cost-effective process, then the SRS is:
    (*a*) Verifiable (*c*) Traceable
    (*c*) Modifiable (*d*) Complete
15. Requirements can be refined using:
    (*a*) The waterfall model (*b*) Prototyping model
    (*c*) The evolutionary model (*d*) Spiral model

3.6 Summary

- Software testing is "the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.
- A state of a system represents some conditions about the system.
- The SRS is to be validated and the feedback from the validation activity may require further analysis or specification.
- The object-oriented model is based on a collection of objects. An object contains values stored in instance variables within the object.
- The object oriented design converts the object oriented analysis model into a design model. This serves an outline for software construction.
- The prototype is constructed to learn about the software problems and their solutions in successive steps. The prototype is initially developed to satisfy few requirements.
- The general trends among software developer shows that they have over dependence on CASE tools.
- A requirements specification for an information system is important for several reasons; it serves as a means of communication between the user and the systems developer.

3.7 Keywords

Consistent: System functions and performance level must be compatible and the required quality features (reliability, safety, security, etc.) must not contradict the utility of the system.

Data Flow Diagrams: It is a graphical representation of flow of data through a system. It pictures a system as a network of functional processes. The basis of DFD is a data flow graph.

Functional Requirements: They define the factors like, I/O formats, storage structure, computational capabilities, timing and synchronization.

Multi-data Machines: These machines can fetch several instructions and objects in parallel. They can also do arithmetic and logical operations simultaneously on different data objects.

Objects of Measurement: They indicate the products and processes to be measured.

Size: It indicates the magnitude of software system. It is most commonly used software measure. It is indicative measure of memory requirement, maintenance effort, and development time.

Standards Compliance: This specifies the requirements for the standards the system must follow. The standards may include the report format and accounting procedures. There may be audit requirements which may require logging of operations.

Lab Exercise

1. Create a data flow diagram to prepare a software design.
2. Prepare a SRS for a software project.

3.8 Review Questions

1. What is the requirements specification language?
2. Defined SRS.
3. Explain is representation of prototyping.
4. What is object oriented design and analysis model?
5. Define shadowing technique.
6. What is the data flow diagram?
7. Discuss components of the data flow model.
8. What is the structured requirements definition?
9. What is data dictionary?
10. What is the object methodology?

Answers for Self Assessment Questions

1.	(*b*)	2.	(*d*)	3.	(*c*)	4.	(*b*)	5.	(*a*)
6.	(*b*)	7.	(*d*)	8.	(*b*)	9.	(*d*)	10.	(*c*)
11.	(*a*)	12.	(*c*)	13.	(*a*)	14.	(*a*)	15.	(*b*)

3.9 Further Readings

Books

Software Requirements by Karl E. Wiegers

Online link

http://books.google.co.in/books?id=WcO3Ca9NuvQC&printsec=frontcover&dq=Software+Requirements&hl=en&sa=X&ei=0qkGUMv_PMuxrAeTlJjKBg&ved=0CDMQ6AEwAA#v=onepage&q=Software%20Requirements&f=false

Notes

Unit 4: Introduction to Validation, Metrics

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the function point and quality metrics
- Describe the software architecture
- Discuss the architecture views
- Define the architecture styles

Introduction

The process of provided that confirmation that the software and its connected products satisfy system requirements allocated to software at the end of each life-cycle action, solve the accurate problem (e.g., correctly model physical laws, implement business rules, use proper system assumptions), and satisfy intended use and user needs.

A more generic definition of corroboration is "the steps and the process needed to ensure that the system configuration, as designed, meets all requirements initially specified by the customer."

The latter definition allows for more flexibility in the interpretation of validation; it answers the question of "Are we providing what the customer wants?" However, both definitions assume the existence of correct requirements and focus on the concept of building the right product as defined by these requirements.

Software validation has always been problematic in software engineering. Unlike many other engineered products, software often cannot be visualized, thus, in many cases, resulting in software validation being a reactive last minute process. It is hard for validation to be a continuous process throughout development as with many physical systems when there is no physical product to monitor. The ultimate validation of software systems is effectively the operational evaluation of the system by the user, and often this is where validation of the system is relegated. The IEEE definition lends itself to this approach as it focuses on the satisfaction of requirements through testing. Quite simply, validation of software has not had the same rigorous research and development of processes as other areas of software engineering, particularly verification. Validation is a relatively misunderstood process.

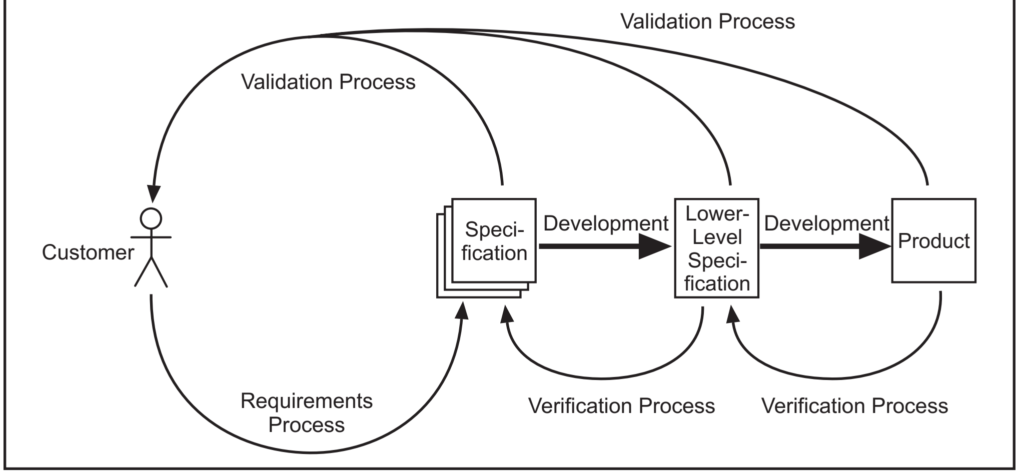
The software engineering discipline has become competent in the area of verification. We can build portions of systems to their applicable specifications with relative success. However, we still build systems that do not meet customer's expectations and requirements. One of the key tools to address this situation is validation. Significant efforts afforded to software validation are now a priority for software engineering. More proactive, rather than the typical reactive, solutions are being sought. The IEEE definition of validation indicates that it is a process to be carried out at the end of each phase (or lifecycle activity); however, this should only be the finalization, or completion, of validation. The validation process should be a proactive and continuous process to be carried out prior to, and in parallel with, the development and verification activities with closure at the end of each phase.

Although validation focuses on ensuring that initial customer requirements are met, there is more to validation than meets the eye. Validation is required whenever a requirements derivation process occurs (i.e., a translation of requirements from one domain to another). An example of this is taking a customer's requirements in their natural language and translating them into a specification. The specification needs to be validated to ensure that it maps back to the cognitive understanding of the stakeholders who originally supplied the requirements. To ensure the traceability of products for validation, the validation process is ongoing throughout the development cycle whenever this translation of requirements takes place. Any higher-level requirement being translated to a lower-level requirement requires a validation process to ensure that the products of the lower-level requirements are indeed valid. In contrast, verification is defined as "The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

The key difference between validation and verification is that verification simply ensures that requirements for a given phase are met. Validation ensures that overall customer requirements (i.e., customer expectations) are met. There is somewhat of an overlap in validation and verification processes, particularly when considering either process in the "middle-levels" of abstraction. They are in fact processes orthogonal to one another. In its purest form, validation ensures that customer expectations are met; failure to meet these expectations (assuming they are constant throughout the project) indicates a failure in the validation process. Figure 4.1 shows graphically how verification and validation contrast.

The end-result of any validation process should be actionable data presented as feedback to the many different customers, or stakeholders, of the system. This effectively creates a feedback loop from any stage in the development process that allows the customer to clarify their expectation of system behaviour.

Figure 4.1: The Verification and Validation Process



Metrics

IEEE Standard defines a metric as "a quantitative measure of the degree to which a system, component or process possesses a given attribute."

Further, a software quality metric is "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which the software possesses a given quality attribute.

Software metrics are therefore quantitative measurements of product (system or component), process, or even project (in this case software projects) indicating the quality of a desired attribute. However, software metrics can be concerned with more than just quantitative measurements. Since we are measuring quality of product, process, or project, qualitative aspects must be considered. Metrics can also be qualitative in nature.

Metrics are Measurements of Quality: In most well established engineering disciplines candidate measurement attributes are well understood. Software engineering is one of the disciplines outside of the "most" category. Software engineering is a young engineering discipline and as such does not have the hundreds of years of empirical scientific foundation as other core engineering disciplines. Partly due to this fact, many measures of software are not well understood or are ill-defined.

Measurement: In the most general sense, is the mapping of numbers to attributes of objects in accordance with some prescribed rule. The mapping must preserve intuitive and empirical observations about the attributes and entities.

Empirical Observation Requires Experimentation: Therefore, strictly speaking, for software metric to be valid it must be based on empirical observation through experimentation, whether qualitative or quantitative.

Often with software development, the single best metric is sought. Given the above definitions, such an approach comes across as foolish. An engineered product cannot be properly understood purely through the application of one measurement. Single software metric can only present a single view of a software product. Multiple metrics are required to sufficiently understand a product or process. Knowing a car's weight gives no indication of overall performance, it is only one piece of the picture. It is also possible that the gathered metric does not answer a question of interest or provide any decision support value. The same is true for software.

Many definitions of metrics require them to be quantitative. However, many qualitative measures of product, process, or project are also valid. Consider a fast car. The "fast" attribute indicates its relative quality. It is a qualitative metric, understood in a qualitative manner. Qualitative metrics are relative to some quantitative measure; however, exact position on the quantitative scale may not be possible with the given data due to the use of a weak ordering. A qualitative metric still provides relevant indication of quality. This is an important aspect of metrics for software since the abstract nature of software can often preclude quantitative measures of certain quality attributes.

4.1 Function Point and Quality Metrics

Software systems, except they are thoroughly unspoken, can be like an ice berg. They are flattering more and more difficult to understand. Development of coding tools allows software developers to manufacture large amounts of software to get together an ever increasing need from users. As systems grow a method to understand and converse size needs to be used. Function Point Analysis is a structured technique of problem solving. It is a method to break systems into smaller components, so they can be better understood and analyzed.

4.1.1 Function Point Analysis

Function points are a unit calculate for software a great deal like an hour is to measuring time, miles are to measuring distance or Celsius is to measuring temperature. Function Points are an ordinal measure much like other measures such as kilometres, Fahrenheit, hours, so on and so forth.

Human beings solve problems by breaking them into smaller understandable pieces. Problems that may appear to be difficult are simple once they are broken into smaller parts dissected into classes. Classifying things, placing them in this or that category is a familiar process. Everyone does it at one time or another shopkeepers when they take stock of what is on their shelves, librarians when they catalogue books, secretaries when they file letters or documents. When objects to be classified are the contents of systems, a set of definitions and rules must be used to place these objects into the appropriate category, a scheme of classification. Function Point Analysis is a structured technique of classifying components of a system. It is a method to break systems into smaller components, so they can be better understood and analyzed. It provides a structured technique for problem solving.

In the world of Function Point Analysis, systems are divided into five large classes and general system characteristics. The first three classes or components are External Inputs, External Outputs and External Inquires each of these components transact against files therefore they are called transactions. The next two Internal Logical Files and External Interface Files are where data is stored that is combined to form logical information. The general system characteristics assess the general functionality of the system.

Function Point Analysis was an attempt to overcome difficulties associated with lines of code as a measure of software size, and to assist in developing a mechanism to predict effort associated with software development. The method was first published in 1979, then later in 1983. In 1984 Albrecht refined the method and since 1986, when the International Function Point User Group (IFPUG) was set up, several versions of the Function Point Counting Practices Manual have been published by IFPUG.

Objectives of Function Point Analysis

Frequently the term end user or user is used without specifying what is meant. This container, the user is a sophisticated user. Someone that would understand the system from a functional perspective more than likely someone that would provide requirements or does acceptance testing.

Since Function Points measures systems from a functional viewpoint they are self-governing of technology. Regardless of language, growth method, or hardware platform used, the number of function points for a system will remain constant. The only variable is the amount of effort needed to deliver a given set of function points; therefore, Function Point Analysis can be used to determine whether a tool, an environment, a language is more productive compared with others within an organization or among organizations. This is a critical point and one of the greatest values of Function Point Analysis.

Function Point Analysis can provide a mechanism to track and monitor scope creep. Function Point Counts at the end of requirements, analysis, design, code, testing and implementation can be compared. The function point count at the end of requirements and/or designs can be compared to function points actually delivered. If the project has grown, there has been scope creep. The amount of growth is an indication of how well requirements were gathered by and/or communicated to the project team. If the amount of growth of projects declines over time it is a natural assumption that communication with the user has improved.

Characteristic of Quality Function Point Analysis

Function Point examination should be performed by taught and knowledgeable workers. If Function Point Analysis is conducted by untaught personnel, it is sensible to take for granted the analysis will do incorrectly. The personnel including function points should utilize the most current version of the Function Point Counting Practices Manual.

Current application documentation should be utilized to complete a function point count. For example, screen formats, report layouts, listing of interfaces with other systems and between systems, logical and/or preliminary physical data models will all assist in Function Points Analysis.

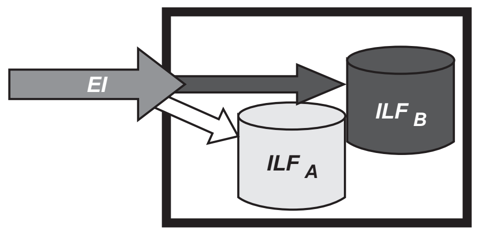
The task of counting function points should be included as part of the overall project plan. That is, counting function points should be scheduled and planned. The first function point count should be developed to provide sizing used for estimating.

The Five Major Components

Since it is ordinary for computer systems to interrelate with other computer systems, a boundary must be haggard around each system to be deliberate prior to classifying components. This boundary must be drawn according to the user's point of view. In short, the boundary indicates the border between the project or application being calculated and the external applications or user domain. Once the border has been established, components can be classified, ranked and tallied.

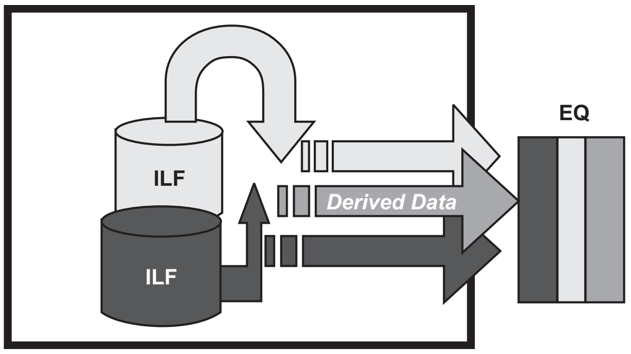
External Inputs (EI): It is an elementary process in which data crosses the boundary from outside to inside. This data may come from a data input screen or another application. The data may be used to maintain one or more internal logical files. The data can be either control information or business information. If the data is control information it does not have to update an internal logical file. The graphic represents a simple EI that updates 2 ILF's (FTR's). (See Figure 4.1)

Figure 4.1: External Inputs



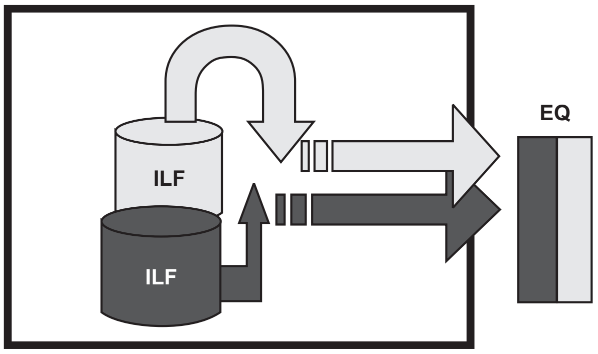
External Outputs (EO): This is an elementary process in which derived data passes across the boundary from inside to outside. Additionally, an EO (See Figure 4.2) may update an ILF. The data creates reports or output files sent to other applications. These reports and files are created from one or more internal logical files and external interface file. The following graphic represents on EO with 2 FTR's there is derived information (green) that has been derived from the ILF's.

Figure 4.2: External Outputs



External Inquiry (EQ): It is an elementary process with both input and output components that result in data retrieval from one or more internal logical files and external interface files. The input process does not update any Internal Logical Files, and the output side does not contain derived data. The Figure 4.3 represents an EQ with two ILF's and no derived data.

Figure 4.3: External Inquiry



Internal Logical Files (ILF's): A user identifiable group of logically related data that resides entirely within the applications boundary and is maintained through external inputs.

External Interface Files (EIF's): A user identifiable group of logically related data that is used for reference purposes only. The data resides entirely outside the application and is maintained by another application. The external interface file is an internal logical file for another application.

After the components have been classified as one of the five major components (EI's, EO's, EQ's, ILF's or EIF's), a ranking of low, average or high is assigned. For transactions (EI's, EO's, EQ's) the ranking is based upon the number of files updated or referenced (FTR's) and the number of data element types (DET's). For both ILF's and EIF's files the ranking is based upon record element types (RET's) and data element types (DET's). A record element type is a user recognizable subgroup of data elements within an ILF or EIF. A data element type is a unique user recognizable, non recursive, field.

Each of the following tables assists in the ranking process (the numerical rating is in parentheses). For example, an EI that references or updates two File Types Referenced (FTR's) and has seven data elements would be assigned a ranking of average and associated rating of 4. Where FTR's are the combined number of Internal Logical Files (ILF's) referenced or updated and External Interface Files referenced.

EI Table

Figure 4.4: EI Table

FTR's	DATA ELEMENTS		
	1-4	5-15	>15
0-1	Low	Low	Ave
2	Low	Ave	High
3 or more	Ave	High	High

Shared EO and EQ Table

Figure 4.5: Shared EO and EQ Table

FTR's	DATA ELEMENTS		
	1-5	6-19	>19
0-1	Low	Low	Ave
2-3	Low	Ave	High
>3	Ave	High	High

Values for transactions

Figure 4.6: Values for Transactions

Rating	VALUES		
	EO	EQ	EI
Low	4	3	3
Average	5	4	4
High	7	6	6

Like all components, are rated and scored. Basically, an EQ is rated (Low, Average or High) like an EO, but assigned a value like and EI. The rating is based upon the total number of unique (combined unique input and out sides) data elements and the file types referenced (combined unique input and output sides). If the same FTR is used on the input and output side, then it is counted only one time. If the same DET is used on the input and output side, then it is only counted one time.

For both ILF's and EIF's the number of record element types and the number of data elements types are used to determine a ranking of low, average or high. A Record Element Type is a user recognizable subgroup of data elements within an ILF or EIF. A Data Element Type (DET) is a unique user recognizable, non recursive field on an ILF or EIF.

The counts for each level of complexity for each type of component can be entered into a table such as the following one. Each count is multiplied by the numerical rating shown to determine the rated value. The rated values on each row are summed across the table, giving a total value for each type of component. These totals are then summed across the table, giving a total value for each type of component. These totals are then summoned down to arrive at the Total Number of Unadjusted Function Points.

Notes

Figure 4.7: Data Element Type

FTR's	DATA ELEMENTS		
	1-19	20-50	>50
1	Low	Low	Ave
2-5	Low	Ave	High
>5	Ave	High	High

Rating	Values	
	ILF	ELF
Low	7	5
Average	10	7
High	15	10

Figure 4.8: Complexity for Each Type of Component

Type of Component	Complexity of Components			
	Low	Average	High	Total
External Inputs	___ × 3 = ___	___ × 4 = ___	___ × 6 = ___	
External Output x	___ × 4 = ___	___ × 5 = ___	___ × 7 = ___	
External Inquiries	___ × 3 = ___	___ × 4 = ___	___ × 6 = ___	
Internal Logical Files	___ × 7 = ___	___ × 10 = ___	___ × 15 = ___	
External Interface Files	___ × 5 = ___	___ × 7 = ___	___ × 10 = ___	
	Total Number of Unadjusted Function Points			
	Multiplied Value Adjustment Factor			
	Total Adjusted Function Points			

The value adjustment factor (VAF) is based on 14 general system characteristics (GSC's) that rate the general functionality of the application being counted. Each characteristic has associated descriptions that help determine the degrees of influence of the characteristics. The degrees of influence range on a scale of zero to five, from no influence to strong influence. The IFPUG Counting Practices Manual provides detailed evaluation criteria for each of the GSC'S, the Table 4.1 below is intended to provide an overview of each GSC.

Table 4.1: 14 General System Characteristics

	General System Characteristic	Brief Description
1.	Data communications	How many communication facilities are there to aid in the transfer or exchange of information with the application or system?
2.	Distributed data processing	How are distributed data and processing functions handled?
3.	Performance	Was response time or throughput required by the user?

Notes

4.	Heavily used configuration	How heavily used is the current hardware platform where the application will be executed?
5.	End-user efficiency	How frequently are transactions executed daily, weekly, monthly, etc.?
6.	On-Line data entry	What percentage of the information is entered On-Line?
7.	End-user efficiency	Was the application designed for end-user efficiency?
8.	On-Line update	How many ILF's are updated by On-Line transaction?
9.	Complex processing	Does the application have extensive logical or mathematical processing?
10.	Reusability	Was the application developed to meet one or many user's needs?
11.	Installation ease	How difficult is conversion and installation?
12.	Operational ease	How effective and/or automated are start-up, back-up, and recovery procedures?
13.	Multiple sites	Was the application specifically designed, developed, and supported to be installed at multiple sites for multiple organizations?
14.	Facilitate change	Was the application specifically designed, developed, and supported to facilitate change?

Once all the 14 GSC's have been answered, they should be tabulated using the IFPUG Value Adjustment Equation (VAF) 14 where: Ci = degree of influence for each General System Characteristic

VAF = 0.65 + [(Ci) / 100] .i = is from 1 to 14 representing each GSC.

i =1 Ã¥ = is summation of all 14 GSC's.

The final Function Point Count is obtained by multiplying the VAF times the Unadjusted Function Point (UAF).

FP = UAF * VAF

4.1.2 Software Quality Metrics

We best manage what we can calculate measurement enables the association to get better the software process; assist in planning, tracking and controlling the software project and assess the quality of the software thus produced. It is the measure of such specific attributes of the process, project and product that are used to compute the software metrics. Metrics are analyzed and they provide a dashboard to the management on the overall health of the process, project and product. Generally, the validation of the metrics is a continuous process spanning multiple projects. The kind of metrics employed generally account for whether the quality requirements have been achieved or are likely to be achieved during the software development process. As a quality assurance process, a metric is needed to be revalidated every time it is used. Two leading firms namely, IBM and Hewlett-Packard have placed a great deal of importance on software quality. The IBM measures the user satisfaction and software acceptability in eight dimensions which are capability or functionality, usability, performance, reliability, ability to be installed, maintainability, documentation, and availability. For the Software Quality Metrics the Hewlett-Packard normally follows the five general quality parameters namely the functionality, the usability, the reliability, the performance and the serviceability. In general, for most software quality assurance systems the common software metrics that are checked for improvement are the Source lines of code, cyclical complexity of the code, Function point analysis, bugs per line of code, code coverage, number of classes and interfaces, cohesion and coupling between the modules etc.

Common software metrics include:

- Bugs per line of code
- Code coverage
- Cohesion
- Coupling
- Cyclomatic complexity
- Function point analysis
- Number of classes and interfaces
- Number of lines of customer requirements
- Order of growth
- Source lines of code
- Robert Cecil Martin's software package metrics

Software Quality Metrics focus on the process, project and product. By analyzing the metrics the organization the organization can take corrective action to fix those areas in the process, project or product which are the cause of the software defects.

The de-facto definition of software quality consists of the two major attributes based on intrinsic product quality and the user acceptability. The software quality metric encapsulates the two attributes, addressing the mean time to failure and defect density within the software components. Finally it assesses user requirements and acceptability of the software. The intrinsic quality of a software product is generally measured by the number of functional defects in the software, often referred to as bugs, or by testing the software in run time mode for inherent vulnerability to determine the software "crash" scenarios. In operational terms, the two metrics are often described by terms namely the defect density (rate) and mean time to failure (MTTF).

Although there are many measures of software quality, correctness, maintainability, integrity and usability provide useful insight.

Correctness

A program must operate correctly. Correctness is the degree to which the software performs the required functions accurately. One of the most common measures is Defects per KLOC. KLOC means thousands (Kilo) Of Lines of Code.) KLOC is a way of measuring the size of a computer program by counting the number of lines of source code a program has.

Maintainability

Maintainability is the ease with which a program can be correct if an error occurs. Since there is no direct way of measuring this indirect way has been used to measure this. MTTC (Mean time to change) is one such measure. It measures when a error is found, how much time it takes to analyze the change, design the modification, implement it and test it.

Integrity

This measure the system's ability to with stand attacks to its security. In order to measure integrity two additional parameters are threat and security need to be defined.

Threat: The probability that an attack of certain type will happen over a period of time.

Security: The probability that an attack of certain type will be removed over a period of time.

Integrity = Summation [(1 - threat) X (1 - security)]

Usability

How usable is your software application? This important characteristic of your application is measured in terms of the following characteristics:

- Physical / Intellectual skill required to learn the system.
- Time required becoming moderately efficient in the system.
- The net increase in productivity by use of the new system.
- Subjective assessment(usually in the form of questionnaire on the new system)

Standard for the Software Evaluation

In context of the Software Quality Metrics, one of the popular standards that address the quality model, external metrics, internal metrics and the quality in use metrics for the software development process is ISO 9126.

Defect Removal Efficiency

Defect Removal Efficiency (DRE) is a measure of the efficacy of your SQA activities. For e.g. If the DRE is low during analysis and design, it means you should spend time improving the way you conduct formal technical reviews.

$$DRE = E / (E + D)$$

Where E = No. of Errors found before delivery of the software and D = No. of Errors found after delivery of the software.

Ideal value of DRE should be 1 which means no defects found. If you score low on DRE it means to say you need to re-look at your existing process. In essence DRE is indicator of the filtering ability of quality control and quality assurance activity. It encourages the team to find as many defects before they are passed to the next activity stage. Some of the Metrics are listed out here:

Test Coverage	=	Number of units (KLOC/FP) tested/total size of the system
Number of tests per unit size	=	Number of test cases per KLOC/FP
Defects per size	=	Defects detected/system size
Cost to locate defect	=	Cost of testing/the number of defects located
Defects detected in testing	=	Defects detected in testing/total system defects
Defects detected in production	=	Defects detected in production/system size
Quality of Testing	=	No. of defects found during Testing/(No. of defects found during testing + No. of acceptance defects found after delivery) *100
System complaints	=	Number of third party complaints/Number of transactions processed Effort Productivity = Test Planning Productivity = No. of Test cases designed/Actual Effort for Design and Documentation
Test Execution Productivity	=	No. of Test cycles executed/Actual Effort for testing
Test efficiency	=	(Number of tests required/The Number of system errors)

Notes

Table 4.2: Measures and Metrics

Measure	Metrics
1. Customer satisfaction index	Number of system enhancement requests per year Number of maintenance fix requests per year User friendliness: call volume to customer service hotline User friendliness: training time per new user Number of product recalls or fix releases (software vendors) Number of production re-runs (in-house information systems groups)
2. Delivered defect quantities	Normalized per function point (or per LOC) At product delivery (first 3 months or first year of operation) Ongoing (per year of operation) By level of severity By category or cause, e.g.: requirements defect, design defect, code defect, documentation/ on-line help defect, defect introduced by fixes, etc.
3. Responsiveness (turnaround time) to users	Turnaround time for defect fixes, by level of severity Time for minor vs. major enhancements; actual vs. planned elapsed time (by customers) in the first year after product delivery
4. Complexity of delivered product	McCabe's cyclomatic complexity counts across the system Halstead's measure Card's design complexity measures Predicted defects and maintenance costs, based on complexity measures
5. Cost of defects	Business losses per defect that occurs during operation Business interruption costs; costs of work-arounds Lost sales and lost goodwill Litigation costs resulting from defects Annual maintenance cost (per function point) Annual operating cost (per function point) Measurable damage to your boss's career
6. Costs of quality activities	Costs of reviews, inspections and preventive measures Costs of test planning and preparation Costs of test execution, defect tracking, version and change control Costs of diagnostics, debugging and fixing Costs of tools and tool support Costs of tools and tool support Costs of test case library maintenance Costs of testing & QA education associated with the product Costs of monitoring and oversight by the QA organization (if separate from the development and test organizations)
7. Re-work	Re-work effort (hours, as a percentage of the original coding hours) Re-worked LOC (source lines of code, as a percentage of the total delivered LOC) Re-worked software components (as a percentage of the total delivered components)
8. Reliability	Availability (percentage of time a system is available, versus the time the system is needed to be available) Mean time between failure (MTBF) Mean time to repair (MTTR) Reliability ratio (MTBF / MTTR) Number of product recalls or fix releases Number of production re-runs as a ratio of production runs
9. Test coverage	Breadth of functional coverage Percentage of paths, branches or conditions that were actually tested Percentage by criticality level: perceived level of risk of paths The ratio of the number of detected faults to the number of predicted faults.

Notes

Did u know? Function Point Analysis was developed first by Allan J. Albrecht in the mid 1970s.

Caution Ideal value of DRE must be 1 which means no defects found. The value other than 1 assumed as software failure.

4.2 Software Architecture

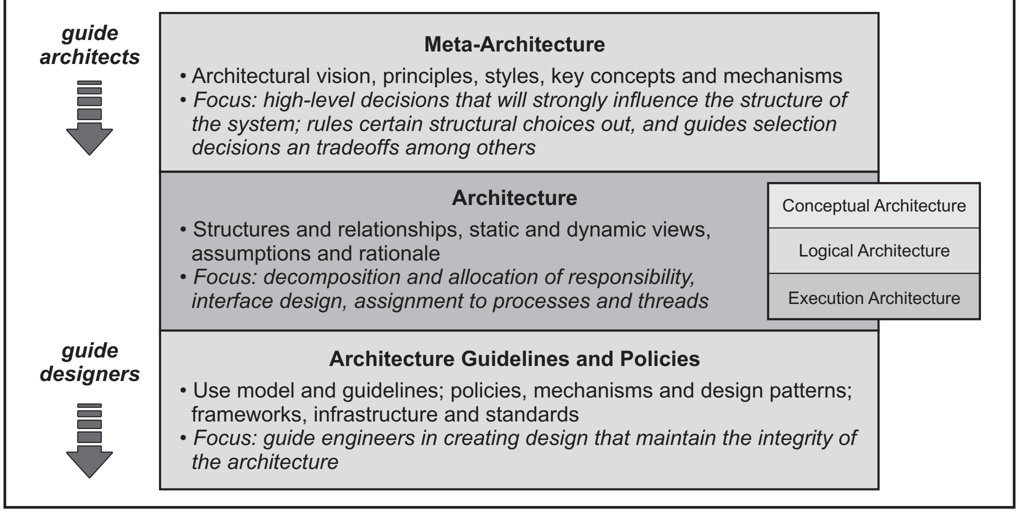
It provides a high-level description of the goals of the architecture, the use cases support by the system and architectural styles and mechanism that have been particular to most excellent achieve the use cases. This structure then allows for the development of the design criteria and documents that define the technical and domain standards in detail. It is these detailed design documents that will guide the development of the actual MedBiquitous content in terms of messages and services.

By analogy the architecture of a structure has to take into account the use of the building, what are the people living/working in it expecting and then has to define the size, shape, structure and so forth. The architecture has a set of guiding principles as well as known criteria and constraints that shape the proposed architecture. The designers then have to develop detailed specifications not only for the selection of materials but the placement of wiring, plumbing, lighting and so forth.

Architecture is at the centre of our layered decision model (Figure 4.9), and at the centre of the architecting activity. It is where the system structures are created, taking into account system priorities and constraints, and ensuring that the system will achieve the system objectives and architectural requirements. This work is informed and constrained by the decisions made in the Meta-Architecture.

Within the architecture layer, we use different views to enhance the understand ability of the architecture and to focus on particular concerns separately. We distinguish between Conceptual, Logical and Execution views, as shown in Figure 4.10.

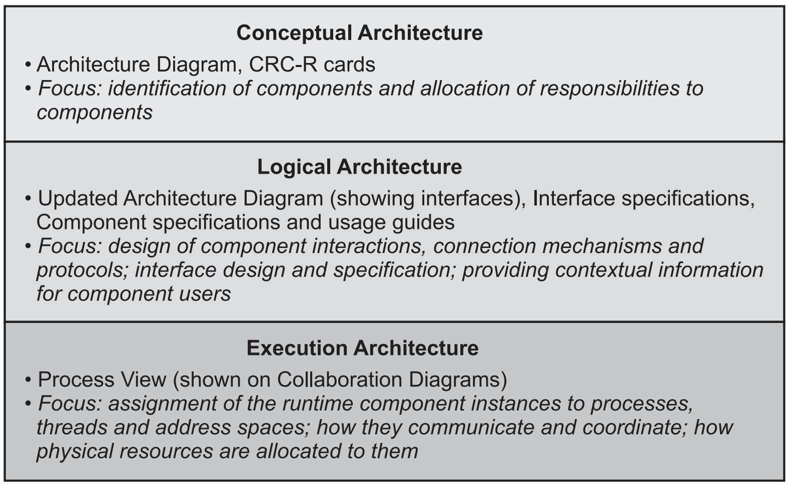
Figure 4.9: Software Architecture Decision Framework


guide architects
Meta-Architecture
• Architectural vision, principles, styles, key concepts and mechanisms
• Focus: high-level decisions that will strongly influence the structure of the system; rules certain structural choices out, and guides selection decisions an tradeoffs among others
Architecture
• Structures and relationships, static and dynamic views, assumptions and rationale
• Focus: decomposition and allocation of responsibility, interface design, assignment to processes and threads
Conceptual Architecture
Logical Architecture
Execution Architecture
guide designers
Architecture Guidelines and Policies
• Use model and guidelines; policies, mechanisms and design patterns; frameworks, infrastructure and standards
• Focus: guide engineers in creating design that maintain the integrity of the architecture

4.2.1 Meta-architecture

The meta-architecture is a set of high-level decisions that will powerfully power the integrity and structure of the system, but is not itself the arrangement of the system. The meta-architecture, through style, patterns of composition or interaction, principles, and philosophy, rules certain structural choices out, and guides selection decisions and trade-offs among others. By choosing message or co-ordination mechanisms that are repeatedly applied across the architecture, a consistent approach is ensured and this simplifies the architecture. It is also very useful at this stage, to find a metaphor or organizing concept that works for your system. It will help you think about the qualities that the system should have, it may even help you think about what components you need (in Conceptual Architecture), and it will certainly help you make the architecture more vivid and understandable.

Figure 4.10: Architecture Views



Conceptual Architecture

- Architecture Diagram, CRC-R cards
- *Focus: identification of components and allocation of responsibilities to components*

Logical Architecture

- Updated Architecture Diagram (showing interfaces), Interface specifications, Component specifications and usage guides
- *Focus: design of component interactions, connection mechanisms and protocols; interface design and specification; providing contextual information for component users*

Execution Architecture

- Process View (shown on Collaboration Diagrams)
- *Focus: assignment of the runtime component instances to processes, threads and address spaces; how they communicate and coordinate; how physical resources are allocated to them*

4.2.2 Conceptual Architecture

The Conceptual Architecture identifies the sophisticated mechanism of the system, and the relations among them. Its purpose is to direct concentration at a suitable decomposition of the system without delving into details. Moreover, it provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and users. It consists of the Architecture Diagram (without interface detail) and an informal component specification for each component.

4.2.3 Logical Architecture

In rational Architecture, the externally visible properties of the components are made precise and unambiguous through well-defined interfaces and component specifications, and key architectural mechanisms are detailed. The rational structural design provides a detailed “blueprint” from which component developers and component users can work in relative independence. It incorporates the detailed structural design Diagram (with interfaces), Component and Interface Specifications, and Component Collaboration Diagrams, along with discussion and explanations of mechanisms, rationale, etc.

4.2.4 Execution Architecture

An Execution Architecture is created for distributed or simultaneous systems. The process view shows the mapping of components onto the processes of the physical system, with attention being focused on such concerns as throughput and scalability. The deployment view shows the mapping of (physical) components in the executing system onto the nodes of the physical system.

Software architecture consists of:

1. ***Elements***: Processing elements (“functions”), connectors (“glue” procedure calls, messages, events, shared storage cells), data elements (what “flows” between the processing elements).

2. ***Form***: Properties (constraints on elements and system) and relationship (configuration, topology).
3. ***Rationale***: Philosophy and pragmatics of the system: requirements, economics, reliability, performance.

There can be "views" of the architecture from the perspective of the process elements, the data, or the connectors. The views might show static and dynamic structure.

Task Create a logical structure of software.

Self Assessment Questions

1. .... is an elementary process in which data crosses the boundary from outside to inside.
   (*a*) External inputs (*b*) External outputs
   (*c*) External inquiry (*d*) Internal logical files
2. A .... is a unique user recognizable, non recursive field on an ILF.
   (*a*) EIF (*b*) DET
   (*c*) EQ (*d*) VAF
3. The final function point count is obtained by multiplying the VAF times the ....
   (*a*) EIF (*b*) IFPUG
   (*c*) UAF (*d*) GSC
4. .... is the ease with which a program can be correct if an error occurs.
   (*a*) Correctness (*b*) Maintainability
   (*c*) Integrity (*d*) Usability
5. The conceptual architecture identifies the high-level components of the system, and the relationships among them.
   (*a*) True (*b*) False

4.3 Architecture Views

In our Software Architecture Decision Framework (Figures 4.2 and 4.3), we presented a set of standardized views. These are what we have found to be useful in guiding architects as they make architectural decisions that is, they provide useful thinking tools for considering decisions and choosing among alternatives. They also become the foundation for the architecture specification, by which we mean the complete set of architecture decisions at the chosen level(s) of abstraction, specificity and precision.

Both structural and behavioural views are important to thinking through and representing architecture:

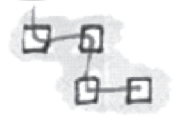
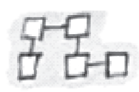
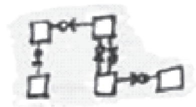
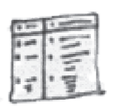
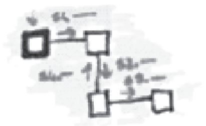
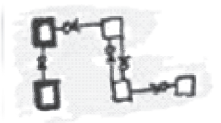
- ***Structural View***: If we accept that "architecture is the high-level structure of the system comprised of components, their interrelationships, and externally visible properties", the structural view is central. It consists of the Architecture Diagram (stereotyped UML Class Diagram), and Component and Interface Specifications.
- ***Behavioural View***: In decomposing the system into components and designing their interfaces, and in designing mechanisms to address key cross-cutting concerns, we have to answer the question "How does this work?" Likewise, in understanding and using the architecture, we have to be able to answer the same question. This is the role of the behavioural view, with its Component Collaboration or Sequence Diagrams (stereotyped UML Sequence and Collaboration Diagrams).

Structural and behavioural views are applicable for each of the Conceptual, Logical and Execution Architecture views (or layers), as shown in Figure 4.11.

In general, however, you would want to at least have:

- An Architecture Diagram and informal component descriptions for Conceptual Architecture,
- Interface Specifications, Component Specifications and an updated (Logical) Architecture Diagram showing interfaces and relationships for Logical Architecture. You should also include Collaborations Diagrams for key Use Case steps. Note that these may be used to describe system behaviour (they are simply illustrative), or to prescribe how system behaviour is to be accomplished, and you should be clear about which of these you intend. In general, you should avoid being prescriptive unless you have a strong architectural reason for curtailing the creative options open to designers and developers.

Figure 4.11: Architecture Views with Structure and Behaviour

	Behavioral View	**Structural View**
Conceptual Architecture (abstract)	 Collaboration trace	 Architecture Diagram Informal Component Specs (CRC-R)
Logical Architecture (detailed)	Collaboration Diagrams	 Architecture Diagram with I/Fs  Interface Space
Execution Architecture (Process View and Deployment View)	 Collaboration Diagrams showing processes	 Architecture Diagram showing Active Components

4.3.1 Architecture Documentation

With the architecture drivers and various architecture views, you have the raw material from which to compose documents and presentations targeted at different audiences ("viewpoints", in the parlance of IEEE1471). At a minimum, your document set should include:

- ***Reference Specification:*** The full set of architecture drivers, views, and supplements such as the architecture decision matrix and issues list, provides your reference specification.
- ***Management Overview:*** For management, you would want to create a high-level overview, including vision, business drivers, Architecture Diagram (Conceptual) and rationale linking business strategy to technical strategy.
- ***Component Documents:*** For each component owner, you would ideally want to provide a system level view (Logical Architecture Diagram), the Component Specification for the component and Interface Specifications for all of its provided interfaces, as well as the Collaboration Diagrams that feature the component in question.

Task Prepare an architectural document for software.

4.4 Architecture Styles

An architectural style, sometimes called an architectural pattern, is a set of values a coarse grained pattern that provides an abstract structure for a family of systems. An architectural style improves partitioning and promotes design use again by providing solutions to frequently recurring problems. You can believe of architecture styles and patterns as sets of principles that shape an application. Garlan and Shaw define an architectural style as:

Family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints say, having to do with execution semantics might also be part of the style definition.

An understanding of architectural styles provides several benefits. The most important benefit is that they provide a common language. They also provide opportunities for conversations that are technology agnostic. This facilitates a higher level of conversation that is inclusive of patterns and principles, without getting into specifics. For example, by using architecture styles, you can talk about client/server versus n-tier. Architectural styles can be organized by their key focus area. The following Table 4.3 lists the major areas of focus and the corresponding architectural styles.

Table 4.3: Major Areas of Focus and the Corresponding Architectural Styles

Category	Architecture styles
Communication	Service-Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain Driven Design
Structure	Component-Based, Object-Oriented, Layered Architecture

4.4.1 Client/Server Architectural Style

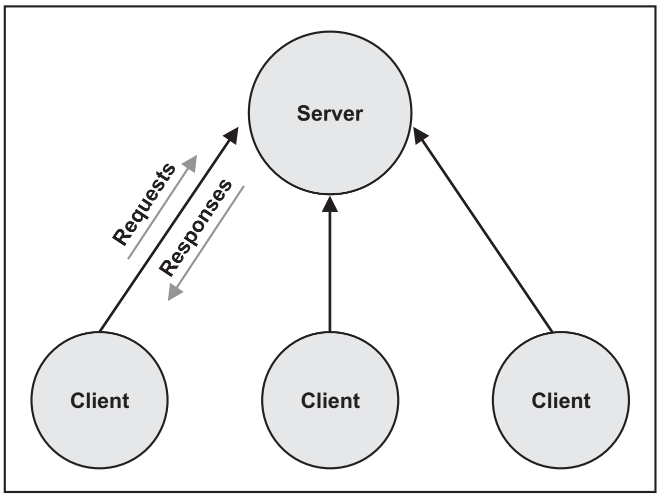
The client-server architectural (CSA) style is most common in enterprise systems, and is also applied for integrating edge and enterprise systems. A "service" a discretely scoped business or technical functionality, and is offered by a "service provider" or server to "service requestors" or clients. Operationally, a service is defined by messages exchanged between clients and servers. This architectural style can be viewed as a special case of the generic data oriented integration architecture, as shown in Figure 4.5.

The request and responses are correlated with a client specific "correlation_id" field in the data model. The QoS and are chosen to achieve the desired flow characteristics. For request and reply topics, QoS are chosen to deliver all the data samples in the order they happened. Content filtering is used on the response topic to receive only the responses intended for the requestor.

It is interesting and ironic to note that despite the hype, the vast majority of the implementations of SOA rely on tightly-coupled technologies and design principles, resulting in tightly-coupled SOA software. This runs counter to the promise of SOA, and can prove detrimental to its success

Notes

Figure 4.12: Client-server Model



in the long run. We have outlined an approach that results in "loosely-coupled SOA" software, and can indeed unleash the full potential of SOA.

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.

Historically, client/server architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures, or with a dedicated file server. More generally, however, the client/server architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client.

Today, some examples of the client/server architectural style include Web browser based programs running on the Internet or an intranet; Microsoft Windows operating system based applications that access networked data services; applications that access remote data stores (such as e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).

Other variations on the client/server style include:

- ***Client-Queue-Client Systems***: This approach allows clients to communicate with other clients through a server-based queue. Clients can read data from and send data to a server that acts simply as a queue to store the data. This allows clients to distribute and synchronize files and information. This is sometimes known as passive queue architecture.
- ***Peer-to-Peer (P2P) Applications***: Developed from the Client-Queue-Client style, the P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients. It extends the client/server style through multiple responses to requests, shared data, resource discovery, and resilience to removal of peers.
- ***Application Servers***: A specialized architectural style where the server hosts and executes applications and services that a thin client accesses through a browser or specialized client installed software. An example is a client executing an application that runs on the server through a framework such as Terminal Services.

The main benefits of the client/server architectural style are:

- ***Higher Security***: All data is stored on the server, which generally offers a greater control of security than client machines.
- ***Centralized Data Access***: Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- ***Ease of Maintenance***: Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Consider the client/server architectural style if your application is server based and will support many clients, you are creating Web-based applications exposed through a Web browser, you are implementing business processes that will be used by people throughout the organization, or you are creating services for other applications to consume. The client/server architectural style is also suitable, like many networked styles, when you want to centralize data storage, backup, and management functions, or when your application must support different client types and different devices.

However, the traditional 2-Tier client/server architectural style has numerous disadvantages, including the tendency for application data and business logic to be closely combined on the server, which can negatively impact system extensibility and scalability, and its dependence on a central server, which can negatively impact system reliability. To address these issues, the client-server architectural style has evolved into the more general 3-Tier (or N-Tier) architectural style, described, which overcomes some of the disadvantages inherent in the 2-Tier client-server architecture and provides additional benefits.

4.4.2 N-Tier/3-Tier Architectural Style

N-tier and 3-tier are architectural deployment styles that explain the departure of functionality into segments in much the same way as the covered style, but with each segment being a tier that can be located on a physically divide computer. They evolved through the component-oriented approach, generally using platform precise methods for communication instead of a message-based move towards.

N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization. Each tier is completely independent from all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request. Communication between tiers is typically asynchronous in order to support better scalability.

N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.

An example of the N-tier/3-tier architectural style is a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network. Another example is a typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.

The main benefits of the N-tier/3-tier architectural style are:

- ***Maintainability***: Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.

- *Scalability*: Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- *Flexibility*: Because each tier can be managed or scaled independently, flexibility is increased.
- *Availability*: Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Consider either the N-tier or the 3-tier architectural style if the processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing in other layers, or if the security requirements of the layers in the application differ. For example, the presentation layer should not store sensitive data, while this may be stored in the business and data layers. The N-tier or the 3-tier architectural style is also appropriate if you want to be able to share business logic between applications, and you have sufficient hardware to allocate the required number of servers to each tier.

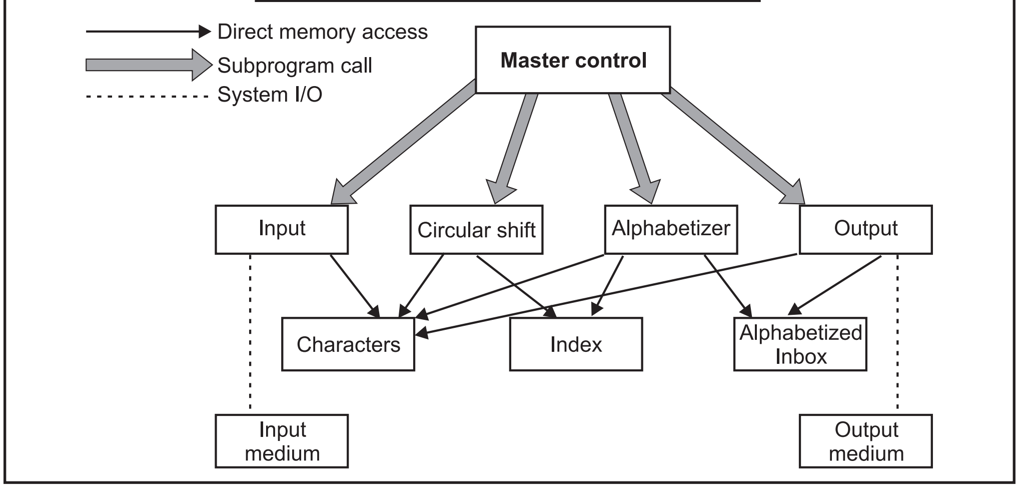
Consider using just three tiers if you are developing an intranet application where all servers are located within the private network; or an Internet application where security requirements do not restrict the deployment of business logic on the public facing Web or application server. Consider using more than three tiers if security requirements dictate that business logic cannot be deployed to the perimeter network, or the application makes heavy use of resources and you want to offload that functionality to another server

4.4.3 Shared Data Architecture

Data is shared between components through shared storage. The computational components are coordinated, with subroutines to a main program sequences through them. Data is then communicated between the components through shared storage. Communication between the computational components and shared data is an unconstrained read-write protocol.

Data is shared, common storage this can be plus and a minus (See Figure 4.13).

Figure 4.13: KWIC: Shared Data Solution



Advantages

- The system is space efficient.
- The system is time efficient.

Since computational aspects are in different modules the design is easier to understand.

Disadvantages

- Changes in data storage format affects all modules, for example if the data structure is changed, then all the modules have to be modified to adjust to that change.
- Changes in algorithm not well supported.
- Poor stability if access to the shard data is not handled appropriately.
- Not supportive of reuse due to the dependency on the shared data.
- Enhancements not easily incorporated because of the above reasons.

Case Study

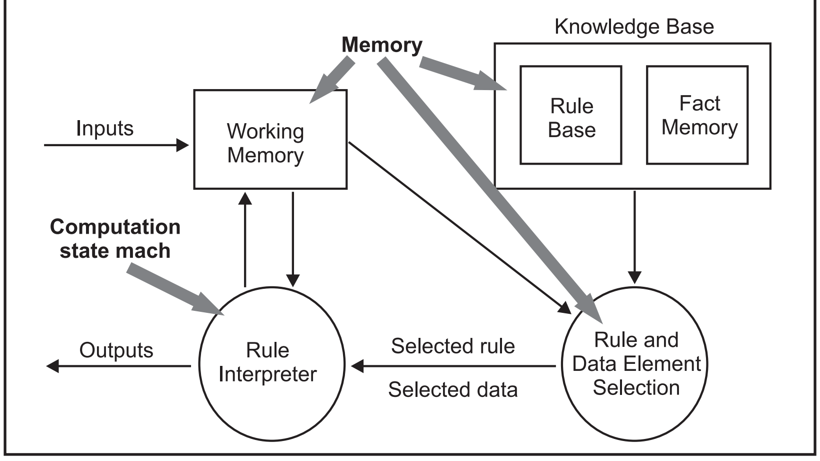
An Interpreter Using Different Idioms for the Components

Rule-based systems provide a means of codifying the problem-solving know how of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Since these rules are not directly executable by available computers, systems for interpreting such rules must be provided. Hayes-Roth surveyed the architecture and operation of rule-based systems.

The basic features of a rule-based system, shown in Hayes-Roth's rendering as in Figure 1, are essentially the features of a table-driven interpreter, as outlined earlier.

- The pseudo-code to be executed, in this case the knowledge base.
- The interpretation engine, in this case the rule interpreter, the heart of the inference engine.
- The control state of the interpretation engine, in this case the rule and data element selector.
- The current state of the program running on the virtual machine, in this case the working memory.

Figure 1: Basic Rule-Based System



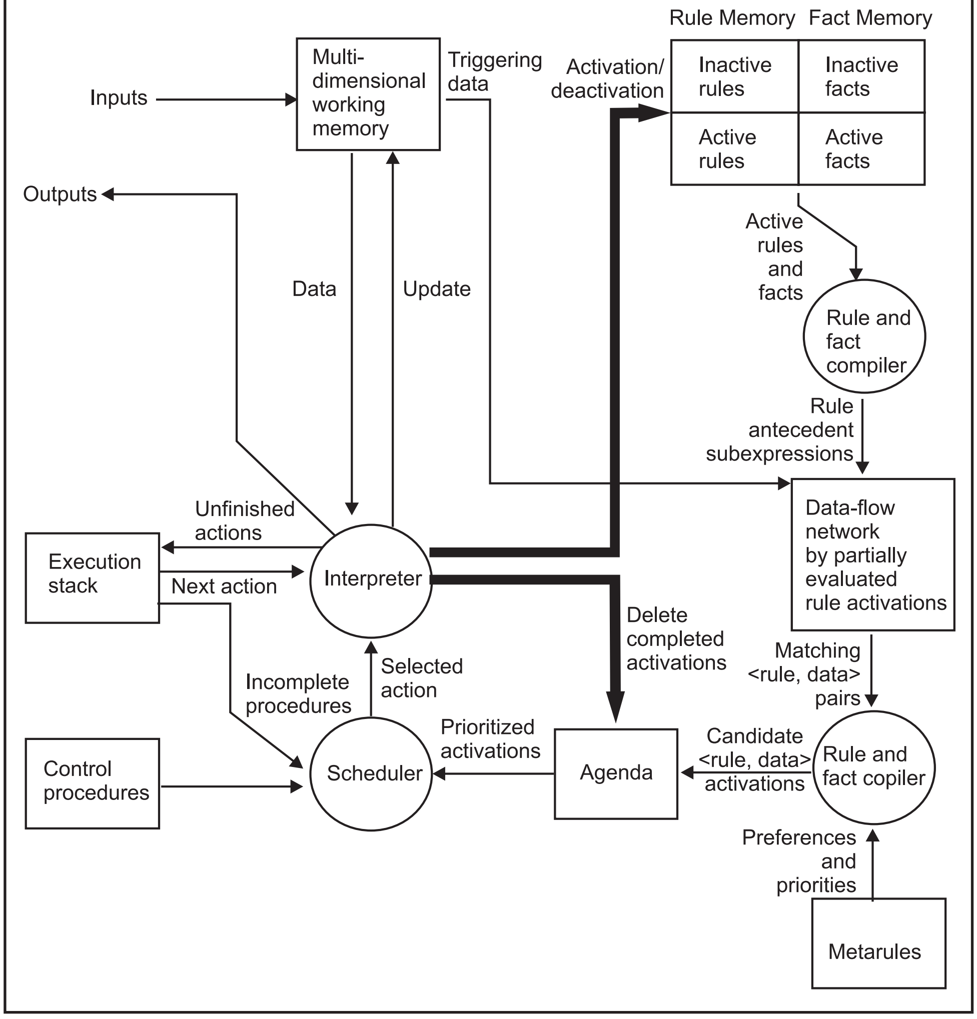
Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design leads to the more complicated view shown in Figure 2. In adding this complexity, the original simple interpreter vanishes in a sea of new interactions and data flows. Although the interfaces among the original modules remain, they are not distinguished from the newly-added interfaces.

Contd...

However, the interpreter model can be rediscovered by identifying the components of Figure 2 with their design antecedents in Figure 2. This is done in Figure 3. Viewed in this way, the elaboration of the design becomes much easier to explain and understand. For example, we see that:

- The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.
- The rule interpreter is expanded with the interpreter idiom (that is, the interpretation engine of the rule-based system is itself implemented as a table-driven interpreter), with control procedures playing the role of the pseudo-code to be executed and the execution stack the role of the current program state.
- "Rule and data element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations; in this pipeline the third filter ("nominators") also uses a fixed database of Meta rules.
- Working memory is not further elaborated.

Figure 2: Sophisticated Rule-based System

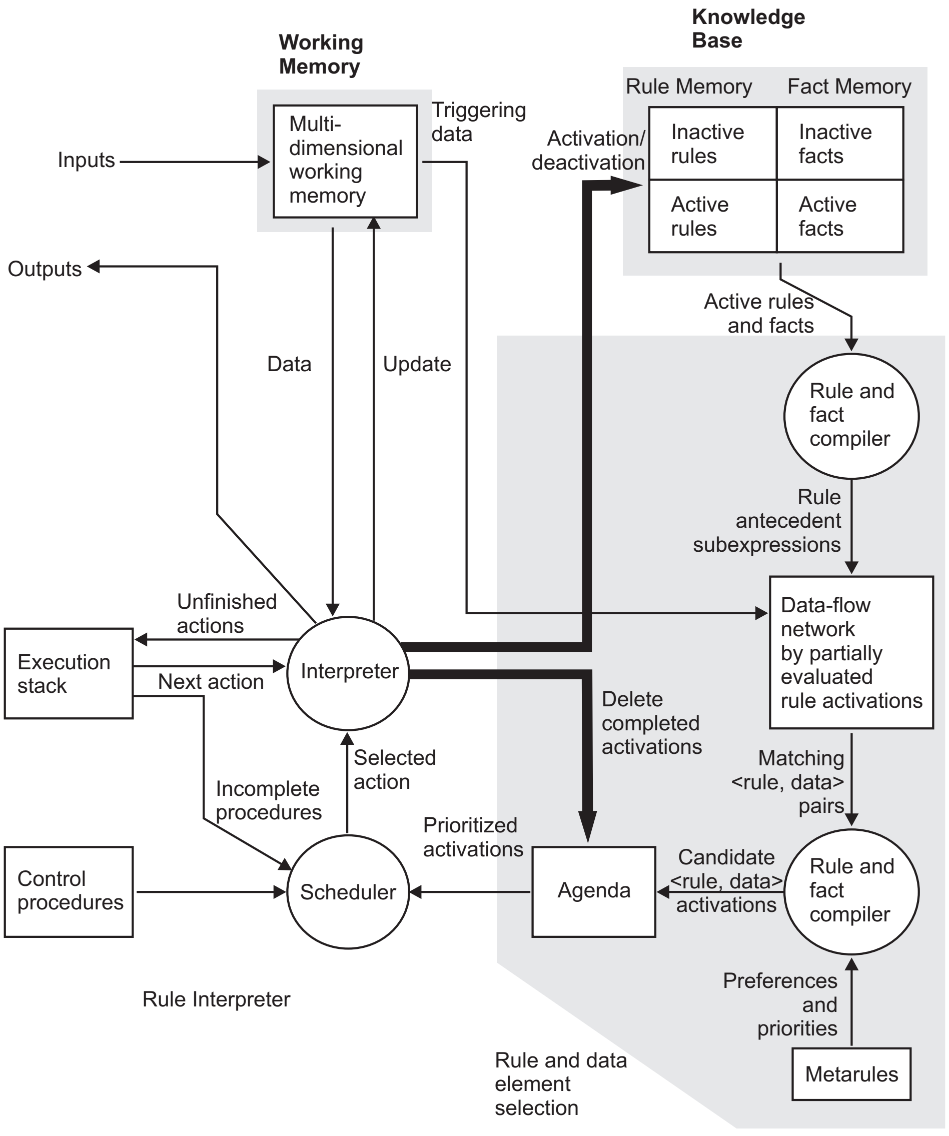


The interfaces among the rediscovered components are unchanged from the simple model except for the two bold lines over which the interpreter controls activations.

Contd...

This example illustrates two points. First, in a sophisticated rule-based system the elements of the simple rule-based system are elaborated in response to execution characteristics of the particular class of languages being interpreted. If the design is presented in this way, the original concept is retained to guide understanding and later maintenance. Second, as the design is elaborated, different components of the simple model can be elaborated with different idioms.

Figure 3: Simplified Sophisticated Rule-based System



Note that the rule-based model is itself a design structure: it calls for a set of rules whose control relations are determined during execution by the state of the computation. A rule-based system provides a virtual machine a rule executor to support this model.

Questions

1. What is sophisticated rule-based system?

2. Explain the simplified sophisticated rule-based system.

Self Assessment Questions

6. The software engineering discipline has become competent in the area of……………..
   (*a*) measurement (*b*) testing
   (*c*) validation (*d*) verification
7. A quantitative measure of the degree to which a system, component or process possesses a given ……………….
   (*a*) attribute (*b*) program
   (*c*) system (*d*) None of these.
8. …………………. is the mapping of numbers to attributes of objects in accordance with some prescribed rule.
   (*a*) Empirical (*b*) Metrics
   (*c*) Measurement (*d*) None of these.
9. Human beings solve problems by breaking them into smaller understandable …………..
   (*a*) functions (*b*) classes
   (*c*) pieces (*d*) methods
10. External Inputs (EI) is an elementary process in which data crosses the boundary from ……………….. to inside.
    (*a*) internal logical files (*b*) internal files
    (*c*) inside (*d*) outside
11. External Inquiry (EQ) is an elementary process with both input and output components that result in data retrieval from one or more …………..
    (*a*) internal logical files (*b*) external logical files
    (*c*) logics (*d*) functions
12. Data is shared between components through …………...
    (*a*) system memory (*b*) shared storage
    (*c*) local network (*d*) None of these.
13. N-tier application architecture is characterized by the functional decomposition of …………………..
    (*a*) applications (*b*) programs
    (*c*) logical files (*d*) networks
14. The client/server architectural style describes ………….. systems that involve a separate client and server system.
    (*a*) network (*b*) programming
    (*c*) main (*d*) distributed
15. A qualitative metric still provides relevant indication of quality.
    (*a*) True (*b*) False

4.5 Summary

- Function Points used to size software applications accurately. Sizing is an important component in determining productivity (outputs/inputs).
- They can be counted by different people, at different times, to obtain the same measure within a reasonable margin of error.
- Function Points are easily understood by the non technical user. This helps communicate sizing information to a user or customer.
- Function Points can be used to determine whether a tool, a language, an environment, is more productive when compared with others.
- Client/Server Architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures.
- Client-Server Architectural (CSA) style is most common in enterprise systems, and is also applied for integrating edge and enterprise systems.
- A "service" a discretely scoped business or technical functionality, and is offered by a "service provider" or server to "service requestors" or clients.

4.6 Keywords

3-Tier: This is architectural deployment styles that describe the separation of functionality into segments in much the same way as the layered style.

Application Servers: A specialized architectural style where the server hosts and executes applications and services that a thin client accesses through a browser or specialized client installed software.

Client-Queue-Client Systems: This approach allows clients to communicate with other clients through a server-based queue.

Measurement: In the most general sense, is the mapping of numbers to attributes of objects in accordance with some prescribed rule.

N-Tier Architectures: It has at least three separate logical parts, each located on a separate physical server.

Software Metrics: These are therefore quantitative measurements of product (system or component), process, or even project (in this case software projects) indicating the quality of a desired attribute.

Software Validation: It has always been problematic in software engineering unlike many other engineered products; software often cannot be visualized, thus, in many cases, resulting in software validation being a reactive last minute process.

Traditional 2-Tier Client/Server: This is architectural style has numerous disadvantages, including the tendency for application data and business logic to be closely combined on the server.

Validation: It is "the steps and the process needed to ensure that the system configuration, as designed, meets all requirements initially specified by the customer."

Verification: It is defined as "The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase."

Lab Exercise

1. Prepare a flow chart to perform the check validation in software.
2. Create a structure to perform the verification.

4.7 Review Questions

1. Define the N-Tier/3-Tier Architectural Style.
2. What is the difference between shared data architecture and client/server architectural style?
3. Explain the architecture styles.
4. Define the architecture documentation.
5. Define the architecture views.
6. What is meta-architecture?
7. Explain the software architecture and software quality metrics, in brief.
8. Define the five major components of software architecture.
9. Explain the function point analysis.
10. Define the metrics and validation.

Answers of Self Assessment Questions

1.	(*a*)	2.	(*b*)	3.	(*c*)	4.	(*b*)	5 .	(*a*)
6.	(*d*)	7.	(*a*)	8.	(*c*)	9.	(*c*)	10.	(*d*)
11.	(*a*)	12.	(*b*)	13.	(*a*)	14.	(*d*)	15.	(*a*)

4.8 Further Readings

Books

"*Metrics and Models in Software Quality Engineering, Second Edition*", by Stephen H. Kan.

"*Software Requirements and Specifications*" By - Michael Jackson, Addison-Wesley

Online link

http://www.bredemeyer.com/ArchitectingProcess/ArchitecturalRequirements.htm

Notes

Unit 5: Software Project Planning

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define the process planning
- Describe the effort estimation
- Discuss the COCOMO model
- Discuss the COCOMO II
- Describe the development effort estimates
- Explain the project scheduling and staffing
- Discuss the software configuration management
- Define the quality plan
- Explain the risk management
- Define the project monitoring

Introduction

The purpose of the software project planning is to describe the processes, procedures, and guidelines that should be followed by a software project manager in order to plan and execute a productive and successful project. It is assumed that the reader is familiar with the SSC San Diego organizational policy for Software Project Planning (SPP).

Software project planning is a Level 2 Capability Maturity Model (CMM) Key Process Area (KPA). Satisfying this KPA is a major step toward achieving Level 2 (Repeatable). This KPA requires a written process for planning a software project. It also requires the development of a project Software Development Plan (SDP), which is the document which describes the plan for the software project. The planning phase is one of the most crucial steps in any software development project. The success of a software project is often determined in the planning phase. Lack of adequate planning often results in a project's failure to meet either cost, schedule, or performance objectives or all three. The quality of your project plan will probably reflect the quality of your project. Be thorough, concise and precise. It is never too early to plan.

This process applies to any software project at SSC San Diego and also to any activity of the software development life cycle. Though this process is written from the viewpoint of a project in the requirements activity, it can be easily modified for use by projects in any activity of the software life cycle. Even software projects that are in life cycle maintenance can implement this process for planning software builds or for implementing Engineering Change Proposals (ECPs).

Purpose: The objective of the process activity. If a sub process activity exists, the details are described in that specific paragraph description.

Role and Responsibility: The responsibilities of individuals or groups for accomplishing a process activity.

Entry Criteria: The elements and conditions necessary to be in place to begin a process activity. Reading lower level activities assumes that the entry criteria for all higher level activities have been satisfied.

Input: Data or material with which a process activity is performed.

Process Activity: Actions to transform an input, as influenced by controls, into a predetermined output.

Notes

Output: Data or material produced by or resulting from a process activity. It must include the input data in some form. The output title differs from the input title in order to indicate that an activity has been performed.

Exit Criteria: Elements and/or conditions necessary to be in place to complete a process activity.

Process Metrics: Data collected which can be analyzed and used to improve the process.

Process Overview

Planning Initiation

The SSC San Diego Software Project Planning (SPP) process begins with the planning initiation step. In this step a software project manager is selected and resources and budget are allocated to the planning and re-planning activities. The Requirements Management (RM) Guidebook is an interfacing process to the SPP process. Requirements are the major driving force in SPP and therefore are a major interface to this process. Without requirements, you would not know what to plan or what to estimate. Initial software estimates and software activities are also developed in this step. The SSC San Diego Software Size, Cost, and Schedule Estimation process is an interfacing process.

Develop SDP

The next step in the process is to develop the Software Development Plan (SDP). The MIL-STD-498 Data Item Description (DID) for the SDP, DI-IPSC-81427, should be used as a format for the SDP. The SSC San Diego SDP template, which is based on the MIL-STD-498 DID, can be used as a guide for developing the project specific SDP. The SDP should include items such as software estimates, schedules, milestones, Work Breakdown Structure (WBS), software development environment, software development methodology, software test methodology, and software risks. The SDP is used to establish commitments on the project. Software estimates are refined using the SSC San Diego Software Size, Cost, and Schedule Estimation process.

Review and Approve SDP

After the SDP has been developed it should undergo formal review and approval. The SSC San Diego Formal Inspection process is one type of review methodology that can be used. Review of the SDP should include all groups internal and external to the organization which will be affected by the work in the SDP. Affected groups, both internal and external to the organization, should also approve the SDP by signing the signature page of the SDP indicating their commitment and acceptance of the SDP.

After the SDP has been approved it should be placed under configuration management. The SSC San Diego Software Configuration Management (SCM) process can be used as a guide to develop the project's SCM process.

Implement SDP Processes and Apply SPTO Process

The project is now ready to implement the activities as described in the SDP. In implementing the SDP, follow the project's tailored version of the SSC San Diego Software Project Tracking and Oversight (SPTO) process to provide information about the software project which can precipitate changes to the SDP. These changes should be implemented in accordance with the project's SCM process. The project should also implement the project's tailored SQA process in order to monitor the project's software development activities. In addition, the SQA group should be monitoring the activities of the SPP process.

Process Measurement and Improvement

Metrics are collected on each process step and are then used to develop process improvements in the SPP process. These measurements are analyzed against both planned and historical data, if available.

Revise SDP

Proposed changes to the SDP are analyzed and approved changes are implemented. After changes are made, the SDP should follow the project's standard review and approval process.

The biggest single problem that afflicts software developing is that of underestimating resources required for a project. Developing a realistic project plan is essential to gain an understanding of the resources required, and how these should be applied.

Types of plan:

- *Software Development Plan*: The central plan, which describes how the system will be developed.
- *Quality Assurance Plan*: Specifies the quality procedures & standards to be used.
- *Validation Plan*: Defines how a client will validate the system that has been developed.
- *Configuration Management Plan*: Defines how the system will be configured and installed.
- *Maintenance Plan*: Defines how the system will be maintained.
- *Staff Development Plan*: Describes how the skills of the participants will be developed.

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimate should attempt to define "best case" and "worst case" scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

Did u know? The "software crisis" of the 1960s and 1970s was so called because of a string of high profile software project failures over budget, overdue, etc.

5.1 Process Planning

The purpose of Software Project Planning is to set up reasonable strategy for performing the software engineering and for supervision the software project.

Software Project Planning involves developing estimates for the work to be performed, establishing the necessary commitments, and defining the plan to perform the work.

The software planning begins with a statement of the work to be performed and other constraints and goals that define and bound the software project (those established by the practices of the Requirements Management key process area). The software planning process includes steps to estimate the size of the software work products and the resources needed, produce a schedule, identify and assess software risks, and negotiate commitments. Iterating through these steps may be necessary to establish the plan for the software project (i.e., the software development plan).

This plan provides the basis for performing and managing the software project's activities and addresses the commitments to the software project's customer according to the resources, constraints, and capabilities of the software project.

"Planning – a clean sheet of paper and anything is possible"

In this phase we will concentrate on the actual planning of a project and include activities such as:

- Setting up milestones

- Identify activities
- Allocate resources
- Create a time-schedule
- Reviewing the budget
- Do a risk analysis
- Set up a meeting schedule
- Project administration

5.1.1 Setting up Milestones

It is a good start to create an overall picture of how the project will be performed before planning the particulars. In the past there used to be milestones along the roads to tell the distance to next village. This is also very useful in project planning.

A milestone in a project could be an approved deliverable, an important decision or a market verification that the project is on the right track. By identifying the milestones in a project and also identifying their order of appearance, an overall picture of the project could be delivered.

A milestone in a project marks something that has been done or accomplished, and gives guidance about where you are in the project.

A milestone should have the following criteria's:

- Everyone should experience them as natural.
- Mark an important control or decision in the project.
- They should be measurable and be able to check.
- There should not be too many of them.
- Appear within reasonable distance.

By this time you can now detect certain natural steps in the project plan. It is easier to review and follow up a project if you take one step at the time. It is suggested that you have shorter steps in the beginning of the project when the uncertainty is at worst.

Regarding the method for project planning it does not follow a structured step-by-step work plan; the team see the project planning as a process which should be easy to use and easy to understand. Because smaller projects need to have a minimum of project planning administration, and software projects in particular need to follow a plan that is not thoroughly detailed, we find their method being proper to investigate.

The method give you space to follow it in detail or just picks out the techniques and processes that is proper for your project.

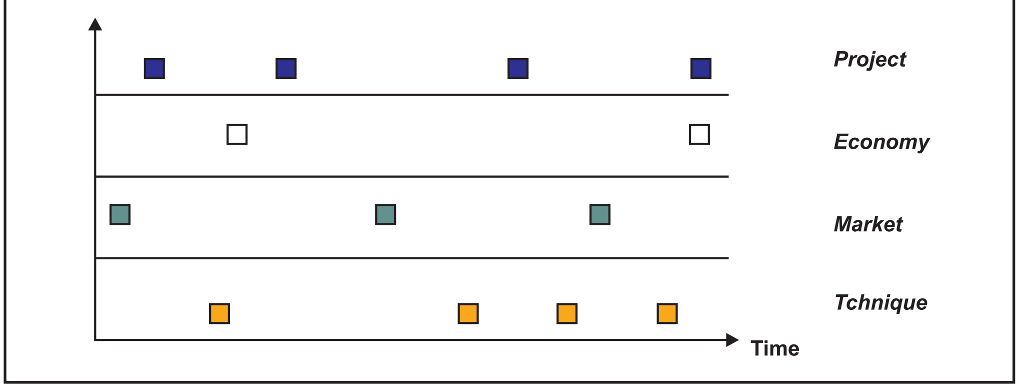
How to make a milestone-chart:

1. Gather project members and stakeholders in a room. (This could be done at the kick-off meeting if the project is relatively small). You will need a large whiteboard.
2. Draw a horizontal arrow at the lower part of the whiteboard, symbolizing time.
3. Divide the diagram into fields, each representing areas like technique, market, project and economy within the project.
4. Turn the deliverables into milestones and write those down on post-it notes. (e.g. report ready, prototype ready).

5. Place the milestones into the model on the whiteboard from the end to the beginning.
6. Think through the project from the start and if necessary create more milestones.
7. Adjust the milestones so that they are right in time.
8. Choose which milestones will be used in the project and document these. Place first milestones at the top of the board and the last ones at the bottom.

Prioritise generality before detail. You will plan the details later if necessary. This process is best if everyone stands in front of the whiteboard and work together doing the milestone-chart.

Figure 5.1: An Example of a Milestone-Chart



The milestones delivered from the group work are either documented in a chart, like the example above (See Figure 5.1), or in a form.

It could be enough for smaller projects to use a more detailed Milestone-chart as the project plan. They also suggest that the Milestone-chart only needs to be complemented with a detailed plan containing activities for the next step in the project. This detailed plan should have assigned resources added to it and other necessary needs of the project.

5.1.2 Activities

Burenius and Lindstedt define activities as the machines of the project.

In the activities:

- All deliverables are produced
- All resources are used
- All follow-up is done (results and pitfalls)

An activity should complete deliverables as final results, be optimal use of resources and easy to follow-up in progress reports.

An activity can be formulated by verbs and nouns e.g. build prototype and write report. In building your list of activities it should, according to the Software Project Management, include not only the effort based, e.g. perform system testing, kind of activities but schedule dependent, e.g. attend weekly meetings, and synchronisation activities, e.g. formal technical reviews, as well.

When producing your activity list, make sure that you make a list that is easily understood and is in a useful form. First break up your list into phases and have a separate list for each phase.

Your list should contain information like name and description of the activity, how long it will be in duration. Another good thing to do is to specify which resources should be allocated to the activity, and if there are activities that depend on each other.

When estimating total amount of time for a milestone or an activity, Burenius and Lindsted suggests a simple formula for calculating:

Pocus formula:

$$\text{Estimated time: } = \boxed{\frac{\text{Pessimistic} + 4 * \text{most likely} + \text{optimistic}}{6}}$$

5.1.3 Resources

When you have identified certain natural steps in a plan and set up activities it is time to allocate resources to each activity. Examples of resources in a project can be:

- Organizations
- Individuals
- Equipment
- Furniture
- Facilities
- Test laboratories
- External contractors
- Utilities

There are many factors to consider when choosing your resources for a software project. These factors can be e.g. computer memory capacity, the computer processor use and the capacity of the communication channel.

Since the estimation of the resources not always is so easy you might want to use a helping tool. Burton and Michael chooses to use an art gallery scenario for this task figure 5.2. This chart is a very simple tool and therefore applicable on the small-scale project.

Figure 5.2: An Art Gallery Scenario

Chunk description	Skills	Materials	Equipment	Time	Cost
Project set up and approved				20 days	

This scenario will help you to determine what skills, materials, equipment, time and costs you will need to achieve your objectives that are stated to the left of the chart. You should beware of looking at the items in isolation, which will fail you from getting the full picture. Every item reflects on another.

Burenius and Lindstedt suggest another way of estimating the resources. They say that for the small-scale project the easiest way to estimate the resources within a project is to discuss which resources will be between milestones in a step. You begin from the start with digging up all the work that is needed to be able to get to the next milestone. You continue this until the first step of the plan is done. Then you summarise the resources needed and the final result should state the total amount of hours for the whole step. This amount divided with available capacity will be the length of the step. Be observant though if people working in the project only work part-time. A certain slack is also necessary to plan.

The authors also give us some guidelines when estimating the resources:

- Avoid individual estimations. Invite others to estimate preferably both optimists and pessimists.

Notes

- Do not estimate a general value. Take in consideration both pessimistic and optimistic views.
- Do not do a single estimation in the beginning of the project. Do as a habit, several estimations at the end of every step in the pan.
- Estimate both revenue and costs.
- Look for references, is the calculated result reasonable?

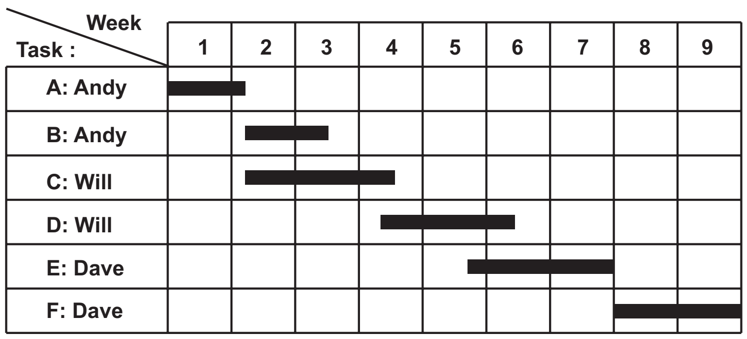
5.1.4 Setting up a Time-Schedule

"We work not only to manufacture but to give value to time".

When you have finished the milestone-chart and identified activities and possessions, and stakeholders and management have approved it, you can make a time-schedule. Another name for the time-schedule is Gantt chart or Bar chart. The schedule maps out, with help of a tool or technique, when things are going to happen and identifies the key milestone dates. There are many tasks that can be done parallel, this is a good way to speed up the work and get things done faster. But some tasks cannot start until others have been completed. It is important that the team members are clear about these linkages. If you fail to spot them it will cause many problems later.

In the case of small projects it is found that the sequencing-scheduling approach suitable. This is particularly so when we wish to allocate individuals to particular tasks at an early planning stage. However, on larger projects they do not recommend it. To get a better picture of the approach, we have drawn up an example of the bar chart in Figure 5.3.

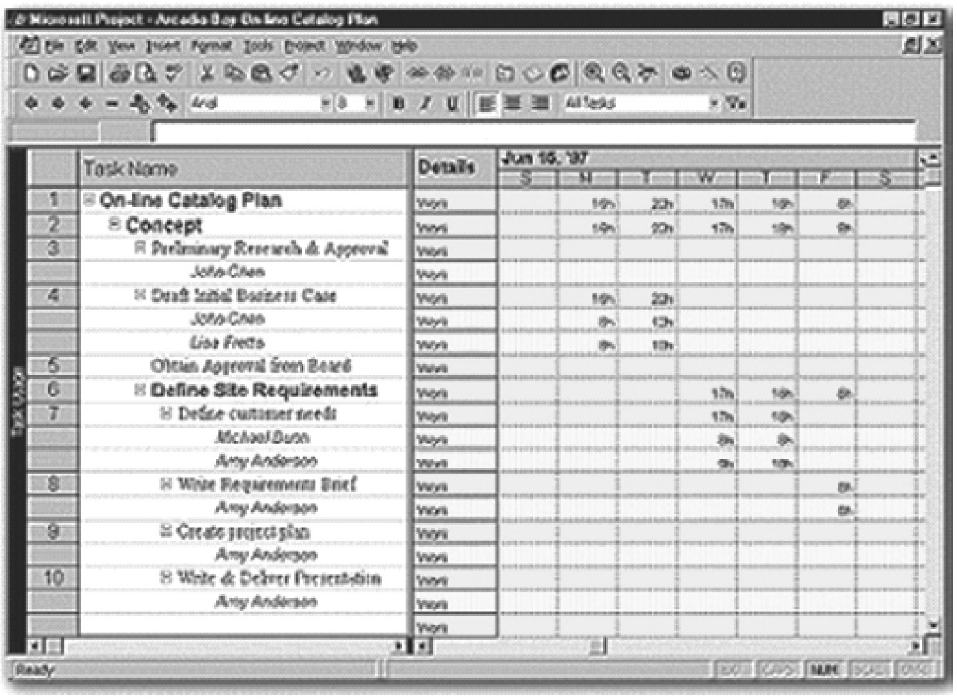
Figure 5.3: A Project Plan as a Bar Chart



One of the simplest and oldest techniques for tracking project development is the Gantt chart. This is a matrix diagram and an activity bar chart indicating scheduled activity dates and duration. Reported progress is recorded on the chart and a "today cursor" provides an immediate visual indication of which activities are ahead or behind schedule. The Gantt chart is named after Henri Gantt who created this planning tool. An example of a Gantt chart developed by Microsoft is given in 5.4.

Microsoft Project is a well-established software planning tool in office environments today. The tool can be very useful in planning projects, and the user can choose which detail to put into the planning chart. It can be used for both making a simple milestone chart and a complex time schedule having both resources and time estimation implemented in the schedule. The user needs training of course, but the amount of training is proportional to the level of detail the user wants to have. Before you use a software tool like MS Project it is essential to discuss and decide how the tool will be used in your specific project. As we mentioned before, a small project might not need a software tool at all.

Figure 5.4: An Example of a Gantt Chart


Microsoft Project - Arcadia Bay On-line Catalog Plan
File Edit View Insert Format Tools Project Window Help
Arial
8
All Tasks
Task Usage
Ready
EXT CAPS NUM SCRL OVR

	Task Name	Details	Jun 15, '97 S	M	T	W	T	F	S
1	On-line Catalog Plan	Work		16h	20h	17h	16h	8h	
2	Concept	Work		16h	20h	17h	16h	8h	
3	Preliminary Research & Approval	Work							
	John Chen	Work							
4	Draft Initial Business Case	Work		16h	20h				
	John Chen	Work		8h	12h				
	Lisa Fretts	Work		8h	10h				
5	Obtain Approval from Board	Work							
6	Define Site Requirements	Work				17h	16h	8h	
7	Define customer needs	Work				17h	16h		
	Michael Dunn	Work				8h	8h		
	Amy Anderson	Work				9h	10h		
8	Write Requirements Brief	Work						8h	
	Amy Anderson	Work						8h	
9	Create project plan	Work							
	Amy Anderson	Work							
10	Write & Deliver Presentation	Work							
	Amy Anderson	Work							
		Work							

At the end of the day the software schedule or Gantt chart is used to allow team members to easily see what they are doing and at what time.

An important part of the planning is the critical path. It means for identifying the critical path you need to find the difference between the earliest date and the latest date for an event known as the slack. This is a measure of how late an event may be without affecting the end date of the project. Any event with a slack of zero is critical in the sense that any delay in achieving that event will delay the completion date of the project as a whole. There will always be at least one path through the plan joining those critical events this path is known as the Critical Path. The significance of the critical path is two-fold:

- In managing the project we must pay particular attention to monitoring activities on the critical path so that the effects of any delay or resource unavailability are detected and corrected at the earliest opportunity.
- In planning the project it is the critical path that we must shorten if we are to reduce the overall duration of the project.

Task Create a Gantt chart for the software project planning strategy.

5.2 Effort Estimation

A software project is urbanized through software processes in which common activities comprise requirement, design and functioning, validation and evolution. Several of these processes have a future-oriented character in the sense that they are concerned with planning activities and products that have yet to be realised. A critical challenge is to create unsurprising software processes so that the software project can be completed on time and in a gainful manner.

Moreover, the work related to software development is typically multi-specialist in character, in the sense that it depends on dissimilar kinds of expert competencies being combined and aligned in different phases. Thus, software development may be described as a knowledge-intensive and combined practice that unfolds over time by way of complex processes of exploration, negotiation, and decision-making. To monitor and control the software processes, planning is significant. The estimation of work effort is a core task in this regard as it is used for purposes such as budgeting, trade-off and risk analysis, project planning and control, and software improvement investments analysis.

Software effort estimation, however, is a huge challenge, particularly as it is an area in which miscalculations can result in delays, low quality software, or the loss of contracts. A review of studies of software development projects shows that 70 to 80% of such projects overrun their estimates and spend on average 30 to 40% more effort than estimated. Today, as in the early days of computer programming, software projects whose plans and budgets are based on effort estimates that are too low experience severe management and delivery problems. Thus, more knowledge about the estimation practice of software professionals and the challenges faced in such work is important for project management.

Most of estimation has been concerned with developing different kinds of formal estimation models. However, judgment-based estimation is the most frequently applied estimation approach in the software industry today. The work a team of software professionals does when estimating the effort of a software project using a judgment-based approach. An effort estimate is here understood as the most likely number of work hours necessary to complete a software development project as assessed by the managers and developers responsible for delivery. In this approach, the quantification step (i.e., "the step where an understanding of the software development estimation problem is translated into a quantitative measure of the required effort") is based on a judgmental process as opposed to an algorithmic calculation. How this judgmental process comes about and how it is collaboratively achieved has, however, proved difficult to reveal. The aims to contribute to filling this gap by examining the work conducted to arrive at an effort estimate as a social and communicative practice. To disclose the details of this Software Effort Estimation as Collective Accomplishment practice we focus on the social interactions through which the estimation tasks are collectively explored, negotiated, and accomplished.

A requirement specification describing the details of the project to be developed forms a point of departure for the estimation process. Usually, this document outlines what the software system should do in detail, including the services and functions the system should provide and the constraints under which the system must operate. Here the constitutive elements of the interactional process as the team's way of approaching the information provided in the requirement specification and the way of using this as a basis for collective exploration and negotiation. We commence, however, by positioning our analysis within related strands of research.

5.3 COCOMO Model

The model of COCOMO-I is also called COCOMO'81 is presented. The underlying software lifecycle is a waterfall lifecycle. Detailed information about the ratings as well as the cost drivers can be found in Boehm proposed three levels of the model: basic, intermediate, detailed.

- The ***basic*** COCOMO'81 model is a single-valued, static model that computes software development effort (and cost) as a function of program size expressed in estimated thousand delivered source instructions (KDSI).
- The ***intermediate*** COCOMO'81 model computes software development effort as a function of program size and a set of fifteen "cost drivers" that include subjective assessments of product, hardware, personnel, and project attributes.

- The ***advanced*** or detailed COCOMO'81 model incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

The COCOMO'81 models depends on the two main equations:

1. *Development effort*: MM = a * $KDSI^b$ based on MM - man-month / person month / staff-month is one month of effort by one person. In COCOMO'81, there are 152 hours per Person month. According to organization this values may differ from the standard by 10% to 20%.
2. *Effort and development time (TDEV)*: TDEV = 2.5 * MM^c The coefficients a, b and c depend on the mode of the development. There are three modes of development:

Table 5.1: Development Modes

Development Mode	Project Characteristics			
	Size	Innovation	Deadline/ constraints	Dev. Environment
Organic	Small	Little	Not tight	Stable
Semi-datached	Medium	Medium	Medium	Medium
Embedded	Large	Greater	Tight	Complex hardware/ customer interfaces

The basic COCOMO (COnstructive Cost Model) applies the parameterised equation without much detailed consideration of project characteristics (See Figure 5.5).

Figure 5.5: Basic COCOMO

*MM = a * $KDSI^b$*

*TDEV = 2.5 * MM^c*

Basic COCOMO	a	b	c
Organic	2.4	1.05	0.38
Semi-deteched	3.0	1.12	0.35
Embeded	3.6	1.20	0.32

5.3.1 Intermediate COCOMO

The same basic equation for the model is used, but fifteen cost drivers are rated on a scale of 'very low' to 'very high' to calculate the specific effort multiplier and each of them returns an adjustment factor which multiplied yields in the total EAF (Effort Adjustment Factor). The adjustment factor is 1 for a cost driver that is judged as normal.

In addition to the EAF, the model parameter "a" is slightly different in Intermediate COCOMO from the basic model. The parameter "b" remains the same in both models (See Figure 5.6).

Figure 5.6: Intermediate COCOMO

*MM = a * $KDSI^b$*

*TDEV = 2.5 * MM^c*

Intermediate COCOMO	a	b	c
Organic	3.2	1.05	0.38
Semi-deteched	3.0	1.12	0.35
Embeded	2.8	1.20	0.32

Man Month correction is now:

$$\boxed{MM_{korr} = EAF*MM_{nominal}}$$

Did u know? Only the Intermediate form has been implemented by USC in a calibrated software tool.

5.3.2 Advanced, Detailed COCOMO

The Advanced COCOMO model computes attempt as a purpose of program size and a set of cost drivers weighted according to each phase of the software lifecycle. The Advanced model applies the in-between model at the component level, and then a phase-based approach is used to combine the estimate.

The four phases used in the detailed COCOMO model are: requirements planning and product design (RPD), detailed design (DD), code and unit test (CUT), and integration and test (IT). Each cost driver is broken down by phases as in the example shown in Table 5.2.

Table 5.2: Analyst Capability Effort Multiplier for Detailed COCOMO

Cost Driver	Rating	RPD	DD	CUT	IT
ACAP	Very Low	1.80	1.35	1.35	1.50
	Low	0.85	0.85	0.85	1.20
	Nominal	1.00	1.00	1.00	1.00
	High	0.75	0.90	0.90	0.85
	Very High	0.55	0.75	0.75	0.70

Estimates for each module are combined into subsystems and eventually an overall project estimate. Using the detailed cost drivers, an estimate is determined for each phase of the lifecycle.

The COCOMO has been continued to evolve and improve since its introduction. Beneath the "ADA COCOMO" model:

The model named "ADA_87" assumes that the ADA programming language is being used. The use of ADA made it easier to develop highly reliable systems and to manage complex applications. The APM_88 model is based upon a different "process model" called the ADA Process Model.

Among the assumptions are:

- ADA is used to produce compiler checked package specifications by the Product Design Review (PDR).
- Small design teams are used.
- More effort is spent in requirements analysis and design.
- Less effort is spent in coding, integration, and testing.

The APM_88 model incorporates changes to the equations, several cost drivers, and a variety of other tables.

Advantages of COCOMO'81

- COCOMO is transparent; one can see how it works unlike other models such as SLIM.
- Drivers are particularly helpful to the estimator to understand the impact of different factors that affect project costs.

Drawbacks of COCOMO'81

- It is hard to accurately estimate KDSI early on in the project, when most effort estimates are required.

- KDSI, actually, is not a size measure it is a length measure.
- Success depends largely on tuning the model to the needs of the organization, using historical data which is not always available.

Caution Use COCOMO model with very carefully because sometimes it becomes extremely vulnerable to mis-classification of the development mode.

5.4 COCOMO II

At the beginning an overall situation is given where the need of the reengineering of COCOMO I am stressed out as well as the focused issues for the new COCOMO version are presented.

5.4.1 Reengineering COCOMO I Needs

- New software processes
- New phenomena: size, reuse
- Need for decision making based on incomplete information

5.4.2 Focused Issues

The reengineering process focussed on issues such as:

1. Non-sequential and rapid-development process models
2. Reuse-driven approaches involving commercial-off-the-shelf (COTS) packages
3. Reengineering [reused, translated code integration]
4. Applications composition
5. Application generation capabilities
6. Object oriented approaches supported by distributed middleware
7. Software process maturity effects
8. Process-driven quality estimation

5.4.3 Strategy

1. Preserve the openness of the original COCOMO
2. Key the structure of COCOMO II to the future software marketplace sectors
3. Key the inputs and outputs of the COCOMO II sub models to the level of information available
4. Enable the COCOMO II submodels to be tailored to a project's particular process strategy (early prototyping stage [application composition model], Early Design stage, post-architecture stage)

The COCOMO II provides a family (COCOMO suite) of increasingly detailed software cost estimation models, each tuned to the sectors' needs and type of information available to support software cost estimation.

Three Primary Premises

Particular Process Drivers: Current and future software projects tailor their processes to their particular process drivers (no single, preferred software life cycle model anymore), process drivers include COTS/reusable software availability, degree of understanding requirements and architectures and other schedule constraints as size and required reliability.

Information: Consistency between granularity of the software model and the information available.

Range Estimates: Coarse-grained cost driver information in the early project stages, and increasingly fine-grained information in later stages. Not point estimates of software cost and effort, but rather range estimates tied to the degree of definition of the estimation inputs.

5.4.4 Differences between COCOMO I and COCOMO II

The major differences between COCOMO I AND COCOMO II are:

- COCOMO'81 requires software size in KDSI as an input, but COCOMO II is based on KSLOC (logical code). The major difference between DSI and SLOC is that a single Source Line of Code may be several physical lines. For example, an "if-then-else" statement would be counted as one SLOC, but might be counted as several DSI.
- COCOMO II addresses the following three phases of the spiral life cycle: applications development, early design and post architecture.
- COCOMO'81 provides point estimates of effort and schedule, but COCOMO II provides likely ranges of estimates that represent one standard deviation around the most likely estimate.
- The estimation equation exponent is determined by five scale factors (instead of the three development modes).
- Changes in cost drivers are:
  - Added cost drivers 7: DOCU, RUSE, PVOL, PLEX, LTEX, PCON, and SITE.
  - Deleted cost drivers (5): VIRT, TURN, VEXP, LEXP, and MODP.
  - Alter the retained ratings to reflect more up-do-date software practices.
- Data points in COCOMO I: 63 and COCOMO II: 161
- COCOMO II adjusts for software reuse and reengineering where automated tools are used for translation of existing software, but COCOMO'81 made little accommodation for these factors.
- COCOMO II accounts for requirements volatility in its estimates.

5.4.5 COCOMO II Model Definition

The COCOMO II provides three stage series of models for estimation of software projects:

Application Composition Model: This is used for earliest phases or spiral cycles (prototyping, and any other prototyping occurring later in the life cycle).

Early Design Model: This is for next phases or spiral cycles. Involves exploration of architectural alternatives or incremental development strategies. Level of detail consistent with level of information available and the general level of estimation accuracy needed at this stage.

Post-Architecture Model: Once the project is ready to develop and sustain a fielded system it should have a life-cycle architecture, which provides more accurate information on cost driver inputs, and enables more accurate cost estimates.

Self Assessment Questions

1. Burenius and Lindstedt .... as the machines of the project.

   (*a*) Setting up Milestones (*b*) Identify activities

   (*c*) Allocate resources (*d*) Create a time-schedule

2. The .... model computes effort as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle.

   (*a*) Intermediate COCOMO (*b*) Advanced COCOMO

   (*c*) detailed COCOMO (*d*) None of these

Notes

3. Reengineering COCOMO I needs:
   (*a*) New software processes
   (*b*) Applications composition
   (*c*) Application generation capabilities
   (*d*) Process-driven quality estimation
4. .... is for next phases or spiral cycles.
   (*a*) Application composition model (*b*) Early design model
   (*c*) Post-architecture model (*d*) None of these
5. The model of COCOMO-II is also called COCOMO'81 is presented.
   (*a*) True (*b*) False

5.5 Development Effort Estimates

Following are the factors which effort the development:

5.5.1 Sizing

A good size estimate is vital for a good model estimate. However determining size can be very demanding because projects are generally composed of new reused (with or without modifications) and automatically translated code. The baseline size in COCOMO II is a count of new lines of code.

SLOC is defined such that only source lines that are DELIVERED as part of the product are included test drivers and other support software (-as long as they are not documented as carefully as source code is excluded). One SLOC is one logical source statement of code (e.g. declarations are counted, comments not) In COCOMO II effort is expressed as person months (PM). Person month is the amount of time one person spends working on the software development project for one month. Code size is expressed in thousands of source lines of code (KSLOC). The goal is to measure the amount of intellectual work put into program development. Counting source lines of Code (SLOC) takes account of new lines of code (reused code has to be adjusted). There are two possibilities: either to count the source lines of code (with the Software Engineering Institute (SEII) checklists or tool supported) or to count the unadjusted function points and to convert them via "backfiring" tables to source lines of code. For further information on counting the unadjusted function points (UFP) as well as the ratios for the conversion.

5.5.2 The Scale Factors

Most significant input to the COCOMO II model is size. Size is treated as a special cost driver in that it has an exponential factor, E. This exponent is an aggregation of five scale factors. They are only used at the project level for both the Early Design and the Post-Architecture model. All scale factors have qualitative rating levels ('extra low' to 'extra high'). Each scale factor is the subjective weighted average of its characteristics.

The five Scale Factors are:

1. PREC Precedentedness (how novel the project is for the organization)
2. FLEX Development Flexibility
3. RESL Architecture / Risk Resolution
4. TEAM Team Cohesion
5. PMAT Process Maturity

5.6 Project Scheduling and Staffing

After establishing a goal on the effort front, we need to establish the goal for delivery schedule. With the effort estimate (in person-months), it may be tempting to pick any project duration based on convenience and then fix a suitable team size to ensure that the total effort matches the estimate. However, as is well known now, person and months are not fully interchangeable in a software project. Person and months can be interchanged arbitrarily only if all the tasks in the project can be done in parallel, and no communication is needed between people performing the tasks. This is not true for software projects—there are dependencies between tasks (e.g., testing can only be done after coding is done), and a person performing some task in a project needs to communicate with others performing other tasks. As Brooks has pointed out, "man and months are identical only for activities that require no communication among men, like sowing wheat or reaping cotton. This is not even approximately true of software."

However, for a project with some estimated effort, multiple schedules (or project duration) are indeed possible. For example, for a project whose effort estimate is 56 person-months, a total schedule of 8 months is possible with 7 people. A schedule of 7 months with 8 people is also possible, as is a schedule of approximately 9 months with 6 people. (But a schedule of 1 month with 56 people is not possible. Similarly, no one would execute the project in 28 months with 2 people.) In other words, once the effort is fixed, there is some flexibility in setting the schedule by appropriately staffing the project, but this flexibility is not unlimited. Empirical data also suggests that no simple equation between effort and schedule fits well.

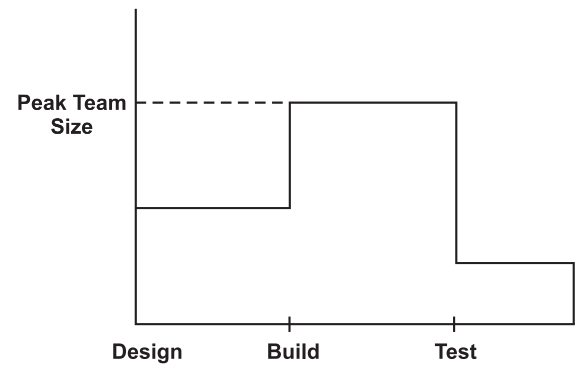
The objective is to fix a reasonable schedule that can be achieved (if suitable number of resources is assigned). One method to determine the overall schedule is to determine it as a function of effort. Such function can be determined from data from completed projects using statistical techniques like fitting a regression curve through the scatter plot obtained by plotting the effort and schedule of past projects. This curve is generally nonlinear because the schedule does not grow linearly with effort. Many models follow this approach. The IBM Federal Systems Division found that the total duration, M, in calendar months can be estimated by M = 4.1E.36. In COCOMO, the equation for schedule for an organic type of software is M = 2.5E.38. As schedule is not a function solely of effort, the schedule determined in this manner is essentially a guideline.

Another method for checking a schedule for medium-sized projects is the rule of thumb called the square root check. This check suggests that the proposed schedule can be around the square root of the total effort in person-months. This schedule can be met if suitable resources are assigned to the project. For example, if the effort estimate is 50 person-months, a schedule of about 7 to 8 months will be suitable. From this macro estimate of schedule, we can determine the schedule for the major milestones in the project. To determine the milestones, we must first understand the manpower ramp-up that usually takes place in a project. The number of people that can be gainfully utilized in a software project tends to follow the Rayleigh curve. That is, in the beginning and the end, few people are needed on the project; the peak team size (PTS) is needed somewhere near the middle of the project; and again fewer people are needed after that. This occurs because only a few people are needed and can be used in the initial phases of requirements analysis and design. The human resources requirement peaks during coding and unit testing, and during system testing and integration, again fewer people are required.

Often, the staffing level is not changed continuously in a project and approximations of the Rayleigh curve are used: assigning a few people at the start, having the peak team during the coding phase, and then leaving a few people for integration and system testing. If we consider design and analysis, build, and test as three major phases, the manpower ramp-up in projects typically resembles the function shown in Figure 5.7. For ease of scheduling, particularly for smaller projects, often the required people are assigned together around the start of the project. This approach can lead to some people being unoccupied at the start and toward the end. This slack time is often used for supporting project activities like training and documentation.

Given the effort estimate for a phase, we can determine the duration of the phase if we know the manpower ramp-up. For these three major phases, the percentage of the schedule consumed in the build phase is smaller than the percentage of the effort consumed because this phase involves more people. Similarly, the percentage of the schedule consumed in the design and testing phases exceeds their effort percentages. The exact schedule depends on the planned manpower ramp-up, and how many resources can be used effectively in a phase on that project. Generally speaking, design requires about a quarter of the schedule, build consumes about half, and integration and system testing consume the remaining quarter. COCOMO gives 19% for design, 62% for programming, and 18% for integration.

Figure 5.7: Manpower Ramp-up in a Typical Project



5.7 Software Configuration Management

Configuration management (CM) is the regulation of domineering the evolution of complex systems; software configuration management (SCM) is its specialization for computer programs and associated documents. General CM is beneficial for any large system that. Due to its complexity, cannot be made perfect for all the uses to which it will be put. Such a system will be subject to numerous, sometimes conflicting changes during its lifetime, giving rise not to a single system, but to a set of related systems, called a system family. A system family consists of a number of components that can be configured to form individual family members. A substantial number of the components must be shared among members to make the family economically viable. Maintaining order in large and expanding system families is the goal of CM.

The SCM differs from general CM in the following two ways. First, software is easier to change than hardware and it therefore changes faster. Even relatively small software systems, developed by a single team, can experience a significant rate of change, and in large systems, such as telecommunications systems, the update activities can totally overwhelm manual configuration management procedures. Second, SCM is potentially more automatable.

Effective software configuration management coordinates programmers working in teams. It improves productivity by reducing or eliminating some of the confusion caused by interaction among team members. The coordinating functions of configuration management are introduced below, and illustrated with questions or statements familiar to anyone who has worked in software development.

Identification

Identifying the individual components and configurations is a prerequisite for controlling their evolution.

Reliable identification helps avoid the following problems:

- "This program worked yesterday. What happened?"
- "I cannot reproduce the error in this configuration."
- "I fixed this problem long ago. Why did it reappear?"
- "The online documentation does not match the program."
- "Do we have the latest version?"

Change Tracking

Change tracking keeps a record of what was done to which component for what reason, at what time, and by whom. It helps answer the following questions:

- "Has this problem been fixed?"
- "Which bug fixes went into this copy?"
- "This seems like an obvious change. Was it tried before?"
- "Who is responsible for this modification?"
- "Were these independent changes merged?"

Version Selection and Baselining

Selecting the right versions of components and configurations for testing and baselining can be difficult. Machine support for version selection helps with composing consistent configurations and with answering the following questions:

- "How do I configure a test system that contains my temporary fixes to the last baseline, and the released fixes of all other components?"
- "Given a list of fixes and enhancements, how do I configure a system that incorporates them?"
- "This enhancement would not be ready until the next release. How do I configure it out of the current baseline?"
- "How exactly does this version differ from the Baseline?"

Software Manufacture

Putting together a configuration requires numerous steps such as pre- and post processing, compiling, linking, formatting, and regression testing. SCM systems must automate that process and at the same time should be open for adding new processing programs. To reduce redundant work, they must manage a cache of recently generated components.

Automation avoids the following problems:

- "I just fixed that. Was something not recompiled?"
- "How much recompilations will this change cost?"
- "Did we deliver an up-to-date binary version to the customer?
- "I wonder whether we applied the processing steps in the right order."
- "How exactly was this configuration produced?"
- "Were all regression tests performed on this version?

Managing Simultaneous Update

Simultaneous update of the same component by several programmers cannot always be prevented. The configuration management system must note such situations and supply tools for merging competing changes later. In so doing it helps prevent problems like the following:

- "Why did my change to this module disappear?"
- "What happened to my unfinished modules while I was out of town?"
- "How do I merge these changes into my version?"
- "Do our changes conflict?"

The software tools for automating the functions introduced. The basis of all tools is representation, so we develop a model for representing multi-version multi configuration systems.

5.7.1 Basic SCM Concepts

The basic elements of a data base for software configuration management. The data base stores all software objects produced during the life-cycle of a project.

A *software object* is any kind of identifiable, machine-readable document generated during the course of a project. The document must be stored online to be fully controllable by an SCM system. Examples of software objects are requirements documents, design documents, specifications, interface descriptions, program code, test programs, test data, test output, binary code, user manuals, or VLSI designs.

Every software object has a unique identifier and a body containing the actual information. A set of attributes associated with software objects and a facility for linking objects via various relations are also needed. For example, attributes record time of creation and last read access, and relations link objects to their revisions and variants. The set of attributes and relations must be extensible; later sections will introduce a basic set. We also need a facility to describe subclasses or subtypes of the general software object. For instance, the subclass may fix the language in which the body is written, or the structure editor used to compose the body, or whether the object represents an interface or an implementation. The subclass also defines the set of operations available on objects of that class, such as compiling, configuring, printing, etc.

The body of a software object is immutable, that is, once the body has been completed, it can only be read. Any "change" of a body actually creates a new software object with the changed body. Immutability is important for configuration management, because it prevents misidentification: an object identifier is associated with one and only one constant body, and not with several different versions. Most other attributes and relations of software objects remain changeable; however, so new information can be added.

Software objects have two orthogonal refinements, one according to how they were created, the other according to the structure of their body. For creation, we distinguish source and derived objects. For internal structure, we distinguish atomic objects and configurations.

5.7.2 Creation of Software Objects

A source object is a software object that is composed manually, for instance with an interactive editor. Creating a source object requires human action; it cannot be produced automatically.

A derived object is generated fully automatically by a program, usually from other software objects. A program that produces derived objects is called a deriver. Examples of derivers are compilers, linkers, document formatters, pretty printers, cross references, and call graph generators. Normally, derived objects need not be stored, since they can be regenerated, provided both the deriver and the input are available or can be re derived. To reduce the delay caused by regeneration, a smart configuration management system maintains a cache of derived objects that are likely to be reused.

Unlike derived objects, which can be deleted to make room, source objects are "sacred", because deleting them may cause irreparable damage or at least significant delay until they are reconstructed. However, derived objects may also become "sacred", i.e., they must not be deleted merely to make room, if it is impossible or time consuming to reproduce them. For instance, derived objects that are imported from other sites, especially vendor supplied programs, must not be deleted, even though they are derived in most cases. Another example are derived objects for which the original derivers have stopped working (if they have not been ported to new hardware, say), or if the corresponding input objects have been lost.

A special case is derived objects that are modified manually. Examples are automatically generated program skeletons and templates that are fleshed out by hand, or object code that is patched manually. In principle, these manual modifications produce new source objects. However, the SCM system should store a traceability link that records the dependency between the two objects. This link can be used for generating a reminder to update the source object if the derived object changes. Traceability links should also be recorded among dependent source objects, for example between a specification and its implementation, or a program and its documentation. In fact, most source objects in an SCM system depend on one or more other objects except perhaps the initial requirements specification. Traceability information is extremely valuable for automatically producing update reminders, for reviewing completeness of changes, and for informing maintainers what information they need to consider when preparing a change.

5.7.3 Structure of Software Objects

The body of a software object is either atomic or structured. An atomic object, or atom, has a body that is not decomposable for SCM; its body is an opaque data structure with a set of generic operations such as copying, deletion, renaming, and editing. An atomic object may consist of a program written in some language, a syntax tree produced by a structure editor, a data structure generated by a WYSIWYG word processor, or an object code module produced by a compiler.

A configuration has a body that consists of sub-objects, which may themselves have sub objects, and so on. Configurations have two subclasses: composites and sequences. A composite object, or simply composite, is a record structure comprised of fields. Each field consists of a field identifier and s field value. A field value is either an object identifier or a version group identifier. An example of a composite object is a software package consisting of a program, a user's manual, and an installation procedure. Another example is a regression test object, consisting of a test program, input data, expected output data, and a comparator for comparing expected and actual output. Thus, fields may contain data as well as operations.

A sequence is a list of object and version group identifiers. Sequences represent ordered multisite of objects. They are used for combining sub-objects that are of the same class, or when the number of sub-objects is indeterminate. In contrast to composites, the individual elements of a sequence fulfil identical roles and are treated in the same way for SCM purposes, such as the list of object code modules constituting a library.

Because of the need to distinguish between "precise" and "loose" configurations, we introduce the following terms. A generic composite is a composite with at least one field value that is either a version group identifier or a generic configuration (i.e., a generic composite or a generic sequence). The opposite of a generic composite is a baseline composite, which is a composite whose field values are atomic objects, baseline composites, or baseline sequences. The subclasses generic sequences and baseline sequence are defined analogously. Finally, a generic configuration, also called a system model, is a generic composite or a generic sequence. A baseline configuration, or simply baseline, is a baseline composite or baseline sequence.

In both composites and sequences, source and derived objects may be freely intermixed. However, including derived objects presents a problem: Since the derived objects may not yet exist, there may be no known identifiers for them. Instead, we must represent a derived object with a descriptor that will cause the object to be generated when it is needed. This descriptor must specify not only the derivers, but all parameter settings for the derivers as well.

For clarity, we should point out some uses of the above definitions. Suppose a software house delivers a single, binary program to a customer. This program is a single, derived object. It most cases, this object was generated from a baseline configuration recorded at the software house. The purpose of the baseline is to guarantee that the derived object can be reproduced when needed. The software house may also deliver a configuration, perhaps a composite that consists of one or more binaries and a manual. The delivered configuration may also contain source programs, because the programs will be interpreted, or because the customer wishes to compile source locally. The customer may also need to adapt the source code to local needs. Thus, depending on how much the customer expects to do, a more or less complete SCM system must be available at the customer site to take over portions of the software house's SCM functions.

5.8 Quality Plan

A quality plan describes how an organisation will achieve its quality objectives. It describes the quality objectives and specifies the quality assurance and control activities to be performed in day-to-day company operations. In the case of a software development organisation individual quality plans may be prepared for each software or systems engineering project.

ISO 9001 requires top management to "ensure that the planning of the quality management system is carried out."

Figure 5.8: Planning for Quality at the Organisation and Project Levels

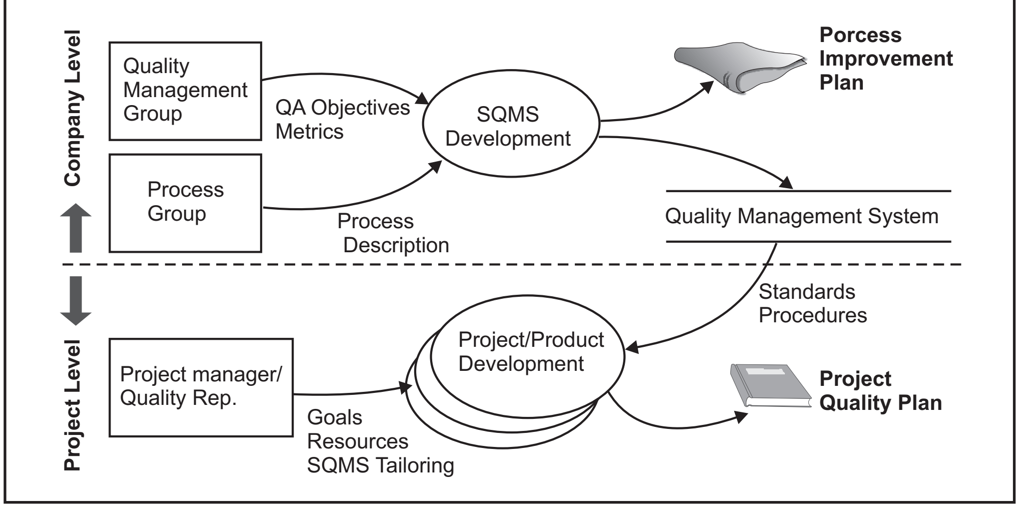


Figure 5.8 shows at the company level quality planning addresses development, maintenance and improvement of the overall quality management system. At the project level quality planning applies the quality management system to individual projects.

5.8.1 Organisational Quality Plan Content

An organisational quality plan is typically prepared by the Quality Manager and covers the following issues:

- ***Quality Objectives and Goals***: The overall objectives of the quality management programme together with measurable goals to be achieved.
- ***Quality Management System Scope***: Who and what will be impacted by the quality management system. For example: process scope, product scope, organizational scope.

- ***Organization & Responsibilities***: Organisational structure, reporting relationships and roles and responsibilities for quality.
- ***Resource Requirements***: Identification of the people and resources required to develop, use, maintain and improve the quality management system.
- ***Cost Benefit Analysis***: A quality program cost benefit analysis addressing issues such as: the cost of poor quality, the cost to improve quality and the cost benefits to be achieved.
- ***Activities and Deliverables***: The quality management system elements will be produced/ improved the quality management system development activities required.
- ***Schedule***: The timeframes in which the work will be achieved together with major milestones for quality management system element delivery, review and deployment.
- ***Risk Analysis***: An analysis of what could go wrong together with strategies for risk reduction.

5.8.2 Project Quality Plan Content

A project quality plan describes the tailoring of an organisation's quality management system for a particular project. The Institute of Electrical and Electronics Engineers (IEEE) Std 730 Standard for Software Quality Assurance Plans2 provides a well tested framework to work from.

Table 5.3: IEEE Standard 730 Standard for Software Quality Assurance Plans

1. Purpose	9. Tools, techniques, and methodologies
2. Reference documents	10. Media control
3. Management	11. Supplier control
4. Documentation	12. Records collection, maintenance, and retention
5. Standards, practices, conventions, and metrics	13. Training
6. Software reviews	14. Risk management
7. Test	15. Glossary
8. Problem reporting and corrective action	16. SQAP change procedure and history.

5.9 Risk Management

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics:

- ***Uncertainty***: The event that characterizes the risk may or may not happen; i.e. there are no 100% probable risks.
- ***Loss***: If the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel, resource, customer, and requirements problems and their impact on a software project. Project complexity, size, and the degree of structural uncertainty were also defined as project risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interfacing, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are:

1. Building an excellent product or system that no one really wants
2. Building a product that no longer fits into the overall business strategy for the company
3. Building a product that the sales force does not understand how to sell
4. Losing the support of senior management due to a change in focus or a change in people and
5. Losing budgetary or personnel commitment. It is extremely important to note the simple categorization would not always work. Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette. Known risks are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

5.9.1 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan. By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that are generic risks and product-specific risks. Generic risks are a potential threat to every software project. Product-specific risks can only be identified by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "what special characteristics of this product may threaten our project plan?"

Both generic and product-specific risks should be identified systematically. Tom glib drives this point home when he states: "if you do not actively attack the risks they will actively attack you".

One method for identifying risks is to create a risk item checklist. The Checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories.

- ***Product Size***: Risks associated with the overall size of the software to be built or modified.
- ***Business Impact***: Risk associated with constraints imposed by management or the marketplace.

- ***Customer Characteristics***: Risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- ***Process Definition***: Risks associated with the degree to which the software process has been defined and is followed by the development organization.
- ***Development Environment***: Risks associated with the availability and quality of the tools to be used to build the product.
- ***Technology to be built***: Risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- ***Staff Size and Experience***: Risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics noted above can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, asset of "risk components and drivers" are listed along with their probability of occurrence.

Product Size Risks

Few experienced managers would debate the following statement: Project risk is directly proportional to product size. The following risk item checklist identifies generic risks associated with product size:

- Estimated size of the product in LOC or FP?
- Degree of confidence in estimated size estimate?
- Estimated size of product in number of programs, files, and transactions?
- Percentage deviation in size of product from average for previous products?
- Size of database created or used by the product?
- Number of users of the product?
- Number of projected changes to the requirements for the product? Before delivery? After delivery?
- Amount of reused software?

In each case, the information for the product to the developed must be compared to past experience. If a large percentage deviation occurs or if numbers are similar, but past results was considerably less than satisfactory, risk is high.

Business Impact Risks

An engineering manager at a major software company placed the following framed plaque on his wall: "god grants me brains to be a good project manager and the common sense to run like hell whenever marketing sets project deadlines!". The marketing department is driven by business considerations, and business considerations sometimes come into direct conflict with technical realities. The following risk item checklist identifies generic risks associated with business impact:

- Effect of this product on company revenue?
- Visibility of this product to senior management?
- Reasonableness of delivery deadline?
- Number of customers who will use this product and the consistency of their needs relative to the product?

- Number of other products/systems with which this product must be interoperable?
- Sophistication of end users?
- Amount and quality of product documentation that must be produced and delivered to the customer?
- Governmental constraints on the construction of the product?
- Costs associated with late delivery?
- Costs associated with a defective product?

Each response for the product to be developed must be compared to past experience. If large percentage deviations occurs or if numbers are similar, but past results were considerably less than satisfactory, risk is high.

Customer Related Risks

All customers are not created equal. Pressman and Herron discuss this issue when they state.

Customers have different needs. Some know what they want; others know what they do not want. Some know what they want others know what they do not want. Some customers are willing to sweat the details, while others are satisfied with vague promises.

Customers have different personalities. Some enjoy being customers-the tension, the negotiation, and the psychological rewards of a good product. Others would prefer not to be customers at all. Some will happily accept almost anything that is delivered and make the very best of a poor product. Others will complain bitterly when quality is lacking, some will show their appreciation when quality is good a few will complain no matter what.

Customers also have varied associations with their suppliers. Some know the product and producer well. Others may be faceless, communicating with the producer only by written correspondence and a few hurried telephone calls.

Customers are often contradictory. They want everything yesterday for free. Often, the producer is caught among the customer's own contradictions.

A bad customer can have a profound impact on a software team's ability to complete a project on time and within budget. A bad customer represents a significant threat to the project plan and a substantial risk for the project manager. The following risk item checklist identifies generic risk associated with different customers.

- Have you worked with the customer in the past?
- Does the customer have a solid idea of what is required? Has the customer spent the time to write it down?
- Will the customer agree to spend time in formal requirements gathering meetings to identify project scope?
- Is the customer willing to participate in reviews?
- Is the customer technically sophisticated in the product area?
- Is the customer willing to let your people do their job that is, will the customer resist looking over your shoulder during technically detailed work?
- Does the customer understand the software process?

Process Risks

If the software process is ill defined if analysis design and testing are conducted in an ad-hoc fashion if quality is a concept that everyone agrees is important, but no one acts to achieve in any tangible way, then the project is at risk. The questions themselves have been adapted from the Software engineering Institute software process assessment questionnaire.

- Does your senior management support a written policy statement that emphasizes the importance of standard process for software development?
- Has your organization developed a written description of the software process to be used on this project?
- Are staff members signed up to the software process as it is documented and willing to use it?
- Is the software process used for other projects?
- Has your organization developed or acquired a series of software engineering training courses for managers and technical staff?
- Are published software engineering standards provided for every software developer and software manager?
- Have document outlines and examples been developed for all deliverable defined as part of the software process?
- Are formal technical reviews of the requirements specification, design, and code conducted regularly?
- Are formal technical reviews of test procedures and test cases conducted regularly?
- Are the results of each formal technical review documented, including errors found and resources used?
- Is there some mechanism for ensuring that work conducted on a project conforms to software engineering standards?
- Is configuration management used to maintain consistency among system/software requirements, design, code and test cases?
- Is a mechanism used for controlling changes to customer requirements that impact the software?
- Is there a documented statement of work, a software requirements specification, and a software development plan for each subcontract?
- Is a procedure followed for tracking and reviewing the performance of subcontractors?

Technical Issues

- Are facilitated application specification techniques used to aid in communication between the customer and developer?
- Are specific methods used for software analysis?
- Do you use a specific method for data and architectural design?
- Is more that 90% of your code written in a high-order language?
- Are specific conventions for code documentation defined and used?
- Do you use specific methods for test case design?
- Are software tools used to support planning and tracking activities?
- Are configuration management software tools used to control and track change activity throughout the software process?

- Are software tools used to support the software analysis and design process?
- Are tools used to create software prototypes?
- Are software tools used to support the testing process?
- Are software tools used to support the production and management of documentation?
- Are quality metrics collected for all software projects?
- Are productivity metrics collected for all software projects?

If a majority of the above questions are answered "no", software process is weak and risk is high.

Technology Risk

Pushing the limits of the technology is challenging and exciting. It is the dream of almost every technical person, because it forces a practitioner to use his or her skills to the fullest. But it is also very risky.

Murphy's Law seems to hold sway in this part of the development universe, making it extremely difficult to foresee risks, much less plan for them.

The following risk item checklist identifies generic risks associated with the technology to be built:

- Do the customer's requirements demand the creation of new algorithms or input or output technology?
- Does the software interface with new or proven hardware?
- Does the software to be built interface with vendor supplied software products that are unproven?
- Does the software to be built interface with a database system whose function and performance have not been proven in this application area?
- Is a specialized user interface demanded by product requirements?
- Do requirements for the product demand the creation of program components that are unlike any previously developed by your organization?
- Do requirements demand the use of new analysis, design, or testing methods?
- Do requirements demand the use of unconventional software development methods such as formal methods, AI-based approaches, and artificial neural networks?
- Do requirements put excessive performance constraints on the product?
- Is the customer uncertain that the functionality requested is "doable"?

If the answer to any of these questions is "yes" further investigation should be undertaken to assess risk potential.

5.10 Project Monitoring

Project Monitoring is separate from both project management and construction monitoring, and can be defined as:

"Protecting the Client's interests by identifying and advising on the risks associated with acquiring an interest in a development that is not under the Client's direct control."

A monitoring system can be defined as an observation system for the project managers to verify whether the project activities are happening according to planning and whether means are used in a correct and efficient manner. The system must supply the project management with a continuous flow of information throughout the course of the project to make it possible to take the right decisions. Monitoring is limited to the relation between the implementation of the activities and the results, in which the results are directly and only determined by the project activities.

Project Monitoring is carried out on behalf of a range of alternative client types including, for example:

- A funding institution, which will acquire the scheme as an investment upon completion.
- A tenant or purchaser which enters into a commitment to lease or purchase a property upon completion.
- A Bank or other development finance company where a loan matures at the end of the development period.
- Grant funders; or
- Private finance initiative funders and end users.

Although the risk profile of each of these client groups differ, in each case the development will be designed, constructed and supervised by a Developer who will employ a design and construction team. The appointment of a Project Monitor does not replace any of these primary functions, but protects the client's interests by monitoring the performance of the Developer and its team.

The Project Monitor's role is one of investigator and advisor to the Client. The Project Monitor is not there as a project manager or director and does not take away any of the responsibilities the Developer, and/or its design and construction team, has under funding or other development agreements. The Project Monitor is there to advise the Client on the risks associated with a development and protect the Client's interests in the development as it proceeds.

If the Client chooses to act on this advice then he or she will endeavour to reach agreement with the Developer separately. Essentially, it is for the Client to decide upon the level of acceptable risk and if appropriate, take executive action. By being proactive and providing accurate and timely advice the Project Monitor is able to improve the Client's decision making competency.

The Project Monitor provides an independent and impartial assessment of the project as it progresses and in so doing gives the Client a managing tool for risk and for protecting his or her interests objectively.

It is recommended that Project Monitor is proactive rather than reactive, and act as an early warning system for the Client by anticipating potential issues which may affect the delivery of the project. In so doing the Project Monitor can also add value to the Developer's team for example, being proactive will keep the Client abreast of material changes to the development or emerging risks and facilitate informed and improved decision making.

Typically a Project Monitor will advise upon:

- Land and property acquisition matters.
- Statutory consents.
- Competency of the Developer, its team and any proposed project management systems.
- Financial appraisals.
- Development, finance, consultancy and construction agreements.
- Construction costs and programs.
- Design and construction quality.

Notes

Historically the Project Monitor has come from a number of professional backgrounds, such as building surveying, quantity surveying and project management. It is important, therefore, that Clients map the scope of service they require with the competencies available from different Project Monitors and thus choose the most appropriate Project Monitor.

As well as an assessment of the Project Monitor's technical competencies Clients should also consider the Project Monitor's performance record and sector knowledge.

Sector knowledge is particularly important as an understanding of the key risks and value drivers within, for example, the residential, commercial, retail, public and PFI sectors will help the Project Monitor to deliver added value to the Client and the wider development team.

Case Study

Quality Assurance and Quality Control during Construction are of utmost Importance

Consultancy services during the construction of chimneys, including Quality Assurance and Quality Control, are of the utmost importance, because even minor construction failures may destabilise a whole chimney and lead to serious material damages and personal injuries.

Due to the fact that chimney construction is performed at lofty heights and very often in remote areas that lack the basic infrastructure, the biggest challenge in such projects is to find experts with the necessary extensive technical expertise and experience in the construction sector.

In order to ensure the best quality throughout the whole construction process and to minimise the risk of failures, Bygging India Ltd., the leading construction company in the chimney, silo and storage tanks field in India, sought a reliable third-party inspection and testing company.

In September 2010, the SGS in India was awarded a contract to conduct specialist third-party services during the construction of chimneys at 15 construction sites in India, thanks to its first-rate competence and know-how in projects of this kind.

SGS Quality Assurance and Quality Control for the Construction of Chimneys in India

From September 2010 to September 2011, SGS performed comprehensive Quality Assurance and Quality Control services for Bygging India Ltd., to ensure top quality in a wide range of their chimney construction projects.

Throughout the entire contract, SGS has verified materials, parts and final products through checks, audits, inspections and witnessing. With Quality Assurance SGS ensures that the chimneys meet predefined quality standards and will perform satisfactorily during service, and with Quality Control SGS focuses on the actual measurement, testing and supervision of the final product, by conducting inspection of each unit and sample testing.

Drawing on its extensive experience and expertise in the construction sector, SGS guarantees the quality of chimneys with respect to defined Indian standards and regulations, such as IS codes and MORTH specifications. An inspection of the construction process was performed in accordance with a checklist that SGS developed for this project, in line with approved specifications and standard operating procedures.

A team of at least two SGS experts at each construction site has conducted Quality Assurance and Quality Control checks at each 200 mm layer during the chimney's construction stage and checked it for any discontinuities or gaps and their sizes, as well as the overall construction quality in terms of work, form and the material used in the structure.

Contd...

In addition, SGS has ensured top quality during the construction process with the slip form technique, a construction method used for tall structures such as chimneys, water tanks and storage bins that is becoming increasingly popular among leading construction companies, due to the considerable reduction in time and cost it offers.

SGS has checked the twisting of the slip form with respect to tolerance limits, controlled the Reinforced Concrete shell thickness and slump as per design requirements, and has ensured that the proper inner and outer diameters, clean cover and steel lap joints are maintained. With SGS's help, the chimney construction projects have been performed in compliance with all international and local regulations and quality standards. SGS is one of very few companies with the requisite technical expertise and manpower to supervise chimney construction. As a result of SGS's competence and experience, the contract with Bygging India Ltd. is being renewed annually, in accordance with the client's requirements. Moreover, SGS operations may be extended to other countries in which Bygging India Ltd. carries out its work.

Questions

1. Explain the quality assurance and quality control during construction of a project
2. Define the quality assurance and quality control for the construction of chimneys in India.

Self Assessment Questions

6. Objective of software project planning is to provide a ………… that enables the manager to make reasonable estimates of resources, cost, and schedule.

   (*a*) framework (*b*) setup

   (*c*) program (*d*) None of these.

7. ……….. central plan, which describes how the system will be developed.

   (*a*) Quality Assurance Plan (*b*) Validation Plan

   (*c*) Maintenance Plan (*d*) Software Development Plan

8. ………………………… defines how the system will be configured and installed.

   (*a*) Validation Plan (*b*) Configuration Management Plan

   (*c*) Quality Assurance (*d*) Software Development Plan

9. A software project is developed through ……………………

   (*a*) development processes (*b*) programmer

   (*c*) software processes (*d*) life cycle of software

10. A requirement specification describing the details of the project to be developed forms a point of departure for the …………… process.

    (*a*) life cycle (*b*) execution

    (*c*) estimation (*d*) maintenance

11. The Advanced COCOMO model computes ………… as a function of program size and a set of cost drivers weighted according to each phase of the software lifecycle.

    (*a*) cost of the program (*b*) range of the program

    (*c*) program (*d*) effort

12. …………….. is a consistency between granularity of the software model and the information available.
    (*a*) Information (*b*) Particular Process Drivers
    (*c*) Range Estimates (*d*) None of these.
13. Application Composition Model is used for earliest phases or ……………..
    (*a*) spiral cycles (*b*) life cycle
    (*c*) water fall phase (*d*) process drivers
14. Most significant input to the COCOMO II model is ………………
    (*a*) range (*b*) cost
    (*c*) spiral (*d*) size
15. The model of COCOMO I also called COCOMO'61.
    (*a*) True (*b*) False

5.11 Summary

- Project Monitor is a key individual, who provide impartial and concise support to Clients who have an interest in a development project.
- Software project planning is a Level 2 Capability Maturity Model (CMM) Key Process Area (KPA).
- Metrics are collected on each process step and are then used to develop process improvements in the SPP process.
- Judgment-based estimation is the most frequently applied estimation approach in the software industry today.
- Metrics are collected on each process step and are then used to develop process improvements in the SPP process.
- The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule should attempt to define "best case" and "worst case" scenarios so that project outcomes can be bounded.
- The software planning begins with a statement of the work to be performed and other constraints and goals that define and bound the software project.
- A milestone in a project could be an approved deliverable the milestone-chart only needs to be complemented with a detailed plan containing activities for the next step in the project.

5.12 Keywords

Art Gallery: This scenario is a and it will help you to determine what skills, materials, equipment, time and costs you will need to achieve your objectives that are stated to the left of the chart.

Configuration Management Plan: It defines how the system will be configured and installed.

Entry Criteria: The elements and conditions necessary to be in place to begin a process activity. Reading lower level activities assumes that the entry criteria for all higher level activities have been satisfied.

Quality Assurance Plan: It specifies the quality procedures and standards to be used.

Software Development Plan: The central plan, which describes how the system will be developed.

Software Effort Estimation: It is a huge challenge, particularly as it is an area in which miscalculations can result in delays, low quality software, or the loss of contracts.

Software Project Planning (SPP): This is the process begins with the planning initiation step.

Software Project Planning: It involves developing estimates for the work to be performed, establishing the necessary commitments, and defining the plan to perform the work.

Software Project: It is developed through software processes in which common activities comprise specification, design and implementation, validation and evolution

Validation Plan: It defines how a client will validate the system that has been developed.

Lab Exercise

1. Prepare a flow chart for implementation of COCOMO model.
2. Create a risk identification chart.

5.13 Review Question

1. What is the software project planning?
2. What is the process planning and discuss its needs?
3. What is the effort estimation of software project planning?
4. Discuss the COCOMO model and its features.
5. What are the differences between COCOMO-I and COCOMO-II?
6. What is intermediate COCOMO?
7. What are the effort estimates for developments?
8. What is the process of scheduling and staffing for a software project?
9. What is software configuration management?
10. What is the term quality planning and risk management?

Answers of Self Assessment Questions

1.	(*b*)	2.	(*b*)	3.	(*a*)	4.	(*b*)	5.	(*b*)
6.	(*a*)	7.	(*d*)	8.	(*b*)	9.	(*c*)	10.	(*c*)
11.	(*d*)	12.	(*a*)	13.	(*a*)	14.	(*d*)	15.	(*b*)

5.14 Further Readings

Books

"*Software Engineering*", By-Sommerville

"*Software Engineering: Theory and Practice*" By- Shari Lawrence Pfleeger, Joanne M. Atlee

Online link

http://cisas.unipd.it/didactics/STS_school/Software_development/Guide_to_the_SW_project_mangement-0508.pdf

Unit 6: Functional Design

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the functional design
- Define functional design principles
- Understand abstraction
- Discuss the basic modularity
- Explain the top down approach
- Discuss the bottom up approach
- Understand coupling
- Explain the cohesion
- Discuss the structure charts
- Explain the data flow diagrams
- Define design heuristics

Introduction

A design method is a systematic approach to creating a design by applying of a set of techniques and rule. Most design methodologies focus on the system design, and do not reduce the design action to a sequence of steps that can be blindly followed by the designer. In function-oriented methods for design and describe one exacting methodology—the structured design methodology—in some detail. In a function-oriented design approach, a system is viewed as a transformation function, transforming the inputs to the desired out-puts. The purpose of the design phase is to specify the components for this transformation function, so that each component is also a transformation function. That is, each module in design supports a functional abstraction. The basic output of the system design phase, when a function oriented design approach is being followed, is the definition of all the major data structures in the system, all the major modules of the system, and how the modules interact with each other. The design activity begins when the requirements document for the software to be developed is available and the architecture has been designed. During design we further refine the architecture. Generally, design focuses on the we have called the module that is, during design we determine what modules should the system have and which have to be developed. Sometimes, the module view may effectively be a module structure of each component in the architecture. That is, the design exercise determines the module structure of the components. However, this simple mapping of components and modules may not always hold. In that case we have to ensure that the module view created in design is consistent with the architecture. Functional Design is a paradigm used to simplify the design of hardware and software devices such as computer software and increasingly, 3D models. A functional design assures that each modular part of a device has only one responsibility and performs that responsibility with the minimum of side effects on other parts. Functionally designed modules tend to have low coupling.

The advantage for implementation is that if a software module has a single purpose; it will be simpler, and therefore easier and less expensive, to design and implement. Systems with functionally designed parts are easier to modify because each part does only what it claims to do. Since maintenance is more than 3/4 of a successful system's life, this feature is a crucial advantage. It also makes the system easier to understand and document, which simplifies training. The result is that the practical lifetime of a functional system is longer. In a system of programs, a functional module will be easier to reuse because it is less likely to have side effects that appear in other parts of the system.

6.1 Concept of Functional Design

The standard way to assure functional design is to review the description of a module. If the description includes conjunctions such as "and" or "or", then the design has more than one responsibility, and is therefore likely to have side effects. The responsibilities need to be divided into several modules in order to achieve a functional design.

A functional requirement defines a function of a software system or its component. A function is described as a set of inputs, the behaviour, and outputs (see also software). Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish. Behavioural requirements describing all the cases where the system uses the functional requirements are captured in use cases. Functional requirements are supported by non-functional requirements (also known as quality requirements), which inflict constraints on the design or implementation (such as performance requirements, security, or reliability). Generally, functional requirements are expressed in the form "system must do <requirement>", while non-functional requirements are "system shall be <requirement>". The plan for implementing functional requirements is detailed in the system design. The plan for implementing non-functional requirements is detailed in the system architecture.

As defined in requirements engineering, functional requirements specify particular results of a system. This should be contrasted with non-functional requirements which specify overall characteristics such as cost and reliability. Functional requirements drive the application architecture of a system, while non-functional requirements drive the technical architecture of a system. In some cases a requirements analyst generates use cases after gathering and validating a set of functional requirements. The hierarchy of functional requirements is: user/stakeholder request -> feature -> use case -> business rule. Each use case illustrates behavioural scenarios through one or more functional requirements. Often, though, an analyst will begin by eliciting a set of use cases, from which the analyst can derive the functional requirements that must be implemented to allow a user to perform each use case.

6.2 Principles of Functional Design

The design of a system is accurate if a system built accurately according to the design satisfies the supplies of that system. Clearly, the goal during the design phase is to produce right designs. However, correctness is not the sole decisive factor during the design phase, as there can be many correct designs. The aim of the design process is not simply to produce a design for the system. Instead, the goal is to find the best probable design within the limitations imposed by the requirements and the physical and social setting in which the system will operate. To evaluate a design, we have to specify some properties and criteria that can be used for evaluation. Ideally, these properties should be as quantitative as possible. In that situation we can precisely evaluate the “goodness” of a design and determine the best design. However, criteria for quality of software design are often subjective or non-quantifiable. In such a situation, criteria are essentially thumb rules that aid design evaluation.

A design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements). However, the two most important properties that concern designers are efficiency and simplicity. Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system is one that consumes less processor time and requires less memory. In earlier days, the efficient use of CPU and memory was important due to the high cost of hardware. Now that the hardware costs are low compared to the software costs, for many software systems traditional efficiency concerns now take a back seat compared to other conside rations. One of the exceptions is real-time systems, for which there are strict execution time constraints. Simplicity is perhaps the most important quality criteria for software systems. We have seen that maintenance of software is usually quite expensive. Maintainability of software is one of the goals we have established. The design of a system is one of the most important factors affecting the maintainability of a system. During maintenance, the first step a maintainer.

6.2.1 Seven Principles of Functional Design

The Reason It All Exists

A software system exists for one reason: To provide value to its users. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: “Does this add real VALUE to the system?” If the answer is “no”, do not do it. All other principles support this one.

KISS (Keep It Simple, Stupid!)

Software design is not a haphazard process. There are many factors to consider in any design effort. All design should be as simple as possible, but no simpler. This facilitates having a more easily understood, and easily maintained system. This is not to say that features, even internal features, should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

Maintain the Vision

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. The most important consideration in system design, Stroustrup also notes: Having a clean internal structure is essential to constructing a system that is understandable, can be extended and reorganized, and is maintainable and testable. From which Brooch summarizes: It is only through having a clear sense of systems architecture that it becomes possible to discover common abstractions and mechanisms. Exploiting this commonality ultimately leads to systems that are simpler, and therefore smaller and more reliable. Compromising the architectural vision of a software system weakens and will eventually break even the most well designed systems. Having an empowered Architect who can hold the vision and enforce compliance helps ensure a very successful software project.

What You Produce, Others Will Consume

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, always specify, design, and implement knowing someone else will have to understand what you are doing. The audience for any product of software development is potentially large. Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: Be Open to the Future

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete when just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. Never design yourself into a corner. Always ask "what if ", and prepare for all possible answers by creating systems that solve the general problem, not just the specific one. This could very possibly lead to the reuse of an entire system.

Plan Ahead for Reuse

Reuse saves time and effort. Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. There are many techniques to realize reuse at every level of the system development process. Those at the detailed design and code level are well known and documented. New literature is addressing the reuse of design in the form of software patterns. However, this is just part of the battle. Communicating opportunities for reuse to others in the organization is paramount. How can you reuse something that you do not know exists? Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

Think!

This last Principle is probably the most overlooked. Placing clear, complete thought before action almost always produces better results. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes valuable experience. A side effect of thinking is learning to recognize when you do not know something, at which point you can research the answer. When clear thought has gone into a system, value comes out. Applying the first six Principles requires intense thought, for which the potential rewards are enormous.

The following other some points in Principles of Functional Design

- The design process should not suffer from tunnel vision.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should "minimize the intellectual distance" between the SW and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently.
- Design is not coding.
- The design should be assessed for quality.
- The design should review to minimize conceptual errors.
- External quality factors: Observed by users.
- Internal quality factors: Important to engineers.

6.3 Abstraction

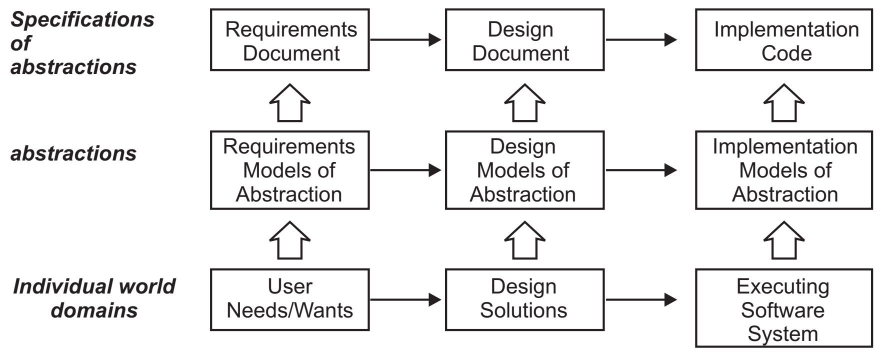
Abstraction is a very influential concept that is used in all engineering disciplines. It is a tool that permits a designer to think a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some armed forces to its environment. An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself.

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated. As we move through different levels of abstraction, we work to create procedural and data abstractions. A procedural abstraction is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word open for a door. Open implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.). A data abstraction is a named collection of data that describes a data object. In the context of the procedural abstraction open, we can define a data abstraction called door. Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door.

Notes

Abstraction in software analysis, design and development is to reduce the complexity to a certain level so that the "relevant" aspects of the requirements, design and development may be easily articulated and understood. This starts with the requirements definition through the actual code implementation. In the three major activities of requirements definition, of design definition, and of implementation code definition, there are different degrees of abstraction based on what is considered relevant and necessary. The general relationships of the individual world domain, the abstractions of those domain entities. The bold, vertical arrows represent the intra-transformations occurring within each individual world domain of requirements, design, and implementation. The horizontal arrows represent the inter-transformations occurring across those domains. In software engineering, we are concerned with both the horizontal and the vertical transformations. (See Figure 6.1)

Figure 6.1: Relationships Among Abstraction, Software Artifacts, and Individual World Domains


Specifications of abstractions
Requirements Document
Design Document
Implementation Code
abstractions
Requirements Models of Abstraction
Design Models of Abstraction
Implementation Models of Abstraction
Individual world domains
User Needs/Wants
Design Solutions
Executing Software System

Abstraction used in the form of a verb, as represented with bold vertical arrows in individual world domain to abstractions, would include the notion of simplification. Simplification represents the concept of categorizing and grouping domain entities into components and relating those components. We simplify by:

(i) Reduction and

(ii) Generalization.

By reduction, we mean the elimination of the details. Generalization, on the other hand, is the identification and specification of common and important characteristics. Through these two specific subtasks of reduction and generalization we carry out the task of abstraction. The process of abstraction, we aim to simplify or decrease the complexity of the domain of software design solution by reducing the details and by generalization. One may view the well established concept of modularization in software engineering as a specific technique within the broader context of abstraction.

The employment of abstraction in software engineering where we aim to reduce complexity is certainly not just limited to the domain of software design. At the early stage of software development, the requirements represent the needs and wants of the users and customers. The user requirement is represented in some form of the user business flow, user functional needs, user information flow, user information representation, user information interface, etc. Each of these categories is an abstraction of the user world. Different degrees of abstraction may be employed Depending on the amount of details that need to be portrayed in the requirements. This intra-transformation is represented by the vertical arrow from user needs/wants domain to requirements models of abstraction.

As we move from requirements towards the development of the solution for the user requirements, a new form of abstraction takes place. The new form of abstraction is necessitated due to the fact that the solution world includes the computing machines and other logical constructs that may not exist in the original user requirements. One of the first artifacts from the solution side is the software architecture and high level design of the software system. At this point, the abstraction, once again, should not include all the details. The inter-transformation of the requirements models of abstraction to the design models of abstraction is shown as the horizontal arrow. Note that the box labelled "Design Models of Abstraction" is the result of two types of transformations:

1. Inter-transformation from the requirements domain and
2. Intra-transformation within the design solution domain.

The "Implementation Models of Abstraction" box is represents the packaging/loading models of the processes, information and control of the mechanical execution of the solution in the form of source code to satisfy the functionalities and the system attributes described in the requirements and design documents. Thus it is a result of the inter-transformations from requirements models of abstraction through the design models of abstractions and the intra-transformations from the actual software execution domain.

The "Implementation Code" box is the specification of this abstraction. The actual execution of the Implementation Code and the interactions with the users formulate the final deployment of the source code abstraction or the "Executing Software System" box. Thus the employment of various abstractions and the transformations of these abstract entities are crucial, integral parts of software engineering.

- Abstraction is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component.
- An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behaviour.
- Abstraction is an indispensable part of the design process and is essential for problem partitioning.

There are two common abstraction mechanisms for software systems:

Functional Abstraction: A module is specified by the function it performs.

For example, a module to compute the log of a value can be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. The decomposition of the system is in terms of functional modules.

Data Abstraction: Any entity in the real world provides some services to the environment to which it belongs. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects.

6.4 Modularity

The genuine power of partitioning comes if a system is partitioned into modules so that the modules are solvable and adjustable unconnectedly. It will be even better if the modules are also separately compliable (then changes in a module will not require recompilation of the whole system). A system is considered modular if it consists of circumspect components so that each component can be implemented separately, and a change to one component has minimal impact on other components. Modularity is a clearly a desirable property in a system. Modularity helps in system debugging—isolating the system problem to a component is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building–a modular system can be easily built by "putting its modules together."

Software architecture embodies modularity, that is, software is divided into unconnectedly named and addressable components, often called modules that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving. Modularity is an important Software Engineering principle. It is a practical application of the principle of "Separation of Concerns" by dividing a complex system into simpler and more manageable modules. The ubiquitous television set is an example of a system made up of a number of modules–speakers, projection tube, channel buttons, volume buttons, etc. Each module has its own defined functionality but when they are put together synergistically, the complete functionalities of a television are realized. Modularization can take place in two ways: composition or decomposition. The composition or bottom-up approach takes modules and puts them together to form a larger system. An application of such an approach is in the assembly of a television set. The composition process begins with the assembling of electronic components such as projection tube, buttons, speakers, control board, etc. into a complete television. The alternative approach is to take a complete system and decompose it into its modules. This approach is known as the decomposition or top-down approach. A car, for example, can be decomposed into a number of sub-systems: Body, fuel system, electrical system, transmission, etc. The electrical system can be further decomposed into modules such as battery, alternator, lights, etc. The Lights module can in turn be decomposed into light bulb, reflector, brackets, etc. This decomposition process repeats itself until all the elementary components are identified.

The measure of inter-module relation is known as coupling. Design goals require modules to have low-coupling and high cohesion. Cohesion is a measure of the inter-relatedness of elements (statements, procedures, declarations) within a module. A module is said to have high cohesion if all the elements in the module are strongly connected with one another. Tight coupling of modules makes analysis, understanding, modification and testing of modules difficult. Reuse of modules is also hindered.

- If a system is partitioned into modules so that the modules are solvable and modifiable separately.
- It will be even better if the modules are also separately compliable.
- A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

6.4.1 Modularity Helps in System

The modularity helps the system in the following way:

- Debugging—Isolating the system problem to a component is easier if the system is modular.
- In system repair—Changing a part of the system is easy as it affects few other parts;
- In system building—A modular system can be easily built by "putting its modules together."
- For modularity, each module needs to support a well defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together.

6.4.2 Module-level Concepts

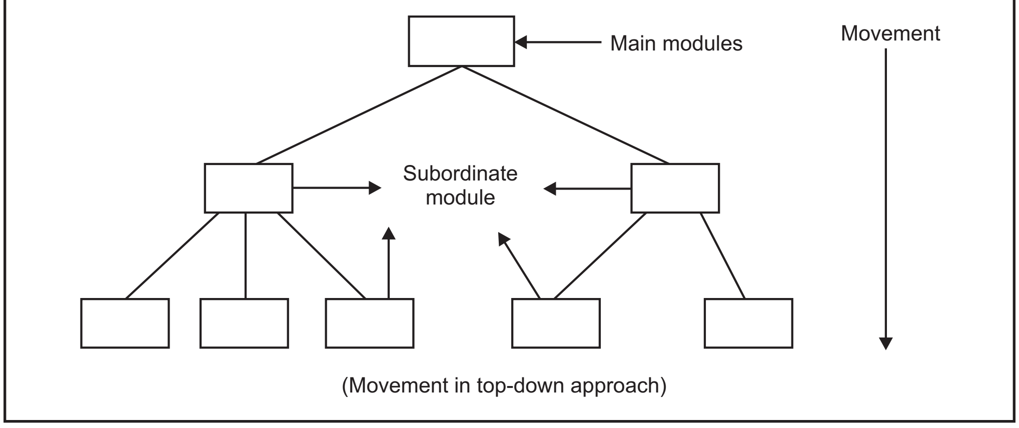
- A module is a logically separable part of a program. It is a program unit that is discreet and identifiable with respect to compiling and loading.
- In terms of common programming language constructs, a module can be a macro, a function, a procedure (or subroutine), a process, or a package.
- In systems using functional abstraction, a module is usually a procedure of function or a collection of these.
- To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately.
- In a system using functional abstraction, coupling and cohesion are two modularization criteria, which are often used together.

6.5 Top-down Approach

A top-down approach (is also known as step-wise design) is essentially the breaking down of a system to increase insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

In this approach testing is conducted from main module to sub module. If the sub module is not developed a temporary program called STUB is used for simulate the sub module.

Figure 6.2: Top-down Approach



Advantages:

(a) Advantageous if major flaws occur toward the top of the program.

(b) Once the I/O functions are added, representation of test cases is easier.

(c) Early skeletal Program allows demonstrations and boosts morale.

Disadvantages:

(a) Stub modules must be produced

(b) Stub Modules are often more complicated than they first appear to be.

(c) Before the I/O functions are added, representation of test cases in stubs can be difficult.

(d) Observation of test output is more difficult.

(e) Allows one to think that design and testing can be overlapped.

(f) Induces one to defer completion of the testing of certain modules.

Did u know? Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth. Mills developed structured programming concepts for practical use and tested them in a 1969 project to automate the New York Times morgue index.

Caution Care should be taken using top-down approach while designing a model, sometimes the test conditions may be impossible, or very difficult, to create.

Self Assessment Questions

1. Software design is a haphazard process.

   (*a*) True (*b*) False

2. ………………is a very powerful concept that is used in all engineering disciplines.

   (*a*) Reduction (*b*) Generalization

   (*c*) Abstraction (*d*) None of these

3. In a ....an overview of the system is formulated, specifying but not detailing any first-level subsystems.

   (*a*) Left-right approach (*b*) right-left approach

   (*c*) bottom-up approach (*d*) top-down approach

4. In a.......................... the individual base elements of the system are first specified in great detail.

   (*a*) Left-right approach (*b*) right-left approach

   (*c*) bottom-up approach (*d*) top-down approach

5. Disadvantages of bottom-up approach:

   (*a*) Driver Modules must be produced

   (*b*) Test conditions are easier to create

   (*c*) Observation of test results is easier

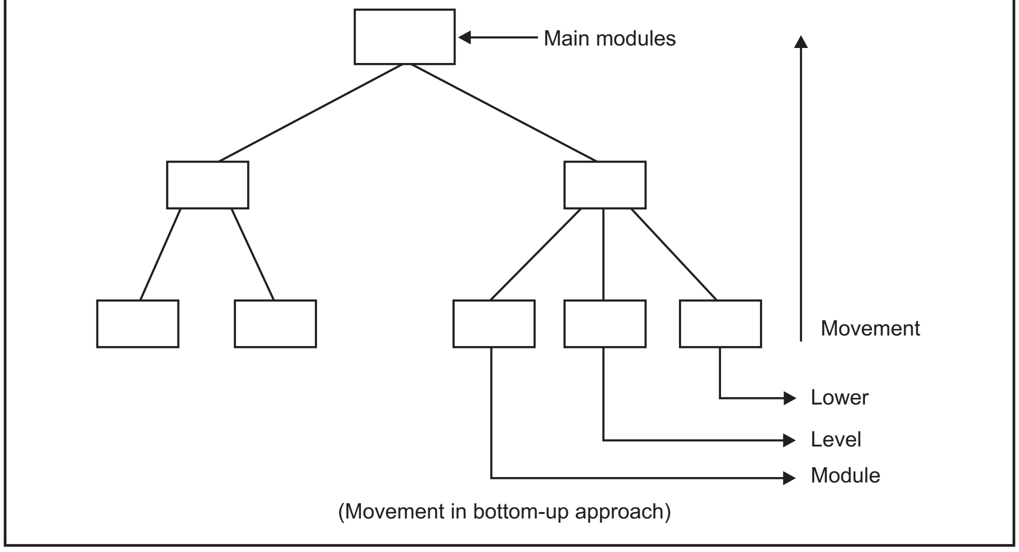
   (*d*) Stub modules must be produced

6.6 Bottom-up Approach

A bottom-up approach is the piecing together of systems to give increase to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose. (See Figure 6.3)

Notes

Figure 6.3: Bottom-up Approach



In this approach testing is conducted from sub module to main module, if the main module is not developed a temporary program called DRIVERS is used to simulate the main module.

Advantages:

1. Advantageous if major flaws occur toward the bottom of the program.
2. Test conditions are easier to create.
3. Observation of test results is easier.

Disadvantages:

1. Driver Modules must be produced.
2. The program as an entity does not exist until the last module is added.

6.6.1 Differentiate between Top-down and Bottom-up Approach

Top-down Approach

1. It starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved.
2. It starts from the highest level component of the hierarchy and proceeds to lower levels.
3. It results in some form of stepwise refinement.
4. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach where no more refinement is needed and the design can be implemented directly.
5. It is suitable only if the specifications are clearly known.
6. It is suitable if a waterfall type of process model is being used.

Bottom-up Approach

1. It starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components.

2. It starts with the lowest level component of the hierarchy and proceeds through progressively higher levels.
3. It works with layers of abstraction.
4. Starting from the very bottom, operations that provide a layer of abstraction are implemented, until the stage is reached where the operations supported by the layer are those desired by the system.
5. It is suitable if a system is to be built from an existing system.
6. It is suitable if an iterative enhancement type of process is being followed.

6.7 Coupling

Couplings are mechanical devices used to attach together two shafts to broadcast power. Coupling can have many functions, but their main purpose is to attach shafts of units that are manufactured unconnectedly and rotating such as motor and generator. They, however, do permit some end movement or misalignment for flexibility and also provide easy disconnection of the two separate devices for repairs or modifications. They also reduce the shock that is transmitted from one shaft to another, protect against overloads and can alter the amount of vibration a rotating unit experiences.

There are many various types of couplings in the industry today. The two most common ones are the rigid and flexible couplings. A rigid coupling has all the functions of a regular coupling alongside with some additional advantages. Using rigid couplings allow for more accurate alignment and a secure hold. With this precise alignment and firm hold, rigid couplings are able to maximize the performance of a machine. Rigid couplings further have two basic types of designs; the sleeve-style and clamped or compression styles couplings. Sleeve-style couplings are basically tubes that have an inner diameter that is the same as the size of the shafts. They can be easily slipped over the shaft and screws can be tightened into it to secure the shaft inside the coupling and ensure that the shaft does not pass all the way through it. The clamped couplings come in two separate parts that can be fitted together onto a shaft. These couplings allow more flexibility as they can be fitted onto shafts that are fixed in place.

In coupling or dependency is the degree to which each program module relies on each one of the other modules. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. The software quality metrics of coupling and cohesion were invented by Larry Constantine, an original developer of Structured Design. Who was also an early proponent of these concepts? Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

6.7.1 Types of Coupling

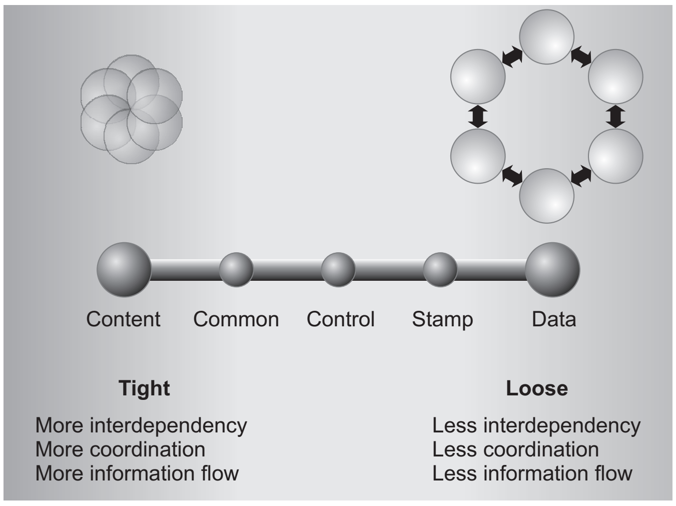
Content Coupling (high): Content coupling (also known as Pathological coupling) is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module. (See Figure 6.4)

Common Coupling: Common coupling (also known as global coupling) is when two modules share the same global data (e.g., a global variable). Changing the shared resource implies changing all the modules using it.

External Coupling: External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

Notes

Figure 6.4: Types of Coupling



Control Coupling: Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

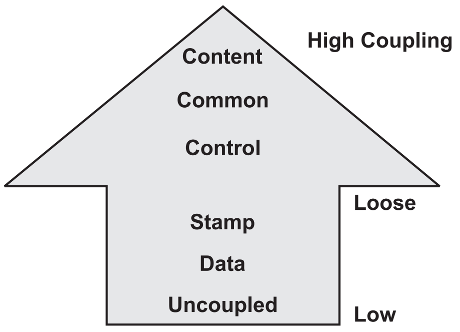
Stamp Coupling (Data-structured coupling): Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it). This may lead to changing the way a module reads a record because a field that the module does not need has been modified.

Data Coupling: Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message Coupling (low): This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing (see Message passing).

No Coupling: Modules do not communicate at all with one another. Conceptual model of Coupling. Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Some types of coupling, in order of highest to lowest coupling, are as follows. (See Figure 6.5)

Figure 6.5: Show the Range of Coupling



6.8 Cohesion

The degree to which all rudiments of a component are heading for towards a single task and all elements directed towards that task are contained in a single component. A Module with cohesion represents how the lightly bound the internal elements of the module are to one another. Whenever the Cohesion is greater, the coupling between the modules is lower. The internal Cohesion of a module is measured in terms of the strength of the hiding of the elements within the modules itself. In computer programming, cohesion is a measure of how well the lines of source code within a module work together to provide a specific piece of functionality. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion" when being discussed. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand. Cohesion is usually contrasted with coupling. High cohesion often correlates with low coupling, and vice versa. The software quality metrics of coupling and cohesion.

6.8.1 Types of Cohesion

There are many different levels of Cohesion are used in several levels of an module.

Coincidental Cohesion: It is one level of cohesion. It occurs when the elements within a given module have no apparent relationship to each other. It occurs when an existing program is making different pieces of modules.

Logical Cohesion: A Module has Logical Cohesion only if there is some Logical relationship between the elements of the module. The elements of the module perform functions which fall in the same logical class.

Temporal Cohesion: This level cohesion is same as logical cohesion, the only different is that the elements are also related in time and are executed at the same time.

Procedural Cohesion: This level Cohesion contains the elements which belong to a common procedural unit. Procedural Cohesion often cuts across functional lines.

Communicational Cohesion: A Module which contains Communicational Cohesion has the elements that are related by a reference to the same Input or Output data.

Sequential Cohesion: This Cohesion occurs only when the output of one element is the input for the next element.

Functional Cohesion: This level Cohesion is the strongest cohesion when compared to other levels. In a functionally bound module, all the elements of the module are related to performing a single function.

6.8.2 Cohesion: degree of dependence among components

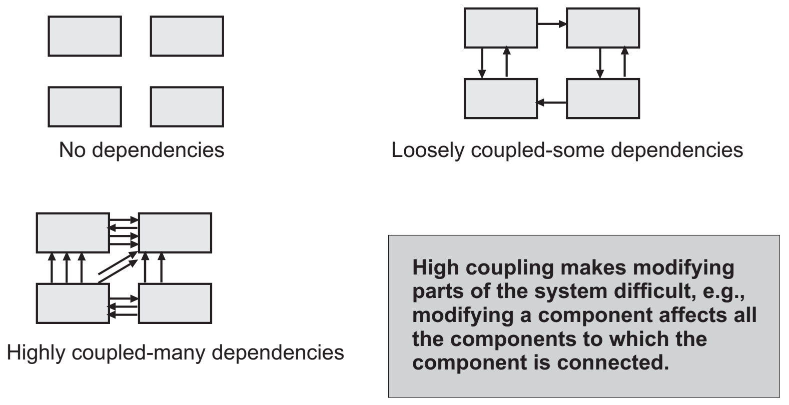
Enumerate different types of coupling and cohesion that may exists between and within the module respectively giving examples.

Notes

Following are types of coupling that may exist between the modules:

1. ***Data Coupling***: The dependency between module A and module B is said to be data coupled if their dependency is based on the fact that they communicate only by passing of data between the two, e.g., an integer, a float, a character, etc. (See Figure 6.6)

Figure 6.6: Cohesion Degree of Dependence Among Components


No dependencies
Loosely coupled-some dependencies
Highly coupled-many dependencies
High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

2. ***Stamp Coupling***: Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
3. ***Control Coupling***: Module A and module B is said to be controlled coupled, if they communicate by passing of control information.
4. ***External Coupling***: The form of coupling in which a module has dependency to other module external to the software being developed. Example of this coupling is various networking or external devices.
5. ***Common Coupling***: With common coupling, module A and module B share global data area and are commonly found in programming language.
6. ***Content Coupling***: Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another module.

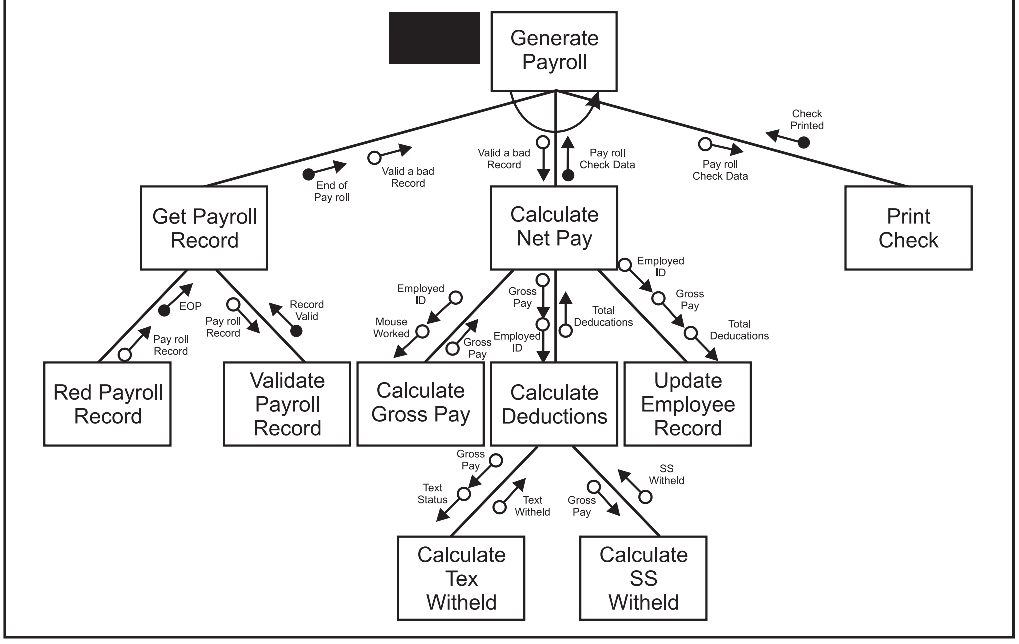
Task Draw the structure of coupling.

6.9 Structure Charts

A Structure Chart (SC) in software engineering and organizational theory is a chart which shows the stop working of a system to its lowest manageable levels. They are used in structured programming to position program modules into a tree. Each module is represented by a box, which contains the module's name. The tree structure visualizes the relationships between modules. Functional modeling is a system engineering technique that requires the graphical design and presentation of complex business models. This visualization design approach provides an elegant method for describing information to a non-technical audience. The functional model decomposes the detail interfaces of a software application into clearly defined components. This chart typically consists of shapes with descriptions and connecting lines that show relationships to other shapes within the chart. Project managers rely on structure charts for managing their daily activities. The most widely used structure chart for project management is the work breakdown structure chart.

Structured Chart Example

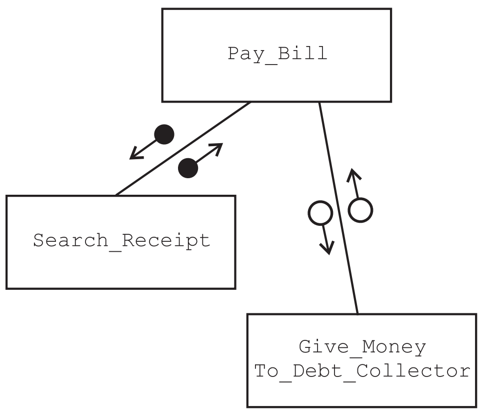
Figure 6.7: Structured Chart



A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. The lines represent the connection and or ownership between activities and sub activities as they are used in organization. (See Figure 6.7)

Sample of Hierarchy

Figure 6.8: Structured Chart of Hierarchy



- The size and complexity of the system, and
- Number of readily identifiable functions and modules within each function and
- Whether each identifiable function is a manageable entity or should be broken down into smaller components. (See Figure 6.8)

Notes

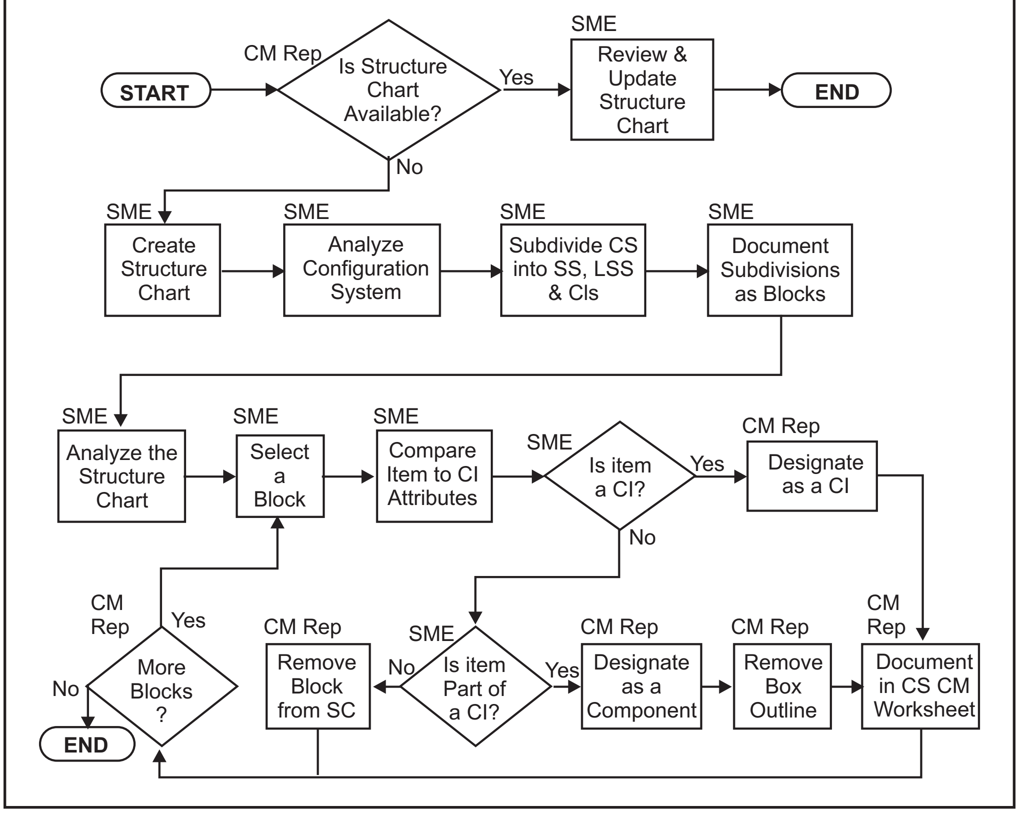
A structure chart is also used to diagram associated elements that comprise a run stream or thread. It is often developed as hierarchical, but other representations are allowable. The representation must describe the breakdown of the configuration system into subsystems and the lowest manageable level. An accurate and complete structure chart is the key to the determination of the configuration items, and a visual representation of the configuration system and the internal interfaces among its CIs. During the configuration control process, the structure chart is used to identify CIs and their associated artifacts that a proposed change may impact.

Structure chart construction

A structure chart can be developed starting with the creating of a structure, which places the root of an upside-down tree which forms the structure chart. The next step is to conceptualize the main sub-tasks that must be performed by the program to solve the problem. Next, the programmer focuses on each sub-task individually, and conceptualizes how each can be broken down into even smaller tasks. Eventually, the program is broken down to a point where the leaves of the tree represent simple methods that can be coded with just a few program statements.

In practice, first it is checked if a Structure Chart has been developed already. If so an expert needs to review it to ensure it represents the current structure and if not, updates the chart where needed. (See Figure 6.9)

Figure 6.9: Structure Chart Construction



6.10 Data Flow Diagrams

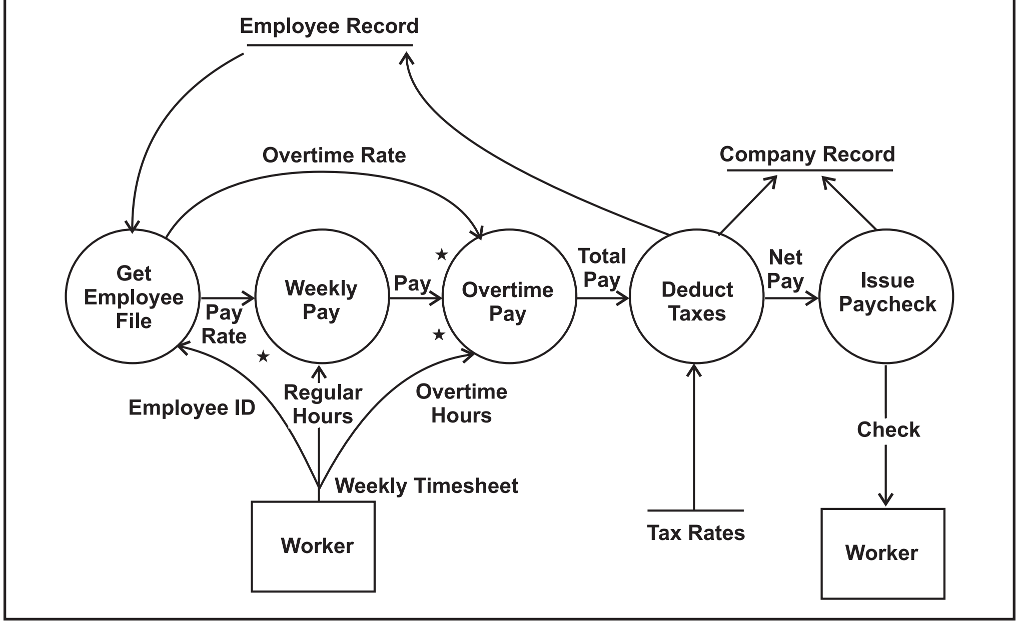
Data flow diagrams (also called data flow graphs) are usually used during problem analysis. Data flow diagrams (DFDs) are quite general and are not incomplete to problem analysis for software requirements specification. They were in use long before the software engineering discipline began. DFDs are very useful in understanding a system and can be effectively used during analysis.

A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs. Any complex system will not perform this transformation in a "single step," and data will typically undergo a series of transformations before it becomes the output. The DFD aims to capture the transformations that take place within a system to the input data so that eventually the output data is produced. The agent that performs the transformation of data from one state to another is called a process (or a bubble). Thus, a DFD shows the movement of data through the different transformations or processes in the system. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the bubbles. A rectangle represents a source or sinks and is a net originator or consumer of data. A source or a sink is typically outside the main system of study.

In this DFD there is one basic input data flow, the weekly timesheet, which originates from the source worker. The basic output is the paycheck, the sink for which is also the worker. In this system, first the employee's record is retrieved, using the employee ID, which is contained in the timesheet. From the employee record, the rate of payment and overtime are obtained. These rates and the regular and overtime hours (from the timesheet) are used to compute the pay. After the total pay is determined, taxes are deducted. To compute the tax deduction, information from the tax-rate file is used. The amount of tax deducted is recorded in the employee and company records. Finally, the paycheck is issued for the net pay. The amount paid is also recorded in company records.

Some conventions used in drawing this DFD should be explained. All external files such as employee record, company record, and tax rates are shown as a labelled straight line. The need for multiple data flows by a process is represented by a "*" between the data flows. This symbol represents the AND relationship. For example, if there is a "*" between the two input data flows A and B for a process, it means that A AND B are needed for the process. In the DFD, for the process "weekly pay" the data flow "hours" and "pay rate" both are needed, as shown in the DFD. Similarly, the OR relationship is represented by a "+" between the data flows. (See Figure 6.10)

Figure 6.10: Data Flow Diagrams



It is necessary that some decomposition and abstraction mechanism be used for such systems. DFDs can be hierarchically organized, which helps in progressively partitioning and analyzing large systems. Such DFDs together are called a leveled DFD set.

A leveled DFD set has a starting DFD, which is a very abstract representation of the system, identifying the major inputs and outputs and the major processes in the system. Often, before the initial DFD, a context diagram may be drawn in which the entire system is shown as a single process.

This DFD is an abstract description of the system for handling payment. It does not matter if the system is automated or manual. This diagram could very well be for a manual system where the computations are all done with calculators, and the records are physical folders and ledgers. The details and minor data paths are not represented in this DFD. For example, what happens if there are errors in the weekly timesheet is not shown in this DFD. This is done to avoid getting bogged down with details while constructing a DFD for the overall system. If more details are desired, the DFD can be further refined. It should be pointed out that a DFD is not a flowchart. A DFD Many system represents the flow of data, while a flowchart shows the flow of control. A DFD does not represent procedural information. So, while drawing a DFD, one must not get involved in procedural details, and procedural thinking must be consciously avoided. For example, considerations of loops and decisions must be ignored. In drawing the DFD, the designer has to specify the major transforms in the path of the data flowing from the input to output. How those transforms are performed is not an issue while drawing the data flow graph.

All its inputs, outputs, sinks, and sources. Then each process is refined and a DFD is drawn for the process. In other words, a bubble in a DFD is expanded into a DFD during refinement. For the hierarchy to be consistent, it is important that the net inputs and outputs of a DFD for a process are the same as the inputs and outputs of the process in the high-level DFD. This refinement stops if each bubble is considered to be "atomic," in that each bubble can be easily specified or understood. It should be pointed out that during refinement, though the net input and output are preserved, a refinement of the data might also occur. That is, a unit of data may be broken into its components for processing when the detailed DFD for a process is being drawn. So, as the processes are decomposed, data decomposition also occurs. In a DFD, data flows are identified by unique names. These names are chosen so that they convey some meaning about what the data is. However, for specifying the precise structure of data flows, a data dictionary is often used. The associated data dictionary states precisely the structure of each data flow in the DFD. To define the data structure, a regular expression type notation is used. While specifying the structure of a data item, sequence or composition is represented by "+", selection by vertical bar " | " (means one OR the other), and repetition by "*".

Did u know? Data flow diagrams were proposed by Larry Constantine, the original developer of structured design, based on Martin and Estrin's "data flow graph" model of computation.

6.11 Design Heuristics

The design steps mentioned earlier do not reduce the design process to a series of steps that can be followed blindly. The strategy requires the designer to exercise sound judgment and common sense. The basic objective is to make the program structure reflect the problem as closely as possible. With this in mind the structure obtained by the methodology described earlier should be treated as an initial structure, which may need to be modified. Here we mention some heuristics that can be used to modify the structure, if necessary. Keep in mind that these are merely pointers to help the designer decide how the structure can be modified. The designer is still the final judge of whether a particular heuristic is useful for a particular application or not. Module size is often considered an indication of module complexity. In terms of the structure of the system, modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function. On the other hand, modules that are too small may not require any additional identity and can be combined with other modules.

However, the decision to split a module or combine different modules should not be based on size alone. Cohesion and coupling of modules should be the primary guiding factors. A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules does not increase. Similarly, two or more modules should be combined only if the resulting module has a high degree of cohesion and the coupling of the resulting module is not greater than the coupling of the sub modules. Furthermore, a module usually should not be split or combined with another module if it is subordinate to many different modules. As a rule of thumb, the designer should take a hard look at modules that will be larger than about 100 lines of source code or will be less than a couple of lines. Another parameter that can be considered while "fine-tuning" the structure is the fan-in and fan-out of modules. Fan-in of a module is the number of arrows coming in the module, indicating the number of super ordinates of a module. Fan-out of a module is the number of arrows going out of that module, indicating the number of subordinates of the module. A very high fan-out is not very desirable, as it means that the module has to control and coordinate too many modules and may therefore be too complex. Fan-out can be reduced by creating a subordinate and making many of the current subordinates subordinate to the newly created module. In general the fan-out should not be increased above five or six. Whenever possible, the fan-in should be maximized. Of course, this should not be obtained at the cost of increasing the coupling or decreasing the cohesion of modules. For example, implementing different functions into a single module, simply to increase the fan-in, is not a good idea. Fan-in can often be increased by separating out common functions from different modules and creating a module to implement that function. Another important factor that should be considered is the correlation of the scope of effect and scope of control. The scope of effect of a decision (in a module) is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision. The scope of control of a module is the module itself and all its subordinates (not just the immediate subordinates). The system is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which the decision is located. Ideally, the scope of effect should be limited to the modules that are immediate subordinates of the module in which the decision is located. Violation of this rule of thumb often results in more coupling between modules.

There are some methods that a designer can use to ensure that the scope of effect of a decision is within the scope of control of the module. The decision can be removed from the module and "moved up" in the structure. Alternatively, modules that are in the scope of effect but are not in the scope of control can be moved down the hierarchy so that they fall within the scope of control.

A heuristic evaluation is a discount usability inspection method for computer software that helps to identify usability problems in the user interface (UI) design. It specifically involves evaluators examining the interface and judging its compliance with recognized usability principles (the "heuristics"). These evaluation methods are now widely taught and practiced in the New Media sector, where UIs are often designed in a short space of time on a budget that may restrict the amount of money available to provide for other types of interface testing.

The heuristic to solve the problem and software engineering is not an exception. In the past also we are seeing the use of some metric based heuristics in design and development of the software. For example, if the number of parameters in a function is more than five gives impression that module may not be having function cohesion. The heuristics are not written as hard and fast rules; they are meant to serve as warning mechanisms which allow the flexibility of ignoring the heuristic as necessary.

The main goal of heuristic evaluations is to identify any problems associated with the design of user interfaces. Heuristic evaluations are one of the most informal methods. of usability inspection in the field of human-computer interaction. There are many sets of usability design heuristics; they are not mutually exclusive and cover many of the same aspects of user interface design. Quite often, usability problems that are discovered are categorized—often on a numeric scale to their estimated impact on user performance or acceptance. Often the heuristic evaluation is conducted in the context of use cases (typical user tasks), to provide feedback to the developers on the extent to which the interface is likely to be compatible with the intended users' needs and preferences.

The simplicity of heuristic evaluation is beneficial at the early stages of design. This usability inspection method does not require user testing which can be burdensome due to the need for users, a place to test them and a payment for their time. Heuristic evaluation requires only one expert, reducing the complexity and expended time for evaluation. Most heuristic evaluations can be accomplished in a matter of days. The time required varies with the size of the artifact, its complexity, the purpose of the review, the nature of the usability issues that arise in the review, and the competence of the reviewers. Using heuristic evaluation prior to user testing will reduce the number and severity of design errors discovered by users. Although heuristic evaluation can uncover many major usability issues in a short period of time, a criticism that is often levelled is that results are highly influenced by the knowledge of the expert reviewer(s). This "one-sided" review repeatedly has different results than software performance testing, each type of testing uncovering a different set of problems.

6.11.1 Most-used usability heuristics for user interface design

Visibility of system status: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world: The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom: Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards: Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention: Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall: Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use: Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Help users recognize, diagnose, and recover from errors: Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Notes

Case Study

Java Card Security Testing

In an effort to enhance payment cards with new functionality—such as the ability to provide secure cardholder identification or remember personal references—many credit-card companies are turning to multi-application smart cards. These cards use resident software applications to process and store thousands of times more information than traditional magnetic-stripe cards.

Security and fraud issues are critical concerns for the financial institutions and merchants spearheading smart-card adoption. By developing and deploying smart-card technology, credit-card companies provide important new tools in the effort to lower fraud and abuse. For instance, smart cards typically use a sophisticated crypto system to authenticate transactions and verify the identities of the cardholder and issuing bank. However, protecting against fraud and maintaining security and privacy are both very complex problems because of the rapidly evolving nature of smart-card technology.

The security community has been involved in security risk analysis and mitigation for Open Platform (now known as Global Platform, or GP) and Java Card since early 1997. Because product security is an essential aspect of credit-card companies' brand protection regimen, companies like Visa and MasterCard spend plenty of time and effort on security testing and risk analysis. One central finding emphasizes the importance of testing particular vendor implementations according to our two testing categories: adherence to functional security design and proper behaviour under particular attacks motivated by security risks.

The latter category, adversarial security testing (linked directly to risk analysis findings), ensures that cards can perform securely in the field even when under attack. Risk analysis results can be used to guide manual security testing. As an example, consider the risk that, as designed, the object-sharing mechanism in Java Card is complex and thus is likely to suffer from security-critical implementation errors on any given manufacturer's card. Testing for this sort of risk involves creating and manipulating stored objects where sharing is involved. Given a technical description of this risk, building specific probing tests is possible

Automating Security Testing

Over the years, Cigital has been involved in several projects that have identified architectural risks in the GP/Java Card platform, suggested several design improvements, and designed and built automated security tests for final products (each of which has multiple vendors).

Several years ago, we began developing an automated security test framework for GP cards built on Java Card 2.1.1 and based on extensive risk analysis results. The end result is a sophisticated test framework that runs with minimal human intervention and results in a qualitative security testing analysis of a sample smart card. This automated framework is now in use at MasterCard and the U.S. National Security Agency.

The first test set, the functional security test suite, directly probes low-level card security functionality. It includes automated testing of class codes, available commands, and crypto functionality. This test suite also actively probes for inappropriate card behavior of the sort that can lead to security compromise.

The second test set, the hostile applet test suite, is a sophisticated set of intentionally hostile Java Card applets designed to probe high-risk aspects of the GP on a Java Card implementation.

Contd...

Notes

Results: Non-functional Security Testing Is Essential

Most cards tested with the automated test framework (but not all) pass all functional security tests, which we expect because smart-card vendors are diligent with functional testing (including security functionality). Because smart cards are complex embedded devices, vendors realize that exactly meeting functional requirements is an absolute necessity for customers to accept the cards. After all, they must perform properly worldwide.

However, every card submitted to the risk-based testing paradigm exhibited some manner of failure when tested with the hostile applet suite. Some failures pointed directly to critical security vulnerabilities on the card; others were less specific and required further exploration to determine the card's true security posture.

As an example, consider that risk analysis of Java Card's design documents indicates that proper implementation of atomic transaction processing is critical for maintaining a secure card. Java Card has the capability of defining transaction boundaries to ensure that if a transaction fails, data roll back to a pre-transaction state. In the event that transaction processing fails, transactions can go into any number of possible states, depending on what the applet was attempting. In the case of a stored-value card, bad transaction processing could allow an attacker to print money by forcing the card to roll back value counters while actually purchasing goods or services. This is called a "torn transaction" attack in credit-card risk lingo.

When creating risk-based tests to probe transaction processing, we directly exercised transaction-processing error handling by simulating an attacker attempting to violate a transaction—specifically, transactions were aborted or never committed, transaction buffers were completely filled, and transactions were nested (a no-no according to the Java Card specification). These tests were not based strictly on the card's functionality—instead, security test engineers intentionally created them, thinking like an attacker given the results of a risk analysis.

Several real-world cards failed subsets of the transaction tests. The vulnerabilities discovered as a result of these tests would allow an attacker to terminate a transaction in a potentially advantageous manner—a critical test failure that would not have been uncovered under normal functional security testing. Fielding cards with these vulnerabilities would allow an attacker to execute successful attacks on live cards issued to the public. Because of proper risk-based security testing, the vendors were notified of the problems and corrected the code responsible before release.

Questions

1. Explain the automating security testing.
2. Describes the term non-functional security testing is essential.

Self Assessment Questions

6. The worst type of coupling is……………..
   (*a*) data coupling (*b*) control coupling
   (*c*) stamp coupling (*d*) content coupling.
7. The desired level of coupling is ……………..
   (*a*) no coupling (*b*) control coupling
   (*c*) common coupling (*d*) data coupling.

8. Pseudo code can replace ………………..

   (*a*) flowcharts (*b*) structure charts

   (*c*) decision tables (*d*) cause-effect graphs

9. The inter-transformation of the requirements models of abstraction to the design models of abstraction is shown as the …………...

   (*a*) horizontal arrow (*b*) vertical arrow

   (*c*) Both of these (*d*) None of these.

10. A top-down model is often specified with the assistance of "…………….", these make it easier to manipulate.

    (*a*) green boxes (*b*) black boxes

    (*c*) white boxes (*d*) None of these.

11. In this approach testing is conducted from …………………..

    (*a*) sub module to main module (*b*) main module to sub module

    (*c*) Both of these (*d*) None of these.

12. The processes are shown by named circles and data flows are represented by named arrows entering or leaving the ………………

    (a) square (*b*) bubbles

    (c) rectangle (*d*) None of these.

13. A Structure Chart (SC) in software engineering and organizational theory, is a chart which shows the …………. of a system to its lowest manageable levels.

    (*a*) step-down (*b*) up-down

    (*c*) break-down (*d*) None of these.

14. Abstraction in software engineering where we aim to reduce complexity is certainly not just limited to the domain of software design.

    (*a*) True (*b*) False

15. The heuristics are written as hard and fast rules.

    (*a*) True (*b*) False

6.12 Summary

- A functional design assures that each modular part of a device has only one responsibility and performs that responsibility with the minimum of side effects on other parts.
- At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution.
- Abstraction in software analysis, design and development is to reduce the complexity to a certain level so that the "relevant" aspects of the requirements, design and development may be easily articulated and understood.
- Modularity, that is, software is divided into separately named and addressable components, often called modules that are integrated to satisfy problem requirements.

- A top-down approach (is also known as step-wise design) is essentially the breaking down of a system to gain insight into its compositional sub-systems.
- A bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system.
- The software quality metrics of coupling and cohesion were invented by Larry Constantine, an original developer of Structured Design.
- Functional modelling is a system engineering technique that requires the graphical design and presentation of complex business models.
- A DFD shows the flow of data through a system. It views a system as a function that transforms the inputs into desired outputs.
- Heuristic evaluation requires only one expert, reducing the complexity and expended time for evaluation.

6.13 Keywords

Domain: A domain name is an identification string that defines a realm of administrative autonomy, authority, or control on the Internet. Domain names are formed by the rules and procedures of the Domain Name System (DNS).

Flag: Any variable or constant that holds data can be used as a flag. You can think of the storage location as a flag pole. The value stored within the variable conveys some meaning and you can think of it as being the flag.

Modular: The basic idea underlying modular design is to organize a complex system (such as a large program, an electronic circuit, or a mechanical device) as a set of distinct components that can be developed independently and then plugged together. Preservation measures in relation to software, both at the development stage and

Preserved: Part of this involved examining the purpose and benefits of employing. Retrospectively to legacy software.

Thread: The Thread Disk for Windows engineering software calculates critical dimensions for threads based on ANSI tables and formulas.

Use-case: A use case in software engineering is a description of a system's behaviour as it responds to a request that originates from outside of that system.

Lab Exercise Draw the flow chart of top-down and bottom-up approach.

Draw the structure of data flow diagrams.

6.14 Review Questions

1. Define functional design and explain the principles of functional design.
1. What is the difference between top-down and bottom-up approach?
2. Write short notes on

   (*i*) Cohesion (*ii*) Heuristics
3. Write a short note on structure chart.

4. Explain the following with the help of an example
   (*i*) Common coupling
   (*ii*) Communicational cohesion
   (*iii*) Structure chart
5. Define coupling. Discuss various types of coupling.
6. Explain the concept of bottom-up and top-down design.
7. Define cohesion and coupling? Give suitable examples.
8. What is the modularity? And explain the module-level.
9. What is the structure chart? And explain the structured chart example.
10. Explain the data flow diagrams?

Answers for Self Assessment Questions

1.	(*b*)	2.	(*c*)	3.	(*d*)	4.	(*c*)	5.	(*a*)
6.	(*d*)	7.	(*d*)	8.	(*a*)	9.	(*a*)	10.	(*b*)
11.	(*a*)	12.	(*b*)	13.	(*c*)	14.	(*a*)	15.	(*b*)

6.15 Further Readings

Books

An Integrated Approach to Software Engineering, by P. Jalote

Software Engineering for Large *Software* Systems, by B. A. Kitchenham

Online link

http://books.google.co.in/books?id=PA28&dq=functional+oriented+design
+in+software+engg.

Unit 7: Introduction to Verification

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the verification
- Define metrics
- Discuss the network, stability, and information flow

Introduction

Software Verification and Validation (V&V) is a systems engineering regulation serving a development organization construct quality into the application software throughout the software life cycle. Validation is worried with checking that the software meets the user's needs, and Verification is concerned with checking that the system is well-engineered. The definition of activities included under software V&V is necessarily quite broad, and includes both technical and management-based activities.

The Federal approach to V&V differs somewhat from the international standard for software V&V, namely that found in the Institute of Electrical and Electronic Engineers (IEEE) Standard for Software Verification and Validation Contrary to the international standard, Federal V&V does not require a continuous on-site presence or extensive testing, nor does it perform actual quality assurance activities or other remediation's. It instead imposes periodic reviews of software development projects that include site visits employing various industry standards to conduct artefact analysis with interviews of a project's team and stakeholder in order to fashion a comprehensive "snapshot" of a project's management and technical processes at work at a given point-in-time. Another distinction the reader will note is in the inclusion of the word "Independent" in front of Verification and Validation (V&V). In other words, Independent V&V is the set of verification and validation activities performed by an agency not under the control of the organization that is developing the software. Services must be provided and managed by an organization that is technically and managerially independent of the subject software development project. This independence takes two mandatory forms. First, technical independence requires that the services provider not organizationally be or have been, nor use personnel who are or were, involved in the software development or implementation effort, or for that matter participated in the project's initial planning and/or subsequent design. Such technical independence helps ensure every review report is free of personal or professional bias, posturing, or gold plating. Secondly, managerial independence is required of the services provider to ensure that the effort is vested in an organization departmentally and hierarchically separate from the software development and program management organizations. Such managerial independence helps ensure that the service provider is able to deliver to both State and Federal executive leadership and management, findings and recommendations of an review without restriction, fear of retaliation, or coercion.

The terms verification and validation are related to software quality checking. Very often we have heard various comments and debates concerning the question if static analysis of program source code should be referred to verification and validation and in what way these notions differ. On a whole, it seems that everyone means something different by these terms and it leads to a mutual misunderstanding.

We decided to make it clear in order to stick to the most proper understanding of these notions. "Software verification methods" it gives a thorough description of the terms and we decided to further rely upon the definitions provided in this work. Here are some extracts from it related to verification and validation.

Verification and validation are the types of activity intended for checking the quality of software and detecting errors in it. Having the same goal, they differ in the origin of the properties, rules and limitations (whose violation is considered an error) being tested during these processes.

The term can be defined as "an artefact of software life-cycle". By artefacts of software life-cycle we appreciate various information entities, documents and models created or used during the process of software development and maintenance. Thus, requirements specification, architecture description, domain model in some graphics language, source code, user documentation etc. are artefacts. Various models used by single developers during software creation and analysis but not fixed in the form of documents available to other people are not considered artefacts.

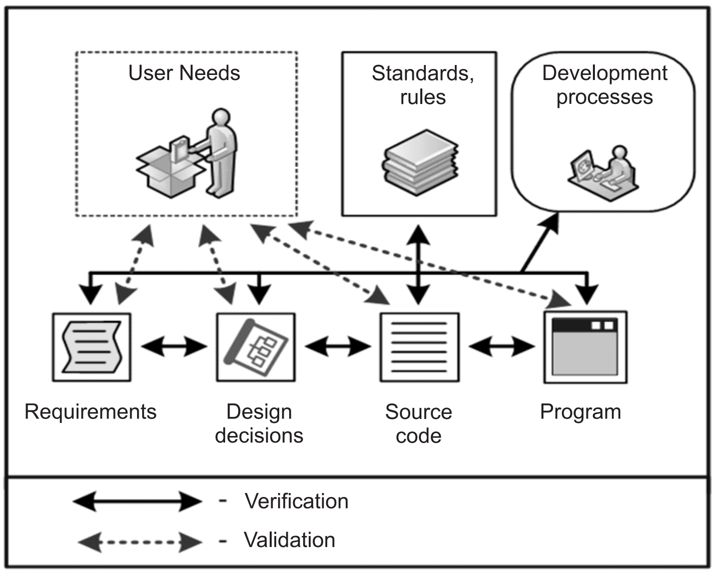
Verification checks if some artefacts being created while developing and maintaining software correspond to some other artefacts created earlier or being used as source data and also if these artefacts and processes of their development correspond to the rules and standards. In particular, verification checks if standards, software requirements specification, design decisions, source code, user documentation and operation of the software itself correspond to each other. Besides, it checks if the requirements, design decisions, documentation and the code are arranged in correspondence with the software development rules and standards accepted in a particular country, branch and organization, and also if all the operations stated in the standards are executed and in the appropriate sequence. Defects and errors detected during verification are divergences between some of the listed documents, between documents and actual program operation or between standards and actual processes of software development and maintenance. Making a decision what document must be corrected (if not both of them) is a separate task.

Validation checks if any artefacts being created or used in the process of software development and maintenance correspond to the needs of the users and customers of this software, with taking into account the laws of the domain and limitations of software usage context. These needs are usually not fixed in the form of a document - when they are, they turn into a requirements specification, i.e. one of the artefacts of the software development process. That is why validation is a less formal activity than verification. It is always carried out in the presence of customers', users', business-analysts' or domain experts' representatives - those whose opinion can be considered a rather good expression of the needs of users, customers and other persons concerned. The methods of its execution often involve specific techniques of discovering knowledge and actual needs of the participants.

Notes

The difference between verification and validation (V&V) is illustrated in Figure 7.1.

Figure 7.1: Relation between Verification and Validation



The definitions given above are drawn through some extension of the definitions of IEEE 1012 standard on the processes of verification and validation. In the standard software engineering glossary IEEE 610.12 of 1990, the definition of verification is nearly the same but the definition of validation is different - it is said there that validation must check if the result software corresponds to the requirements set to it in the beginning of the development process. In this case, validation would be an instance of verification but it is not mentioned anywhere in literature on software engineering, and this is the reason, together with the fact that this definition was corrected in IEEE 1012 of 2004, that it should be considered incorrect. This phrase by B. Boehm which is used very frequently.

Verification answers the question "Are we making the product properly?" while validation - "Are we making a proper product?" also contributes to the confusion because its aphoristic character, unfortunately, combines with ambiguity. But multiple works of allow us to think that he meant by verification and validation nearly the same notions that were defined. The described discrepancies can be traced in the content of software engineering standards. Thus, ISO standards consider testing a kind of validation but not verification and it seems to result from sticking to the incorrect definition of the standard glossary.

7.1 Meaning of Verification

Verification is the process of difficult like set of documents, plans, specifications and requirements. The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase. This is an preliminary report on the use of model-based verification techniques within software development and upgrade practices. It presents the specific activities and everyday job that are required of engineers who use the model-based confirmation paradigm and describes proposed approaches for integrating model-based verification into an organization's software engineering practices. The approaches outlined in this report are preliminary concepts for the integration of model building and analysis techniques into software engineering review and inspection practices. These techniques are presented as both practices within peer review processes and as autonomous engineering investigations. The objective of this report is to provide a starting point for the use of model-based verification techniques and a framework for their evaluation in real-world applications. It is expected that the results of pilot studies that employ the preliminary approaches described here will form the basis for improving the practices themselves and software verification generally. Verification is the process of complaining a work product with its parent specification or a standard for the purpose of detecting errors. Thus design is verified by comparing it with the requirements and code is verified by comparing it with the design. There are four basic methods of verification: inspection which involves the visual comparison of two items, test which is an exactly reproducible method of comparing expected behaviour to actual behaviour, demonstration which is like testing but not exactly reproducible, and analysis which is the application of statistical methods to processes that do not produce repeatable results.

The terms Verification and Validation have different meaning depending on the discipline in engineering or quality management systems. It is the act of reviewing, inspecting or testing, in order to establish and document that a product, service or system meets regulatory or technical standards. According to the regulatory body FDA Verification is defined in Quality System Regulation as confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.

Benefit

The Verification and Validation subprogram provides methods and measures necessary to assess the credibility of the ASC codes and models, quantify uncertainties in ASC calculation results, measure the progress in the ASC predictive capabilities, and provide confidence when applying simulations for stockpile deliverables.

V&V Methods

The area provides methods and measures necessary to assess the credibility of the ASC codes and models, quantify uncertainties in ASC calculation results, and measure the progress in the ASC predictive capabilities. In this role, V&V will be aware of leading research, perform its own research, and be an advocate for advanced research and methods development in the areas of code verification, solution verification, validation metrics and methodology, and uncertainty quantification (UQ) as enabling technologies for validation and quantification of margins and uncertainties (QMU) in a risk-informed decision context.

V&V Assessments

The area delivers science-based assessments of the predictive capability and uncertainties in ASC integrated performance, engineering, and specialized codes' phenomenological models, numerical methods, and related models, to support the needs of the Stockpile Stewardship Program. This area focuses on establishing credibility in integrated simulation capabilities by collecting evidence that the numerical methods and simulation models are being solved correctly, and whether the simulation results from the mathematical and computational models implemented into the codes agree with real-world observations. This requires extensive collaboration with the various ASC elements, DSW, and the Science and Engineering Campaigns.

Data Validation, Archiving, SQA, and Training

The area provides traceable and reproducible work products and processes for stockpile certification (short and long term), as well as foundational elements for establishing software quality standards and training weapons scientists in the application of verification, validation, and UQ methods. The scope of this product includes integral validation of physical property data that are used as inputs for various weapon relevant simulations. Additionally, this product includes work product and data archiving and simulation pedigree tracking. It also includes establishing high-level software quality requirements, assessment techniques and methods, and development of Software Quality Engineering (SQE) tools. Finally, it supports the adoption

Notes

of stockpile QMU assessment methodologies through computational simulation by providing training for use of V&V and UQ tools to establish credible simulation-based performance margin and uncertainty estimates.

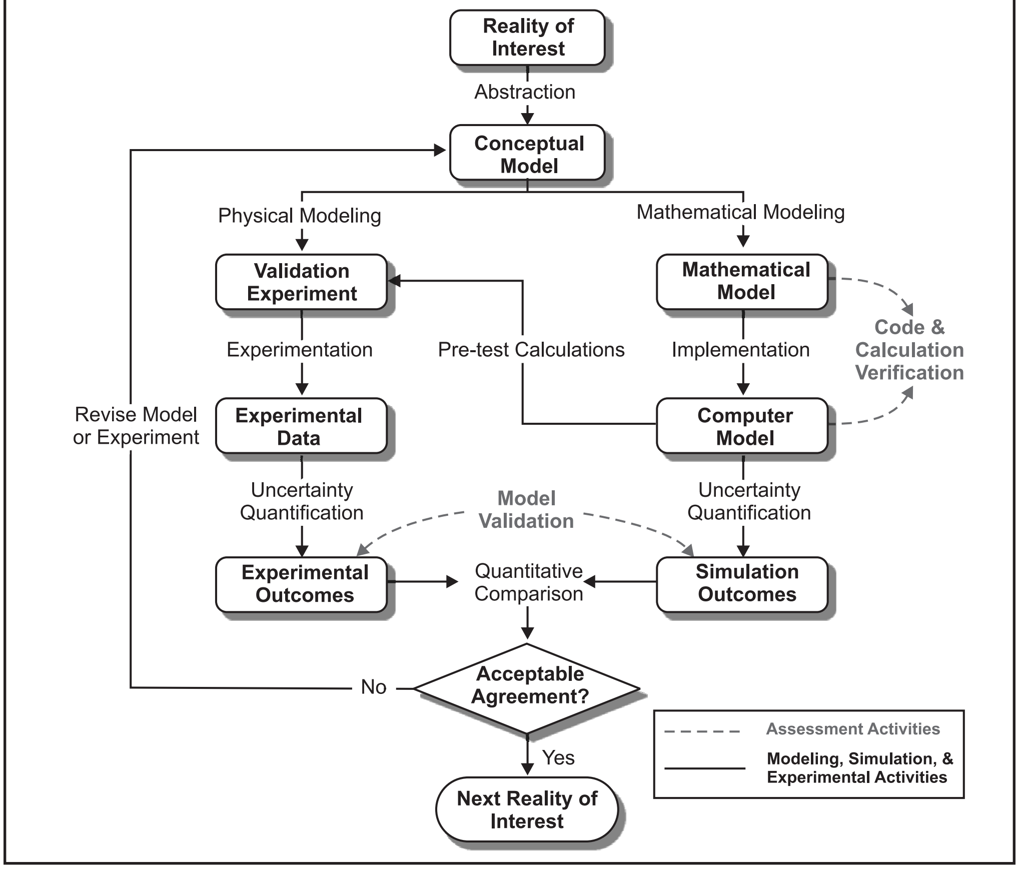
Advantages and Disadvantages of Verification

The Verification and Validation model commonly known as V-Model is considered to be an extension of the Waterfall model. This is because just like the waterfall model, it is a well structured method in which the different phases progress in a sequential or linear way. That means each phase begins only after the completion of the phase.

An important aspect of this model is that testing activities like planning, test designing happens well before coding. The advantage is that it saves ample amount of time and since the testing team is involved early on, they develop a very good understanding of the project at the very beginning.

The biggest disadvantage of V-model is that it is very rigid and the least flexible. If any changes happen mid way, not only the requirements documents but also the test documentation needs to be updated. Even with these disadvantages, it is the most favoured software development process as it is simple and easy to use. (See Figure 7.2)

Figure 7.2: Verification



Did u know? Baffle Ball, the first mechanical pinball game, was advertised as being "free of bugs" in 1931.

Self Assessment Questions

1. The terms verification and validation have different meaning depending on the discipline in engineering or quality management systems.

   (*a*) True (*b*) False

2. .... are established during the requirements phase of the conceptual model development and incorporate numerical and experimental uncertainty.

   (*a*) verification metric (*b*) validation metric

   (*c*) verification and validation metric (*d*) None of these

3. .... are established during the requirements phase of the conceptual model development and incorporate numerical and experimental uncertainty.

   (*a*) Network metrics (*b*) Validation metric

   (*c*) Verification and validation metric (*d*) Verification metric

4. The Verification and Validation model commonly known as V Model is considered to be an extension of the......................

   (*a*) phenomenological models (*b*) domain model

   (*c*) computational models (*d*) Waterfall model

5. The biggest disadvantage of V-model is that it is not very rigid and the least flexible.

   (*a*) True (*b*) False

7.2 Meaning of Metrics

Metric of attention is complexity, as one of our goals is to strive for straightforwardness and ease of understanding. A possible use of complexity metrics at design time is to improve the design by plummeting the complexity of the modules that have been established to be most complex. This will directly improve the testability and maintainability. If the complexity cannot be reduced because it is inherent in the problem, complexity metrics can be used to highlight the more complex modules. As complex modules are often more error-prone, this feedback can be used by project management to ensure that strict quality assurance is performed on these modules as they evolve. Overall, complexity metrics are of great interest at design time and they can be used to evaluate the quality of design, improve the design, and improve quality assurance of the project. We will describe some of the metrics that have been proposed to quantify the complexity of design.

A Validation metric is the basis for comparing features from experimental data with model predictions. Validation metrics are established during the requirements phase of the conceptual model development and incorporate numerical and experimental uncertainty. If the error, e , between experimental data, y , and model prediction, *y , is given by *e = y − y , a simple metric could be the expected value of the error, E( ) e , or the variance of the error, V( ) e . Other metrics could include, for example: P ( ) e > 0 , where P( ) is the probability.

Percentiles on the probability distribution of e; or a hypothesis test such as E ( ) e > 0, where the validation metric is a pass/fail decision of whether or not the model is contradicted by the data. In selecting the validation metric, the primary consideration should be what the model must predict in conjunction with what types of data could be available from the experiment. Additionally, the metrics should provide a measure of agreement that includes uncertainty requirements. Complex model simulations generate an enormous amount of information from which to choose. The selection of the simulation outcome should first be driven by application requirements. For example, if a design requirement is that the peak strain at specified location should not exceed some value, and then the model validation should focus on comparison of measured and computed strains at that location. Features of experimental data and model outputs must be carefully selected. A feature may be simple, such as the maximum response for all times at a specific location in the computational domain, or more complex, such as the complete response history at a specific location, modal frequencies, mode shapes, load-deflection curve slopes and inflection points, peak amplitudes, signal decay rates, temporal moments, shock response spectra, etc. In some cases, a feature can be used directly as a validation metric; in other cases, the feature must be processed further into a form more suitable for comparison with experimental data.

Now we discuss some of the metrics that can be extracted from a design and that could be useful for evaluating the design. We do not discuss the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring. Size is always a product metric of interest, as size is the single most influential factor deciding the cost of the project. As the actual size of the project is known only when the project ends, at early stages the project sizes only an estimate. Our ability to estimate size becomes more accurate as development proceeds. Hence, after design, size (and cost) re-estimation are typically done by project management. After design, as all the modules in the system and major data structures are known, the size of the final system can be estimated quite accurately.

For estimating the size, the total number of modules is an important metric. This can be easily obtained from the design. By using an average size of a module, from this metric the final size in LOG can be estimated. Alternatively, the size of each module can be estimated, and then the total size of the system will be estimated as the sum of all the estimates.

7.2.1 Network Metrics

Network metrics for design focus on the structure chart (mostly the call graph component of the structure chart) and define some metrics of how "good" the structure or network is in an effort to quantify the complexity of the call graph. As coupling of a module increases if it is called by more modules, a good structure is considered one that has exactly one caller. That is, the call graph structure is simplest if it is a pure tree. The more the structure chart deviates from a tree, the more complex the system. Deviation of the tree is then defined as the graph impurity of the design. Graph impurity can be defined as:

Graph impurity = n – e – 1

Where n is the number of nodes in the structure chart and e is the number of edges. As in a pure tree the total number of nodes is one more than the number of edges, the graph impurity for a tree is O. Each time a module has a fan-in of more than one, the graph impurity increases. The major drawback of this approach is that it ignores the common use of some routines like library or support routines. An approach to handle this is not to consider the lowest-level nodes for graph impurity because most often the lowest-level modules are the ones that are used by many different modules, particularly if the structure chart was factored. Library routines are also at the lowest level of the structure chart (even if they have a structure of their own, it does not show in the structure chart of the application using the routine). Other network metrics have also been defined. For most of these metrics, significant correlations with properties of interest have not been established. Hence, their use is limited to getting some idea about the structure of the design.

7.2.2 Stability Metrics

We know that maintainability of software is a highly preferred quality. Maintenance activity is hard and error-prone as changes in one module require changes in other modules to maintain consistency, which require further changes, and so on. It is clearly desirable to minimize this ripple effect of performing a change, which is largely determined by the structure of the software. Stability of a design is a metric that tries to quantify the resistance of a design to the potential ripple effects that are caused by changes in modules. The higher the stability of a program design, the improved the maintainability of the program. Here we define the stability metric. At the lowest level, stability is defined for a module. From this, the stability of the whole system design can be obtained. The aim is to define a measure so that the higher the measure the less the ripple effect on other modules that in some way are related to this module. The modules that can be affected by change in a module are the modules that invoke the module or share global data (or files) with the module. Any other module will clearly not be affected by change in a module. The potential ripple effect is defined as the total number of assumptions made by other modules regarding the module being changed. Hence, counting the number of assumptions made by other modules is central to determining the stability of a module. As at design time only the interfaces of modules are known and not their internals, for calculating design stability only the assumptions made about the interfaces need be considered. The interface of a module consists of all elements, through which this module can be affected by other modules, i.e., through which this module can be coupled with other modules. Hence, it consists of the parameters of the modules and the global data the module uses. Once the interface is identified, the structure of each element of the interface is examined to determine all the minimal entities in this element for which assumptions can be made. The minimal entities generally are the constituents of the interface element. For example, a record is broken into its respective fields as a calling module can make assumptions about a particular field. For each minimal entity at least two categories of assumptions can be made—about the type of the entity and about the value of the entity. (The assumption about the type is typically checked by a compiler if the programming language supports strong typing.) Each minimal entity in the interface is considered as contributing one assumption in each category. In this formulation, the effect of a change in a module i on another module j is represented as a probability. For the entire system, the effect of change is captured by the probability of change metrics C. An element C[i, j] of the matrix represents the probability that a change in module i will result in a change in module j. With this matrix the ripple effect of a change in a module can also be easily computed. This can then be used to model the stability of the system. The main problem with this metric is to estimate the elements of the matrix.

7.2.3 Information Flow Metrics

The network metrics of graph impurity had the basis that as the graph impurity increases, the coupling increases. However, it is not a very good approximation for coupling, as coupling of a module increases with the complexity of the interface and the total number of modules a module is coupled with, whether it is the caller or the callee. So, if we want a metric that is better at quantifying coupling between modules, it should handle these. The information flow metrics attempt to define the complexity in terms of the total information flowing through a module.

In one of the earliest work on information flow metrics, the complexity of a module is considered as depending on the intramodule complexity and the intermodule complexity. The intramodule complexity is approximated by the size of the module in lines of code (which is actually the estimated size at design time). The intermodule complexity of a module depends on the total information flowing in the module (inflow) and the total information flowing out of the module (outflow). The inflow of a module is the total number of abstract data elements flowing in the module (i.e., whose values are used by the module), and the outflow is the total number of abstract data elements that are flowing out of the module (i.e., whose values are defined by this module and used by other modules). The module design complexity, Dc, is defined as:

$$Dc, = \text{size} * (\text{inflow} * \text{out flow})^2$$

The term (inflow * outflow) refers to the total number of combinations of input source and output destination. This term is squared, as the interconnection between the modules is considered a more important factor (compared to the internal complexity) determining the complexity of a module. This is based on the common experience that the modules with more interconnections are harder to test or modify compared to other similar-size modules with fewer interconnections. The metric defined earlier defines the complexity of a module purely in terms of the total amount of data flowing in and out of the module and the module size. A variant of this was proposed based on the hypothesis that the module complexity depends not only on the information flowing in and out, but also on the number of modules to or from which it is flowing. The module size is considered an insignificant factor, and complexity D, for a module is defined as:

$$D_{c'} = \text{fan\_in} * \text{fan\_out} + \text{inflow} * \text{outflow}$$

Notes

Where fan_in represents the number of modules that call this module and fan_out is the number of modules this module calls. The main question that arises is how good these metrics are. For "good," we will have to define their purpose, or how we want to use them. Just having a number signifying the complexity is, in itself, of little use, unless it can be used to make some judgment about cost or quality. One way to use the information about complexity could be to identify the complex modules, as these modules are likely to be more error prone and form "hot spots" later, if they are left as is. Once these modules are identified, the design can be evaluated to see if the complexity is inherent in the problem or if the design can be changed to reduce the complexity. To identify modules that are "extra complex," we will have to define what complexity number is normal. Having a threshold complexity above which a module is considered complex assumes the existence of a globally accepted threshold value. This may not be possible, as designs in different problem domains produce different types of modules. Another alternative is to consider a module against other modules in the current design only, instead of comparing the modules against a prespecified standard. That is, evaluate the complexity of the modules in the design and highlight modules that are, relatively speaking, and more complex. In this approach, the criterion for marking a module complex is also determined from the current design. One such method for highlighting the modules was suggested. Let avg _complexity be the average complexity of the modules in the design being evaluated and let std_deviation be the standard deviation in the design complexity of the modules of the system. The proposed method classifies the modules in three categories: error-prone, complex, and normal. If D_c, is the complexity of a module.

Caution Validation metrics must be established during the validation requirement phase of the conceptual model development and should include estimates of the numerical and experimental error.

Case Study

The NENE Code Project

The Defence Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) Program is sponsoring a series of case studies to identify the life cycles, workflows, and technical challenges of computational science and engineering code development that are representative of the program's participants. A secondary goal is to characterize how software development tools are used and what enhancements would increase the productivity of scientific-application programmers. These studies also seek to identify "lessons learned"? That can be transferred to the general computational science and engineering community to improve the code development process.

The NENE code is the fifth science-based code project to be analyzed by the Existing Codes sub team of the DARPA HPCS Productivity Team. The NENE code is an application code for analyzing scientific phenomena and predicting the complex behaviour and interaction of individual physical systems and individual particles in the systems. The core NENE development team is expert, agile, and of moderate size, consisting of a professor and another permanent staff member, five post docs, and 11 graduate students. NENE is an example of a distributed development project; the core team is anchored at a university, but as many as 250 individual researchers have made contributions from other locations.

Questions

1. What is the NENE Code Project?

2. What is the DARPA?

Self Assessment Questions

6. Verification is the …………………. of testing like set of documents, plans, specifications and requirements.

   (*a*) software (*b*) process

   (*c*) model. (*d*) None of these.

7. The NENE code is the ………………… science-based code project.

   (*a*) fifth (*b*) six

   (*c*) fourth (*d*) None of these.

8. Program consists of …………………. and as the information flow between the Modules increased.

   (*a*) modules (*b*) process

   (*c*) Program (*d*) None of these.

9. That determining GRx and …………………., is not always possible when pointers and indirect referencing are used.

   (*a*) GRx (*b*) SDx

   (*c*) GDx (*d*) PDx

10. A Validation metric is the basis for comparing features from …………………., with model predictions.

    (*a*) referencing data (*b*) flow data

    (*c*) experimental data (*d*) None of these.

11. The biggest disadvantage of …………………., is that It is very rigid and the least flexible

    (*a*) R-model (*b*) V-model

    (*c*) G-model (*d*) None of these.

12. The terms verification and …………………., have different meaning depending on the discipline in engineering or quality management systems.

    (*a*) validation (*b*) techniques

    (*c*) verification (*d*) None of these.

13. These techniques are presented as …………………., practices within peer review processes and as autonomous engineering investigations.

    (*a*) both (*b*) techniques

    (*c*) some (*d*) path

14. This is an introductory report on the use of model-based verification techniques within …………………., and upgrade practices.

    (*a*) both (*b*) techniques

    (*c*) models (*d*) software development

15. The scope of this product includes integral validation of …………………., property data.

    (*a*) physical (*b*) techniques

    (*c*) models (*d*) software development

Notes

7.3 Summary

- Software Verification and Validation (V&V) is a systems engineering discipline helping a development organization build quality into the application software during the software life cycle.
- Services must be provided and managed by an organization that is technically and managerially independent of the subject software development project.
- Verification is the process of testing like set of documents, plans, specifications and requirements.
- The area provides traceable and reproducible work products and processes for stockpile certification (short and long term), as well as foundational elements for establishing software quality standards and training weapons scientists in the application of verification, validation, and UQ methods.
- A validation metric is the basis for comparing features from experimental data with model predictions. Validation metrics are established during the requirements phase of the conceptual model development and incorporate numerical and experimental uncertainty
- Network metrics for design focus on the structure chart (mostly the call graph component of the structure chart) and define some metrics of how "good" the structure or network is in an effort to quantify the complexity of the call graph.
- The inflow of modules the total number of abstract data elements flowing in the module and the outflow is the total number of abstract data elements that are flowing out of the module.

7.4 Keywords

Amplitude: It is the objective measurement of the degree of change (positive or negative) in atmospheric pressure (the compression and rarefaction of air molecules) caused by sound waves.

Impurity: It is that substance which is or is added in elements mostly in semiconductors in order to make it more stabilize as required.

Shock Response Spectra: The shock response spectrum (SRS) is quite often used in the analysis of transient shock to help establish design criteria and test specifications.

Testing: It is the process of running a system with the intention of finding errors. Testing enhances the integrity of a system by detecting deviations in design and errors in the system.

Validation Metric: It is the basis for comparing features from experimental data with model predictions.

Lab Exercise

1. Search about verification modules.
2. Differentiate between Verification and Validation.

7.5 Review Questions

1. Explain the term verification.
2. What are the benefits of verification?
3. Explain the concept of metrics.
4. Describe the network metrics.

5. What is the stability metrics?
6. Define information flow metrics.
7. Write the advantage and disadvantage verification.
8. What is the stockpile stewardship program?
9. What is the testing? How to use in verification?
10. Explain data validation, archiving, SQA, and training.

Answers for Self Assessment Questions

1.	(*a*)	2.	(*b*)	3.	(*b*)	4.	(*d*)	5.	(*b*)
6.	(*b*)	7.	(*a*)	8.	(*a*)	9.	(*c*)	10.	(*c*)
11.	(*b*)	12.	(*a*)	13.	(*a*)	14.	(*d*)	15.	(*a*)

7.6 Further Reading

Books

Hardware Verification with SystemVerilog: An Object-oriented Framework by Mike Mintz, Robert Ekendahl.

Online link

http://books.google.co.in/books?id=Mawfv_drBJoC&pg=PA6&dq=Further+Readings+Introduction+to+Verification&hl=en&sa=X&ei=t2QKUPWSKY3NrQfZk_DHCA&ved=0CDcQ6AEwAQ#v=onepage&q=Further

Unit 8: Detailed Design

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the detailed design
- Discuss the process design language
- Explain the logic/algorithm design
- Describe the verification of logic/algorithm design

Introduction

We focus on the modules in a system and how they interrelate with each other. Once the modules are identified and specified during the high-level design, the interior logic that will implement the given stipulation can be designed, and is the focus of this section.

The detailed design activity is often not performed formally and archived as it is almost not possible to keep the detailed design document consistent with the code. Due to this, developing the detailed design is useful for the more multifaceted and important modules, and is often done informally by the programmer as part of the personal process of developing code.

The design of a system is a plan for a solution such that if the plan is implemented, the implemented system will satisfy the requirements of the system and will preserve its architecture. The module-level design specifies the modules that should be there in the system to implement the architecture, and the detailed design the processing logic of modules.

Software design is the 'process of defining the architecture, components, interfaces, and other characteristics of a system or component'. Detailed design is the process of defining the lower level components, modules and interfaces.

Production is the process of:

- *Programming*: Coding the components;
- *Integrating*: Assembling the components;
- *Verifying*: Testing modules, subsystems and the full system.

The physical model outlined in the AD phase (Architecture Design Phase) is comprehensive to produce a structured set of component specifications that are consistent, coherent and complete. Each specification defines the functions, Inputs, Outputs and Internal processing of the component.

The software components are documented in the Detailed Design Document (DDD). The DDD is a comprehensive specification of the code. It is the primary reference for maintenance staff in the Transfer phase (TR phase) and the Operations and Maintenance phase (OM phase).

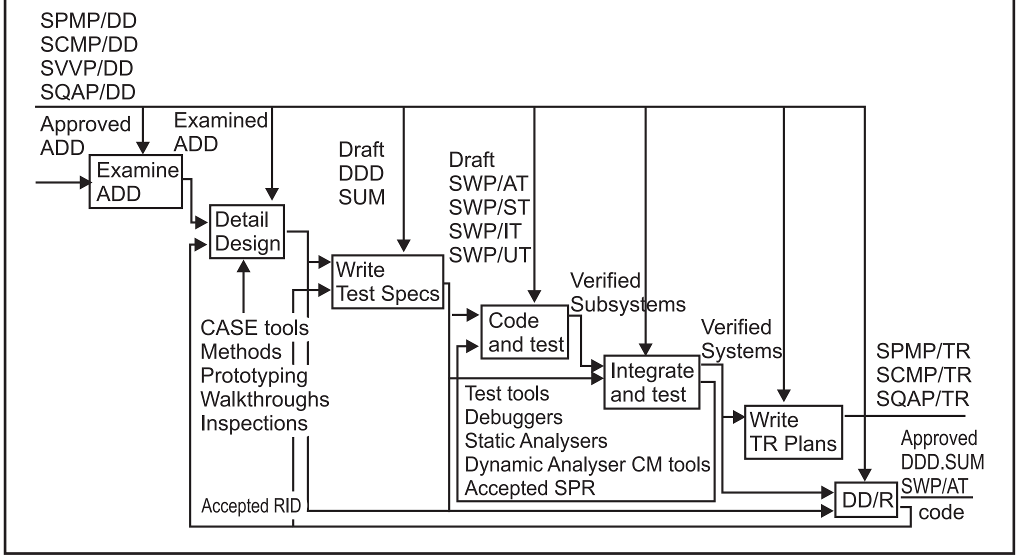
The main outputs of the DD phase are the: Source and Object code;

- Detailed Design Document (DDD);
- Software User Manual (SUM);
- Software Project Management Plan for the TR phase (SPMP/TR);
- Software Configuration Management Plan for the TR phase (SCMP/TR);
- Software Quality Assurance Plan for the TR phase (SQAP/TR);
- Acceptance Test specification (SVVP/AT).

Progress reports, configuration status accounts, and audit reports are also outputs of the phase. These should always be archived. The detailed design and production of the code is the responsibility of the developer. Engineers developing systems with which the software interfaces may be consulted during this phase. User representatives and operations personnel may observe system tests. DD phase activities must be carried out according to the plans defined in the AD phase. Progress against plans should be continuously monitored by project management and documented at regular intervals in progress reports. Figure 8.1 is an ideal representation of the flow of software products in the DD phase. The reader should be aware that some DD phase activities can occur in parallel as separate teams build the major components and integrate them. Teams may progress at different rates; some may be engaged in coding and testing while others are designing. The following subsections discuss the activities shown in Figure 8.1.

Notes

Figure 8.1: The DD Phase Actives



The design of a software system is split into two phases:

1. High-level design
   - A system viewed as a set of modules
2. Low-level design
   - Providing detailed information about each module
3. After high-level design, a designer's focus shifts to low-level design:

Each module's responsibilities should be specified as precisely as possible

- constraints on the use of its interface should be specified
- pre and post conditions can be identified
- module-wide invariants can be specified
- internal data structures and algorithms can be suggested

In detailed design of procedures, the logic for implementing the procedure is specified in a semiformal notation. For classes, state diagrams can be used to model the relationships between methods.

8.1 Meaning of Detailed Design

Design standards have to be set at the create of the DD (detailed design) phase by project management to organize the collective labors of the team. This is especially necessary when development team members are working in parallel. The developers must first complete the top-down rotting of the software started in the AD phase and then outline the processing to be carried out by each component. Developers must continue the structured approach and not introduce unnecessary complexity. They must build defenses against likely problems. Developers should verify detailed designs in design reviews, level by level. Review of the design by walkthrough or inspection before coding is a more efficient way of eliminating design errors than testing.

The DD phase can be called the 'implementation phase' of the life cycle. The purpose of the DD phase is to detail the design outlined in the ADD, and to code, document and test it. The detailed design and production is the responsibility of the software engineers. Other kinds of engineers may be consulted during this phase, and representatives of users and operations personnel may observe system tests. The software may be independently verified by engineers not responsible for detailed design and coding. Important considerations before starting code production are the adequacy and availability of computer resources for software development. There is no point in starting coding and testing if the computer, operating system and system software are not available or sufficiently reliable and stable. Productivity can drop dramatically if these resources are not adequate. The principal output of this phase is the code, the Detailed Design Document (DDD) and Software User Manual (SUM). The DD phase terminates with formal approval of the code, DDD and SUM by the Detailed Design Review (DD/R).

8.1.1 Definition of Design Standards

Anywhere probable, standards and conventions used in the AD phase should be carried over into the DD phase. They should be documented in part one of the DDD. Standards and conventions should be defined for:

- Design Methods
- Documentation
- Naming Components
- Computer Aided Software Engineering (CASE) tools;
- Error handling.

8.1.2 Decomposition of the Software into Modules

As in architectural design, the first stage of detailed design is to define the functions, Inputs and Outputs of each software component. Whereas architectural design only considered the major, top-level components, this stage of detailed design must specify all the software components.

The developer starts with the major components defined in the ADD and continues to decompose them until the components can be expressed as modules in the selected programming languages.

Decomposition should be carried out using the same methods and tools employed in the AD phase. CASE tools and graphical methods employed in architectural design can still be used.

Component processing is only specified to the level of detail necessary to answer the question: ‘is further decomposition necessary?’

Decomposition criteria are:

- Will the module have too many statements?
- Will the module be too complex?
- Does the module have low cohesion?
- Does the module have high coupling?
- Does the module contain similar processing to other modules?

8.1.3 Reuse of the Software

Software reuse questions can arise at all stages of design. In the AD phase decisions may have been taken to reuse software for all or some major components, such as:

- Application generators;
- Database management systems;
- Human-computer interaction utilities;
- Mathematical utilities;
- Graphical utilities.

In the DD phase developers may have to:

- Decide which library modules to use;

- Build shells around the library modules to standardize interfaces (e.g. for error handling) and enhance portability;
- Define conventions for the library modules to use (more than one combination of library modules might do the same job).

Developers should resist the temptation to clone a library module and make minor modifications, because of the duplication that result. Instead they should put any extra processing required in a shell module that calls the library module. A common reason for reusing software is to save time at the coding and unit testing stage. Reused software is not normally unit tested because this has already been carried out in a project or by an external organization. Nevertheless developers should convince themselves that the software to be reused has been tested to the standards appropriate for their project. Integration testing of reused software is always necessary to check that it correctly interfaces with the rest of the software. Integration testing of reused software can identify performance and resource problems.

8.1.4 Definition of Module Processing

The developer defines the processing steps in the second stage of detailed design. The developer should first outline the module processing in a Program Design Language (PDL) or pseudo-code and refine it, step-by step, into a detailed description of the processing in the selected programming language. The processing description should reflect the type of programming language. When using a procedural language, the description of the processing should contain only:

- Sequence constructs (e.g. assignments, invocations);
- Selection constructs (e.g. conditions, case statements);
- Iteration constructs (e.g. do loops).

The definition of statements that do not affect the logic (e.g. I/O statements, local variable declarations) should be deferred to the coding stage. Each module should have a single entry point and exit point. Control should flow from the entry point to exit point. Control should flow back only in an iteration construct, i.e. a loop. Branching, if used at all, should be restricted to a few standard situations (e.g. on error), and should always jump forward, not backward, in the control flow.

Recursion is a useful technique for processing a repeating data structure such as a tree, or a list, and for evaluating a query made up of arithmetic, relational, or logical expressions. Recursion should only be used if the programming language explicitly supports it. PDLs and pseudo-code can be included in the code as comments, easing maintenance, whereas flowcharts cannot. Flowcharts are not compatible with the stepwise refinement technique, and so PDLs and pseudo-code are to be preferred to flowcharts for detailed design.

8.1.5 Defensive Design

Developers should anticipate possible problems and include defences against them. There are described three principles of defensive design:

- Mutual suspicion;
- Immediate detection;
- Redundancy.

The principle of mutual suspicion says that modules should assume other modules contain errors. Modules should be designed to handle erroneous input and error reports from other modules.

Every input from another module or component external to the program (e.g. a file) should be checked. When input data is used in a condition (e.g. CASE statement or IF... THEN.. ELSE...), an outcome should be defined for every possible input case. Every IF condition should have an ELSE clause. When modules detect errors and return control to the caller, they should always inform the caller that an error has occurred. The calling module can then check the error flag for successful completion of the called module. It should be unnecessary for the caller to check the other module outputs. It is possible for a subordinate to fail to return control. Modules should normally set a 'timeout' when waiting for a message, a rendezvous to be kept, or an event flag to be set. Modules should always act appropriately after a timeout (e.g. retry the operation).

The principle of immediate detection means that possible errors should be checked for immediately. If the reaction is not immediate, the error should be flagged for later action. Conventions for taking action after detection of an error should be established. Normally error control responsibilities of a module should be at the same level as normal control responsibilities. Error logging, if done at all, should be done at the point where action is taken, and not at the point of error detection, since the significance of the error may not be apparent at the point of detection. It may be appropriate to insert diagnostic code at the point of detection, however. Only the top level module should have responsibility for stopping the program. Developers should always check that there are no 'STOP' statements lurking in an otherwise apparently useful module. It may be impossible to reuse modules that suddenly take over responsibility for the control flow of the whole system. Halting the program and giving a trace back may be acceptable in prototype software (because it helps fault diagnosis), but not in operational software.

In detailed design, redundancy considerations can lead designers to include checksums in records and identity tags to confirm that an item is really what it is assumed to be (e.g. header record).

Myers also makes a useful distinction between 'passive fault detection' and 'active fault detection'. The passive fault detection approach is to check for errors in the normal flow of execution. Examples are modules that always check their input, and status codes returned from system calls. The active fault detection approach is to go looking for problems instead of waiting for them to arise. Examples are 'monitor' programs that continuously check disk integrity and attempts to violate system security. Often most system code is dedicated to error handling. Library modules for error reporting should be made available to prevent duplication of error handling code. When modules detect errors, they should call error library modules to perform standard error handling functions such as error logging and display.

Defensive design principles have influenced the design of most modern languages. Strong type-checking languages automatically check that the calling data type matches the called data type, for example. Analysis and design algorithm goes further and builds range checking into the language. The degree to which a language supports defensive design can be a major factor in its favours.

8.1.6 Optimization

Conventionally, optimization earnings to make the best compromise between opposing tendencies. Improvement in one area is often associated with degradation in another. Software performance is often traded-off against maintainability and portability, for example.

The optimization process is to:

- Define the attributes to change (e.g. execution time);
- Measure the attribute values before modifying the software;
- Measure the attribute values after modifying the software;
- Analyze the change in attribute values before deciding whether to
- Modify the software again.

Optimization can stop when the goals set in the SRD have been met. Every change has some risk, and the costs and benefits of each change should be clearly defined. The law of diminishing returns can also be used to decide when to stop optimization. If there are only slight improvements in the values of attribute values after optimization, the developers should stop trying to seek improvements.

Failure to get a group of people to agree about the solution to an optimization problem is itself significant. It means that the attribute is probably optimized, and any improvement in one attributes results in an unacceptable degradation in another. The structured programming method discourages optimization because of its effect on reliability and maintainability. Code should be clear and simple, and its optimization should be left to the compiler. Compilers are more likely to do a better job of optimization than programmers, because compilers incorporate detailed knowledge of the machine. Often the actual causes of inefficiency are quite different from what programmers might suspect, and can only be revealed with a dynamic analysis tool.

8.1.7 Prototyping

Experimental prototyping can be useful for:

- Comparing alternative designs;
- Checking the feasibility of the design.

The high-level design will usually have been recognized during the AD phase. Detailed designs may have to be prototyped in the DD phase to find out which designs best meet the requirements.

The feasibility of a novel design idea should be checked by prototyping. This ensures that an idea not only works, but also that it works well enough to meet non-functional requirements for quality and performance.

8.1.8 Design Reviews

Detailed designs should be reviewed top-down, level by level, as they are generated during the DD phase. Reviews may take the form of walkthroughs or inspections. Walkthroughs are useful on all projects for informing and passing on expertise. Inspections are efficient methods for eliminating defects before production begins.

Two types of walkthrough are useful:

- Code reading;
- 'What-if?' analysis.

In a code reading, reviews trace the logic of a module from beginning to end. In "what-if?" analysis, component behaviour is examined for specific inputs. Static analysis tools evaluate modules without executing them. Static analysis functions are built in to some compilers. Output from static analysis tools may be input to a code review.

When the detailed design of a major component is complete, a critical design review must certify its readiness for implementation. The project leader should participate in these reviews, with the team leader and team members concerned.

8.1.9 Documentation

The developers must produce the DDD and the sum. While add specifies tasks, files and programs, the DDD specifies modules. The sum describes how to use the software, and may be affected by documentation requirements in the SRD.

The recommended approach to module documentation is:

- Create the module template to contain headings for the standard DDD entries:
- Component identifier
  1. Type
  2. Purpose
  3. Function
  4. Subordinates
  5. Dependencies

6. Interfaces
7. Resources
8. References
9. Processing
10. Data

- Detail the design by filling, with the processing section containing the high-level definition of the processing in a PDL or pseudo-code;
- Assemble the completed templates for insertion in the DDD.

A standard module template enables the DDD component specifications to be generated automatically. Tools to extract the module header from the source code and create an entry for the DDD greatly simplify maintenance. When a module is modified, the programmer edits, compiles and verifies the source module, and then runs the tool to generate the new DDD entry.

The SUM contains the information needed by the users of the software to operate it. The SUM should be gradually prepared during the DD phase.

8.1.10 Coding Standards

Coding standards should be established for all the languages used, and recognized or referenced in the DDD. They should provide rules for:

- Presentation, (e.g. header information and comment layout);
- Naming programs, subprograms, files, variables and data;
- Limiting the size of modules;
- Using library routines, especially:
  - Operating system routines;
  - Commercial library routines (e.g. numerical analysis);
  - Project-specific utility routines;
- Defining constants;
- Defining data types;
- Using global data;
- Using compiler specific features not in the language standard;
- Error handling.

Caution The developers should exercise the procedures described in the SUM while testing the software.

8.2 Process Design Language

Process Design Languages have been used for some time in the informal description of software and as an aid to top-down program building. In general, such languages have not been amenable to the automatic age group of project management information (such as module interconnection details and complexity measurements) useful in the software engineering process.

The need to improve software productivity and reliability has become an area of increasing importance to the software engineering community. During the early phases of the software life cycle, there is a strong need to emphasize the use of tools which provide a representation of the program structure that can be easily understood and modified. Thus, a significant usage of Program Design Languages (PDLs) as a design tool in practical environments has surfaced in recent years. The desired characteristics of such a design tool and survey a representative sample of existing program design languages. Finally, a new POL Environment is proposed for further consideration. Index Terms - design tools program design languages, formally defined design constructs, software reusability, software metrics, PDL Environment.

The Unified modelling Language (UML) has various types of diagrams to model different properties, which allow both static be used to show how a scenario is implemented by involving different objects. Using the UML notation, object oriented designs of a system can be created. The object-oriented design methodology focuses on identifying classes and relationships between them, and validating the class definitions using dynamic and functional modeling.

Self Assessment Questions

1. Decomposition criteria are:
   (*a*) Application generators
   (*b*) Database management systems
   (*c*) Human-computer interaction utilities
   (*d*) Does the module have low cohesion?
2. In the DD phase developers may have to:
   (*a*) Application generators (*b*) decide which library modules to use
   (*c*) Mathematical utilities (*d*) Graphical utilities
3. The principle of mutual suspicion says that modules should assume other modules contain errors.
   (*a*) Mutual suspicion (*b*) Immediate detection
   (*c*) Redundancy (*d*) None of these
4. The optimization process is to:
   (*a*) Human-computer interaction utilities
   (*b*) decide which library modules to use
   (*c*) measure the attribute values before modifying the software
   (*d*) Graphical utilities
5. A significant usage of program design languages as a design tool in practical environments has surfaced in recent years.
   (*a*) True (*b*) False

8.3 Logic and Algorithm Design

Algorithms and Logic section is the technical basis for constructing healthy, efficient, and intelligent software applications based on mathematically sound solutions. Two crucial aspects when designing efficient software are large data sets and essentially hard problems. The naive algorithm or data structure usually only suffices for solving small scale problems. A simple search engine can easily index the contents of a drive on a personal computer, but indexing the web is much more difficult. Similarly, an autonomous robot can plan its actions optimally by an exhaustive state space search only if the number of possible actions and states is very limited. Thus more advanced algorithms/methods are needed to solve these problems. In software applications for complex problems, often some degree of intelligence is needed in addition to efficiency. Intelligence means that the software is able to collect and categorize knowledge, do logical reasoning, learn from experiences, and communicate and negotiate with other software applications. The scientific foundation for such applications includes computational logic and logic-based AI.

Algorithmic: Algorithmic is the part of computer science that deals with the design and analysis of algorithms and data structures and constitutes the scientific foundation for reasoning about resources used in computing such as time and space. This covers both the design and analysis of efficient algorithms solving concrete problems, and also with identifying common patterns of problems and associated algorithmic paradigms that can lead to efficient solutions for classes of problems.

Computational Logic: Computational logic is the study of logic and logical methods within computer science. Logic is the study of valid inferences, and in computational logic it is studies how to automate such inferences on a computer. Computational logic is used for the specification and verification of software and hardware systems, for topics in databases and programming languages, and for logic-based AI, such as automated reasoning and knowledge-based systems

The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design. Specifying the logic will require developing an algorithm that will implement the given specifications. Here we consider some principles for designing algorithms or logic that will implement the given specifications. The term algorithm is quite general and is applicable to a wide variety of areas. For software we can consider an algorithm to be an unambiguous procedure for solving a problem. A procedure is a finite sequence of well-defined steps or operations, each of which requires a. finite amount of memory and time to complete. In this definition we swum that termination is an essential property of procedures. From now on we will use procedures, algorithms, and logic interchangeably.

There are a number of steps that one has to perform while developing an algorithm. The starting step in the design of algorithms is statement of the problem. The problem for which an algorithm is being devised has to be precisely and clearly stated and properly understood by the person responsible for designing the algorithm. For detailed design, the problem statement comes from the system design. That is, the problem statement is already available when the detailed design of a module commences. The next step is development of a mathematical model for the problem. In modelling, one has to select the mathematical structures that are best suited for the problem. It can help to look at other similar problems that have been solved. In most cases, models are constructed by taking models of similar problems and modifying the model to suit the current problem. The next step is the design of the algorithm. During this step the data structure and program structure are decided. Once the algorithm is designed, its correctness should be verified.

No clear procedure can be given for designing algorithms. Having such a procedure amounts to automating the problem of algorithm development, which is not possible with the current methods. However, some heuristics or methods can be provided to help the designer design algorithms for modules. The most common method for designing algorithms or the logic for a module is to use the stepwise refinement technique.

The stepwise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The problem starts by converting the specifications of the module into an abstract description of an Algorithm containing a few abstract statements, in each step; one or several statements in the algorithm developed so far are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise that they can easily be converted into programming language statements. During refinement, both data and instructions have to be refined. A guideline for refinement is that in each step the amount of decomposition should be such that it can be easily handled and that represents one or two design decisions. Generally, detailed design is not specified using formal programming languages, but using languages that have formal programming like outer

Notes

structures (like loops, conditionals etc). but a freer format for internal description. This allows the designer to focus on the logic and not its representation in the programming language.

8.3.1 State Modelling of Classes

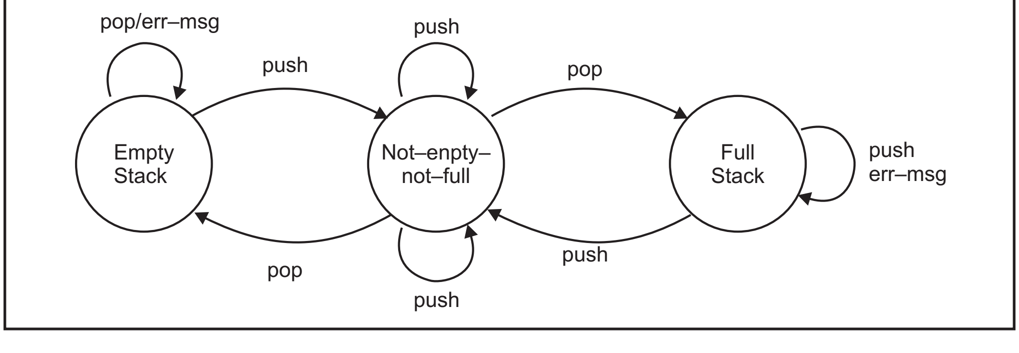
For object-oriented design for obtaining the thorough design can be used for designing the logic of methods. But a class is not a functional concept such cannot be viewed as merely a compilation of functions (methods).

The technique for getting a more detailed understanding of the class as whole, without talking about the logic of different methods, has to be different from the refinement-based approach. An object of a class has some state and many operations on it. To better understand a class, the relationship between the state and various operations and the effect of interaction of various operations has to be understood. This can be viewed as one of the objectives of the detailed design activity for object-oriented development. Once the overall class is better understood, the algorithms for its various methods can be developed.

A method to understand the behaviour of a class is to view it as a finite state Automaton, which consists of states and transitions between states. When modelling an object, the state is the value of its attributes, and an event is the performing of an operation on the object. A state diagram relates events and status by showing how the state changes when an event is performed. A state diagram for an object will generally have an initial state, from which all states are reachable (i.e. there is a path from the initial state to all other states).

A state diagram for an object does not represent all the actual states of the object, as there are many possible states. A state diagram attempts to represent only the logical states of the object. A logical state of an object is a combination of all those states from which the behaviour of the object is similar for all possible events. Two logical states will have different behaviour for at least one event. For example, for an object that represents a stack. all states that represent a stack of size more than 0 and less than same defined maximum are similar as the behaviour of all operations defined on the stack will be similar in all such slates (e.g., push will add an element, pop will remove, etc.). However, the state representing an empty stack is different as the behaviour of top and pop operations are different now (an error message may be returned). Similarly, the state representing a full stack is different. The state model for this bounded size stack is shown in Figure 8.2.

Figure 8.2: The FSA Model of a Stack



The finite state modelling of objects is an aid to understand the effect of various operations defined on the class on the state of the object. A good understanding of this can aid in developing the logic for each of the operations. To develop the logic of operations, regular approaches for algorithm development can be used. The model can also be used to validate if the logic for an operation is correct.

Did u know? The term CASE was originally coined by software company Nastec Corporation of Southfield, Michigan in 1982 with their original integrated graphics and text editor GraphiText, which also was the first microcomputer-based system to use hyperlinks to cross-reference text strings in documents—an early forerunner of today's web page link.

8.4 Verification of Logic and Algorithm Design

The output of the design action should be established before proceeding with the activities of the next phase. If the design is spoken in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.) If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used. The most common approach for verification is design review.

The purpose of design reviews is to ensure that the design satisfies the requirements and is of good quality. If errors are made during the design process, they will ultimately reflect themselves in the code and the final system. As the cost of removing faults caused by errors that occur during design increases with the delay in detecting the errors, it is best if design errors are detected early, before they manifest themselves in the system. Detecting errors in design is the purpose of design reviews. The system design review process is similar to the inspection process, in that a group of people get together to discuss the design with the aim of revealing designs errors or undesirable properties. The review group must include a member of both the system design team and the detailed design team, the requirements document, the author responsible for maintaining the design document, and an independent software quality engineer. As with any review, it should be kept in mind that the aim of the meeting is to uncover design errors, not to try to fix them; fixing is done later.

The number of ways in which errors can enter a design is limited only by the creativity of the designer. The most important design error, however, is that the design does not fully support some requirement. For example, some exception case scenario cannot be handled, or some design constraint has not been satisfied. For design quality, modularity is the main criterion. However, since there is a need to validate whether performance requirements can be met by a design, efficiency is another key property for which a design is evaluated.

8.4.1 Metrics

Here we discuss some of the metrics that can be extracted from a design and that could be useful for evaluating the design. We do not discuss the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring.

Size is always a product metric of interest. For size of a design, the total number of modules is a commonly used metric. (By using an average size of a module, from this metric the final size in LOC can be estimated and compared with project estimates.)

Another metric of interest is complexity. A possible use of complexity metrics at design time is to improve the design by reducing the complexity of the modules that have been found to be most complex. This will directly improve the testability and maintainability. We describe some of the metrics that have been proposed to quantify the complexity of design. We first discuss metrics of function oriented design and then for object oriented design.

Network Metrics Network metrics is a complexity metric that tries to capture how "good" the structure chart is. As coupling of a module increases if it is called by more modules, a good structure is considered one that has exactly one caller. That is, the call graph structure is simplest if it is a pure tree. The more the structure chart deviates from a tree, the more complex the system.

Notes

Case Study

UFAD Building Case Studies and Project Database

Project Objective

Develop a body of information to document completed buildings having underfloor air distribution (UFAD) systems. Build a comprehensive database of UFAD buildings, a series of descriptive project profiles, and in-depth field studies.

Significance to Industry

Underfloor air distribution has proven to be an effective method of delivering conditioned air to localized diffusers in the occupied zone of the building. As more UFAD buildings are completed and occupied, there is a great opportunity to learn from the experiences of these projects in terms of occupant satisfaction, building operations, and whole-building performance.

This project serves as a resource for all parties interested in developing a better understanding of successful designs as well as the barriers to implementation of UFAD systems. It will provide objective information about systems design, operation, and performance as well as how designers have addressed practical issues and overcome limitations in the availability of design tools and product offerings. This information will be useful for building owners, developers, designers, contractors, and manufacturers.

Research Approach

In this project we have developed three levels of progressively more detailed information about existing UFAD projects: (1) Candidate study projects are reviewed and added to our database along with their basic parameters; (2) Project profiles are developed via site visits and interviews with project designers, owners, and facility managers; and (3) In-depth field studies are conducted on selected projects. The project profiles document and characterize underfloor system design issues and solutions, describe the operation of the system, evaluate the overall performance, and list noteworthy attributes discovered during the studies. The field studies focus in more detail on design and operation, and evaluate performance from on-site measurements, analysis of occupant satisfaction surveys, and analysis of energy usage.

Progress on this research includes the identification of over 250 UFAD projects in the U.S. and Canada, and documentation of their key characteristics. We have also conducted an on-line questionnaire to investigate trends and performance of a number of these buildings. Six project profiles have been completed and posted to CBE's under floor technology website, two field studies have been completed, and several Summary Reports and Internal Reports have been produced and distributed to CBE industry partners.

Questions

1. Write the UFAD building significance to industry.

2. Discuss the project objective of the UFAD building.

Self Assessment Questions

6. Which of the following is a tool in design phase?

   (*a*) Abstraction (*b*) Refinement

   (*c*) Information hiding (*d*) All of these.

7. Information hiding is to hide from user. Details ....
   (*a*) that are relevant to him
   (*b*) that are not relevant to him
   (*c*) that may be maliciously handled by him
   (*d*) that are confidential
8. Which of the following comments about object-oriented design of software, is not true?
   (*a*) Objects inherit the properties of the class
   (*b*) Classes are defined based on the attributes of objects
   (*c*) An object can belong to two classes
   (*d*) Classes are always different
9. Design phase includes
   (*a*) Data, architectural and procedural designs only
   (*b*) Architectural, procedural and interface designs only
   (*c*) Data, architectural and interface designs only
   (*d*) Data, architectural, interface and procedural designs
10. The UML stands for …………….
    (*a*) User Modelling language (*b*) Unified Modelling Language
    (*c*) User Manual Language (*d*) Unified Manual Language
11. CASE stands for ……………..
    (*a*) Computer Aided Software Engineering
    (*b*) Computer advance Software
    (*c*) Commonly Aided Software Engineering
    (*d*) None of these.
12. The most creative and challenging phase of system life cycle is
    (*a*) Feasibility study (*b*) Maintenance
    (*c*) Design (*d*) None of these.
13. The database design activity deals with the design of
    (*a*) Logical database (*b*) Physical database
    (*c*) Both (*a*) and (*b*) (*d*) None of these.
14. The advantages of creating a prototype are
    (*a*) It allows developers to experiment with number of different design options
    (*b*) It can serve as a means of communication between developers and customer;
    (*c*) Both (*a*) and (*b*)
    (*d*) None of these.

15. One demerit of functional model is
    (*a*) It is complex to build.
    (*b*) If we change data structure we must modify all the function related to it.
    (*c*) It is difficult to implement.
    (*d*) None of these.

8.5 Summary

- The design of a system is a plan for a solution such that if the plan is implemented, the implemented system will satisfy the requirements of the system and will preserve its architecture.
- A system is considered modular if each module has a well-defined abstraction and if change in one module has minimal impact on other modules.
- A structure chart for a procedural system represents the modules in the system and the call relationship between them.
- The Unified modelling Language (UML) has various types of diagrams to model different properties, which allow both static be used to show how a scenario is implemented by involving different objects.
- Using the UML notation, object-oriented designs of a system can be created. The object-oriented design methodology focuses on identifying classes and relationships between them, and validating the class definitions using dynamic and functional modelling.
- In detailed design of procedures, the logic for implementing the procedure is specified in a semiformal notation. For classes, state diagrams can be used to model the relationships between methods.
- The most common method for verifying a design is design reviews, in which a team of people reviews the design for the purpose of finding defects.

8.6 Keywords

Computer-aided Software Engineering (CASE): The CASE is the use of a computer-assisted method to organize and control the development of software, especially on large, complex projects involving many software components and people.

Program Design Language (PDL): Program Design Languages have been used for some time in the informal description of software and as an aid to top-down program construction.

Redundancy: Redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system, usually in the case of a backup or fail-safe.

Software Design: Software design is the 'process of defining the architecture, components, interfaces, and other characteristics of a system or component'.

Unified Modelling Language (UML): The UML stands for unified modelling language Unified Modelling Language (UML) is a standardized general-purpose modelling language in the field of object-oriented software engineering.

Lab Exercise

1. Draw the structure of detailed design.
2. Draw the flow chart of verification of logic/algorithm design.

8.7 Review Questions

1. What do you mean by software design?
2. Explain the detailed design process.
3. What do you mean by program designed language? Explain it.
4. Describe the logic/algorithm design in detail.
5. Explain the concept of verification of logic/algorithm design.
6. Write is the metrics in detailed design.
7. What is the computational logic?
8. What do you mean by algorithmic?
9. What is the prototyping?
10. What do you mean by defensive design?

Answers for Self Assessment Questions

1.	(*d*)	2.	(*b*)	3.	(*a*)	4.	(*c*)	5.	(*a*)
6.	(*d*)	7.	(*c*)	8.	(*c*)	9.	(*d*)	10.	(*b*)
11.	(*a*)	12.	(*c*)	13.	(*c*)	14.	(*c*)	15.	(*b*)

8.8 Further Readings

Books

Software Engineering 3: Domains, Requirements, and Software Design, by Dines Bjorner

Software Engineering, by Sommerville

Online link

http://books.google.co.in/books?id=TU_RBNTaCIC&printsec=frontcover&dq=SOFTWARE+ENGNEERING+DETAILED+DESIGNED&hl=en&sa=X&ei=T9wHUNKKGIfqrQey55nVAg&ved=0CEwQ6AEwAw#v=onepage&q=SOFTWARE%20ENGNEERING%20DETAILED%20DESIGNED&f=false

Unit 9: Metrics

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the cyclomatic complexity
- Define data binding application software engineering
- Explain the cohesion metric
- Explain the different type of control flow

Introduction

The essential reason of metrics is to offer quantitative data to assist monitor the project. Here we discuss some of the metrics that can be extracted from a plan and that could be useful for evaluating the design. We do not talk about the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring. Size is always a product metric of interest, as dimension is the single the majority influential issue deciding the cost of the project. As the actual size of the project is known only when the project ends, at early stages the project size is only an estimate. Our ability to estimate size becomes more accurate as development proceeds. Hence, after design, size (and cost) re-estimation are typically done by project management. After design, as all the modules in the system and major data structures are known, the size of the final system can be estimated quite accurately. For estimating the size, the total number of modules is an important metric. This can be easily obtained from the design. By using an average size of a module, from this metric the final size in LOC can be estimated. Alternatively, the size of each module can be estimated, then the total size of the system will be estimated as the sum of all the estimates. As a module is a small, clearly specified programming unit, estimating the size of a module is relatively easy.

9.1 Cyclomatic Complexity

Basis Pathway Testing: Basis path testing is a white box testing technique primary proposed by Tom McCabe. The foundation path technique enables to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test Cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

Flow Graph Notation: The flow graph depicts logical control flow using a diagrammatic notation. Each structured construct has a corresponding flow graph symbol.

Cyclomatic Complexity: Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of a basis path testing method, the value computed for Cyclomatic complexity defines the number for independent paths in the basis set of a program and provides us an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition.

Computing Cyclomatic Complexity: Cyclomatic complexity has a foundation in graph theory and provides us with extremely useful software metric. Complexity is computed in one of the three ways:

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G is defined as V (G) = E-N+2P Where E, is the number of flow graph edges, N is the number of flow graph nodes, P is independent component.
3. Cyclomatic complexity, V (G) for a flow graph, G is also defined as V (G) = Pie+1 where Pie is the number of predicate nodes contained in the flow graph G.
4. Graph Matrices: The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. To develop a software tool that assists in basis path testing, a data structure, called a graph matrix can be quite useful.

A Graph Matrix is a square matrix whose size is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections between nodes. The connection matrix can also be used to find the cyclomatic complexity

Cyclomatic complexity measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology. First, it gives the number of recommended tests for software. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow.

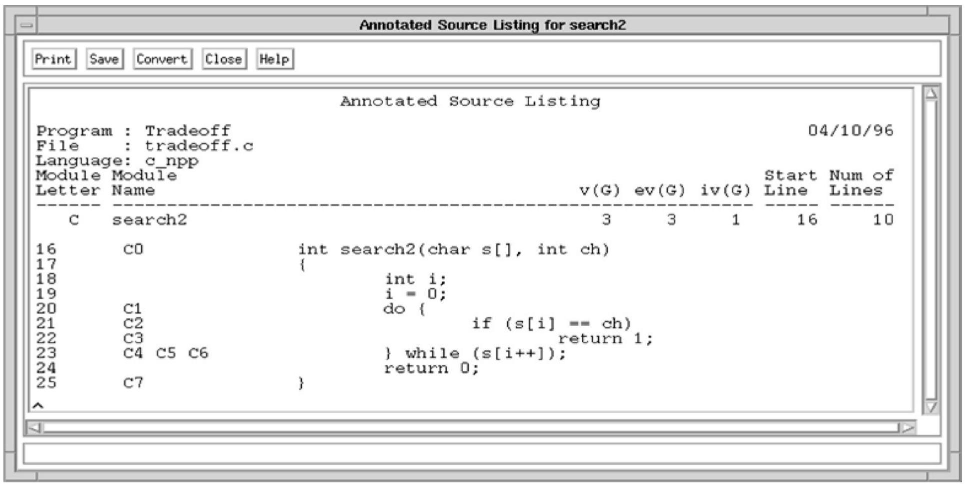
9.1.1 Control Flow Graphs

Control sprint graphs describe the logic arrangement of software modules. A module corresponds to a solitary function or subroutine in typical languages, has a single entry and exit point, and is brilliant to be used as a design component via a call/return mechanism. This document uses C as the language for examples, and in C a module is a function. Each flow graph consists of nodes and edges. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes.

Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph. This correspondence is the foundation for the structured testing methodology.

As an example, consider the C function in which implements Euclid's algorithm for finding greatest common divisors. The nodes are numbered A0 through A13. The control flow graph is shown in, in which each node is numbered 0 through 13 and edges are shown by lines connecting the nodes. Node 1 thus represents the decision of the "if" statement with the true outcome at node 2 and the false outcome at the collection node 5. The decision of the "while" loop is represented by node 7, and the upward flow of control to the next iteration is shown by the dashed line from node 10 to node 7. The path resulting when the module is executed with parameters 4 and 2, as in "Euclid." Execution begins at node 0, the beginning of the module, and proceeds to node 1, the decision node for the "if" statement. Since the test at node 1 is false, execution transfers directly to node 5, the collection node of the "if" statement, and proceeds to node 6. At node 6, the value of "r" is calculated to be 0, and execution proceeds to node 7, the decision node for the "while" statement. Since the test at node 7 is false, execution transfers out of the loop directly to node 11, 8 then proceeds to node 12, returning the result of 2. The actual return is modelled by execution proceeding to node 13, the module exit node. (See Figure 9.1 and Figure 9.2)

Figure 9.1: Annotated Source Listing for Module "Euclid"



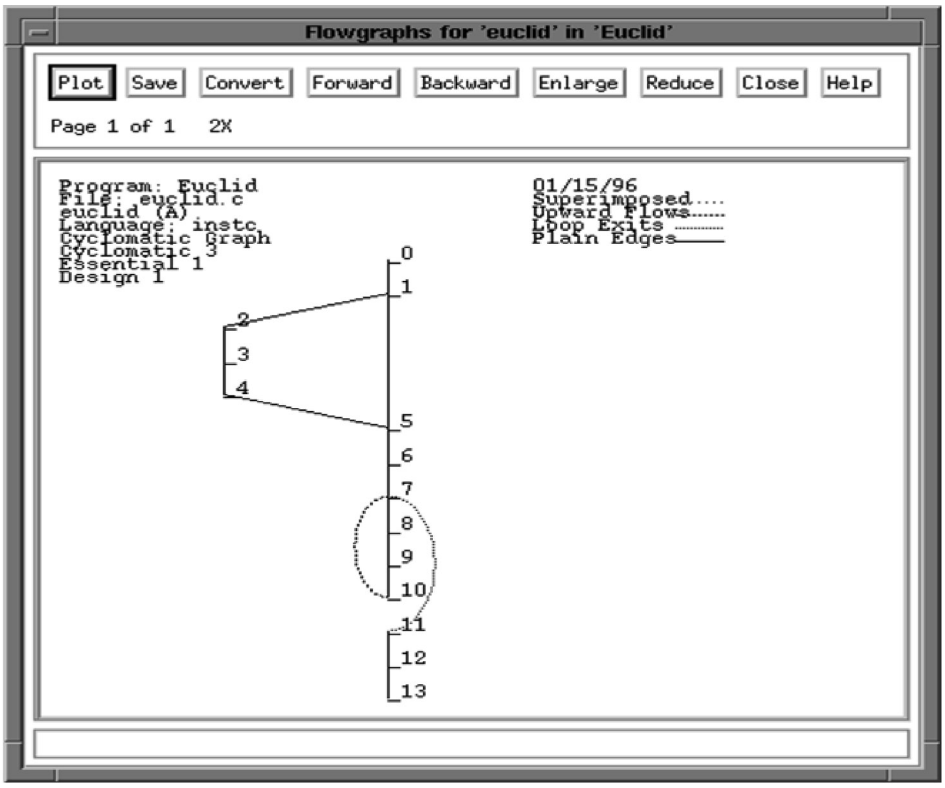
```
                         Annotated Source Listing

Program : Tradeoff                                                04/10/96
File    : tradeoff.c
Language: c_npp
Module Module                                              Start Num of
Letter Name                         v(G) ev(G) iv(G) Line  Lines
------ ------------------------------------------------- ----- ------
   C   search2                         3     3     1    16     10

16      C0            int search2(char s[], int ch)
17                    {
18                            int i;
19                            i = 0;
20      C1                    do {
21      C2                            if (s[i] == ch)
22      C3                                    return 1;
23      C4 C5 C6              } while (s[i++]);
24                            return 0;
25      C7            }
^
```

Definition of cyclomatic complexity v(G)

Cyclomatic complexity is defined for each module to been + 2, where and n are the number of edges and nodes in the control flow graph, respectively. Thus, for the Euclid's algorithm example, the complexity is 3 Cyclomatic complexity is also known as v(G), where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph.

The word "cyclomatic" comes from the number of fundamental (or basic) cycles in connected, undirected graphs [BERGE]. More importantly, it also gives the number of independent Paths

Figure 9.2: Control Flow Graph for Module "Euclid"



```
Program: Euclid                         01/15/96
File: euclid.c                          Superimposed....
euclid (A)                              Upward Flows......
Language: instc                         Loop Exits .........
Cyclomatic Graph                        Plain Edges______
Cyclomatic 3
Essential 1
Design 1
```

through strongly connected directed graphs. A strongly connected graph is one in which each node can be reached from any other node by following directed edges in the Graph. The cyclomatic number in graph theory is defined as n + 1. Program control flow Graphs are not strongly connected, but they become strongly connected when a "virtual edge" Is added connecting the exit node to the entry node. The cyclomatic complexity definition for Program control flow graphs is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge. This definition makes the cyclomatic Complexity equal the number of independent paths through the standard control flow graph model, and avoids explicit mention of the virtual edge.

Control flow graph of with the virtual edge added as a dashed line. This virtual edge is not just a mathematical convenience. Intuitively, it represents the control flow through the rest of the program in which the module is used. It is possible to calculate the amount of (virtual) control flow through the virtual edge by using the conservation of flow equations at the entry and exit nodes, showing it to be the number of times that the module has been executed. For any individual path through the module, this amount of flow is exactly one. Although the virtual edge will not be mentioned again in this document, note that since its flow can be calculated as a linear combination of the flow through the real edges, its presence or absence makes no difference in determining the number of linearly independent paths through the module.

9.1.2 Characterization of v(G) Using a Basis set of Control Flow Paths

Cyclomatic complexity can be characterized as the number of elements of a foundation put of control pours paths through the module. Some familiarity with linear algebra is required to follow the details, but the point is that cyclomatic complexity is precisely the minimum number of Paths that can, in (linear) combination, generate all probable paths through the module. To see this, consider the following mathematical model, which gives a vector space corresponding to each flow graph. Each path has an associated row vector, with the elements corresponding to edges in the flow graph. The value of each element is the number of times the edge is traversed by the path.

Consider the path described in through the graph edges in the graph, the vector has 15 elements. Seven of the edges are traversed exactly once as part of the path, so those elements have value 1. The other eight edges were not traversed as part of the path, so they have value 0.

Considering a set of several paths gives a matrix in which the columns correspond to edges and the rows correspond to paths. From linear algebra, it is known that each matrix has a unique rank (number of linearly independent rows) that is less than or equal to the number of columns. This means that no matter how many of the (potentially infinite) number of possible paths are added to the matrix, the rank can never exceed the number of edges in the graph. In fact, the maximum value of this rank is exactly the cyclomatic complexity of the graph. A minimal set of vectors (paths) with maximum rank is known as a basis, and a basis can also be described as a linearly independent set of vectors that generate all vectors in the space by linear combination. This means that the cyclomatic complexity is the number of paths in any independent set of paths that generate all possible paths by linear combination.

Given any set of paths, it is possible to determine the rank by doing Gaussian Elimination on the associated matrix. The rank is the number of non-zero rows once elimination is complete. If no rows are driven to zero during elimination, the original paths are linearly independent. If the rank equals the cyclomatic complexity, the original set of paths generates all paths by linear combination. If both conditions hold, the original set of paths is a basis for the flow graph.

There are a few important points to note about the linear algebra of control flow paths. First, although every path has a corresponding vector, not every vector has a corresponding path. This is obvious, for example, for a vector that has a zero value for all elements corresponding to edges out of the module entry node but has a nonzero value for any other element cannot correspond to any path. Slightly less obvious, but also true, is that linear combinations of vectors that correspond to actual paths may be vectors that do not correspond to actual paths. This follows from the non-obvious fact that it is always possible to construct a basis consisting of vectors that correspond to actual paths, so any vector can be generated from vectors corresponding to actual paths. This means that one cannot just find a basis set of vectors by algebraic methods and expect them to correspond to paths one must use a path-oriented technique such as that to get a basis set of paths. Finally, there are a potentially infinite number of basis sets of paths for a given module. Each basis set has the same number of paths in it (the cyclomatic complexity), but there is no limit to the number of different sets of basis paths. For example, it is possible to start with any path and construct a basis that contains

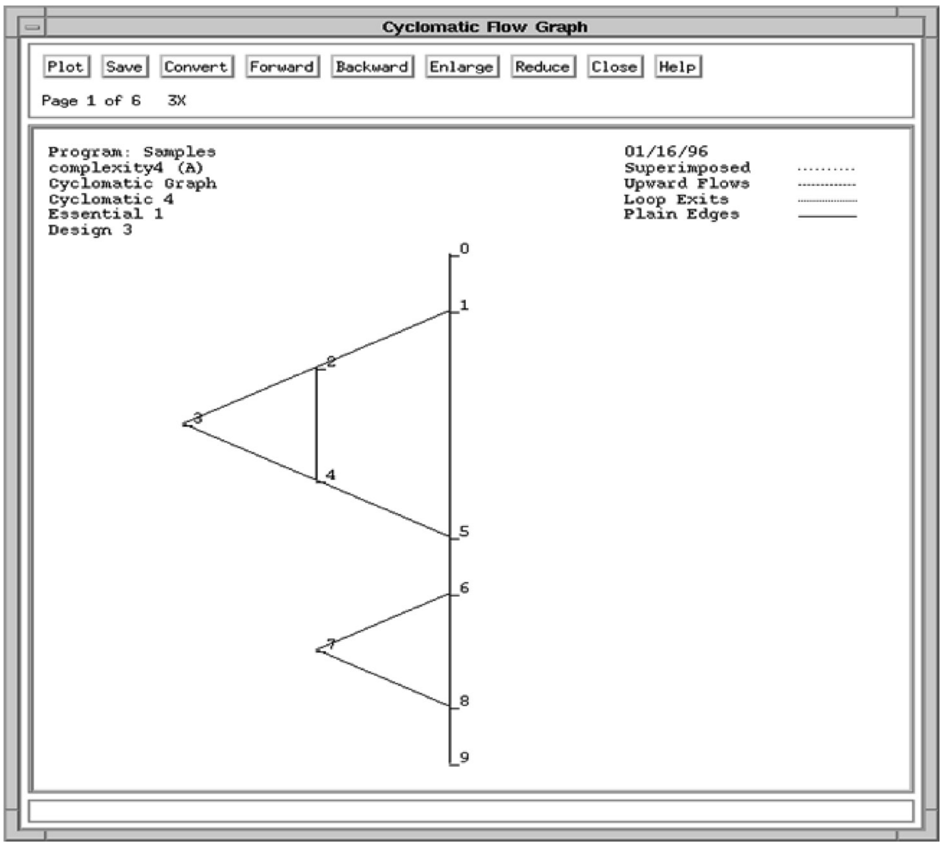
The details of the theory behind cyclomatic complexity are too mathematically complicated to be used directly during software development. However, the good news is that this mathematical Insight yields an effective operational testing method in which a concrete basis set of Paths is tested: structured testing.

Example of v(G) and basis paths

Control flow graph for module "Euclid" with the fifteen edges numbered n0 to 14 along with the fourteen nodes numbered 0 to 13. Since the cyclomatic complexity is 3(15 - 14 + 2), there is a basis set of three paths. One such basis set consists of paths B1 through B3. (See Figure 9.3)

Notes

Figure 9.3: Control Flow Graph with Edges Numbered



```
Program: Samples                               01/16/96
complexity4 (A)                                Superimposed   ..........
Cyclomatic Graph                               Upward Flows   ----------
Cyclomatic 4                                   Loop Exits     ..........
Essential 1                                    Plain Edges    ________
Design 3
```

A basis set of paths, B1 through B3.

Module	:	Euclid
Basis Test Paths	:	3 Paths
Test Path B1	:	0 1 5 6 7 11 12 13
8(1)	:	n>m ==> FALSE
14(7)	:	r!=0 ==> FALSE
Test Path B2	:	0 1 2 3 4 5 6 7 11 12 13
8(1)	:	n>m ==> TRUE
14(7)	:	r!=0 ==> FALSE
Test Path B3	:	0 1 5 6 7 8 9 10 7 11 12 13
8(1)	:	n>m ==> FALSE
14(7)	:	r!=0 ==> TRUE
14(7)	:	r!=0 ==> FALSE

Any arbitrary path can be expressed as a linear combination of the basis paths B1 through B3.

For example, the path P is equal to B2 - 2 * B1 + 2 * B3

Module: Euclid

User Specified Path: 1 Path P

Test Path P	:	0 1 2 3 4 5 6 7 8 9 10 7 8 9 10 7 11 12 13

Notes

8(1)	:	n>m ==> TRUE
14(7)	:	r!=0 ==> TRUE
14(7)	:	r!=0 ==> TRUE
14(7)	:	r!=0 ==> FALSE

This shows the number of times each edge is executed along each path. (See Table 9.1)

Table 9.1: Matrix of Edge Incidence for Basis Paths B1-B3 and Other Path P

Path/ Edge	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
B1	1	0	1	0	0	0	1	1	0	1	0	0	0	1	1
B2	1	1	0	1	1	1	1	1	0	1	0	0	0	1	1
B3	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1
P	1	1	0	1	1	1	1	1	2	1	2	2	2	1	1

One interesting property of foundation sets is that each edge of a flow graph is traversed through at least one path in every foundation set. Put another way, executing a foundation set of paths will always cover every control branch in the module. This income that to cover all edges never requires more than the cyclomatic complexity number of paths. However, executing a basis set just to cover all edges is overkill. Covering all edges can usually be accomplished with fewer paths. In this example, paths B2 and B3 cover all edges without path B1. The relationship between basis paths and edge coverage is discussed.

Note that apart from forming a basis together, there is nothing special about paths B1 through B3. Path P in combination with any two of the paths B1 through B3 would also form a basis. The fact that there are many sets of basis paths through most programs is important for testing, since it means it is possible to have considerable freedom in selecting a basis set of paths to test.

Task Show the relationship of the attribute for metric value?

9.1.3 Limiting Cyclamate Complexity to 10

There are many reasons to limit cyclomatic complexity. Overly complex modules are more prone to error, are harder to understand, are harder to test, and are harder to modify. Deliberately limiting complexity at all stages of software development, for example as a departmental standard, helps avoid the pitfalls associated with high complexity software. Many organizations have successfully implemented complexity limits as part of their software programs. The precise number to use as a limit, however, remains somewhat controversial. The original limit of 10 as proposed by McCabe has significant supporting evidence, but limits as high as 15 have been used successfully as well. Limits over 10 should be reserved for projects that have several operational advantages over typical projects, for example experienced staff, formal design, a modern programming language, structured programming, code walkthroughs, and a comprehensive test plan. In other words, an organization can pick a complexity limit greater than 10, but only if it is sure it knows what it is doing and is willing to devote the additional testing effort required by more complex modules. Somewhat more interesting than the exact complexity limit are the exceptions to that limit.

For example, McCabe originally recommended exempting modules consisting of single multiday decision ("switch" or "case") statements from the complexity limit. The multiday decision issue has been interpreted in many ways over the years, sometimes with disastrous results. Some naive developers wondered whether, since multiway decisions qualify for exemption from the complexity limit, the complexity measure should just be altered to ignore them. The result would be that modules containing several multiday decisions would not be identified as overly complex. One developer started reporting a "modified" complexity in which cyclomatic complexity was divided by the number of multiday decision branches. The stated intent of this metric was that multiday decisions would be treated uniformly by having Complexity limitation affects the allocation of code among individual software modules, limiting the amount of code in any one module, and thus tending to create more modules for the same application. Other than complexity limitation, the usual factors to consider when allocating code among modules are the cohesion and coupling principles of structured design: The ideal module performs a single conceptual function, and does so in a self-contained manner without interacting with other modules except to use them as subroutines. Complexity limitation attempts to quantify an "except where doing so would render a module too complex to understand, test, or maintain" clause to the structured design principles. This rationale provides an effective framework for considering exceptions to a given complexity limit.

Rewriting a single multi-way decision to cross a module boundary is a clear violation of structured design. Additionally, although a module consisting of a single multi-way decision may

Require many tests, each test should be easy to construct and execute. Each decision branch can be understood and maintained in isolation, so the module is likely to be reliable and maintainable. Therefore, it is reasonable to exempt modules consisting of a single multi-way decision statement from a complexity limit. Note that if the branches of the decision statement contain complexity themselves, the rationale and thus the exemption does not automatically apply. However, if all branches have very low complexity code in them, it may well apply. Although constructing "modified" complexity measures is not recommended, considering the maximum complexity of each multi-way decision branch is likely to be much more useful than the average. At this point it should be clear that the multi-way decision statement exemption is not a bizarre anomaly in the complexity measure but rather the consequence of a reasoning process that seeks a balance between the complexity limit, the principles of structured design, and the fundamental properties of software that the complexity limit is intended to control. This process should serve as a model for assessing proposed violations of the standard complexity limit. For developers with a solid understanding of both the mechanics and the intent of complexity limitation, the most effective policy is: "For each module, either limit cyclomatic complexity to 10 or provide a written explanation of why the limit was exceeded."

Task Solve control flow graph with complexity 17.

9.1.4 Examples of Cyclomatic Complexity

Independence of complexity and size

There is a big difference between complexity and size. Consider the difference between the cyclomatic complexity measure and the number of lines of code, a common size measure. Just

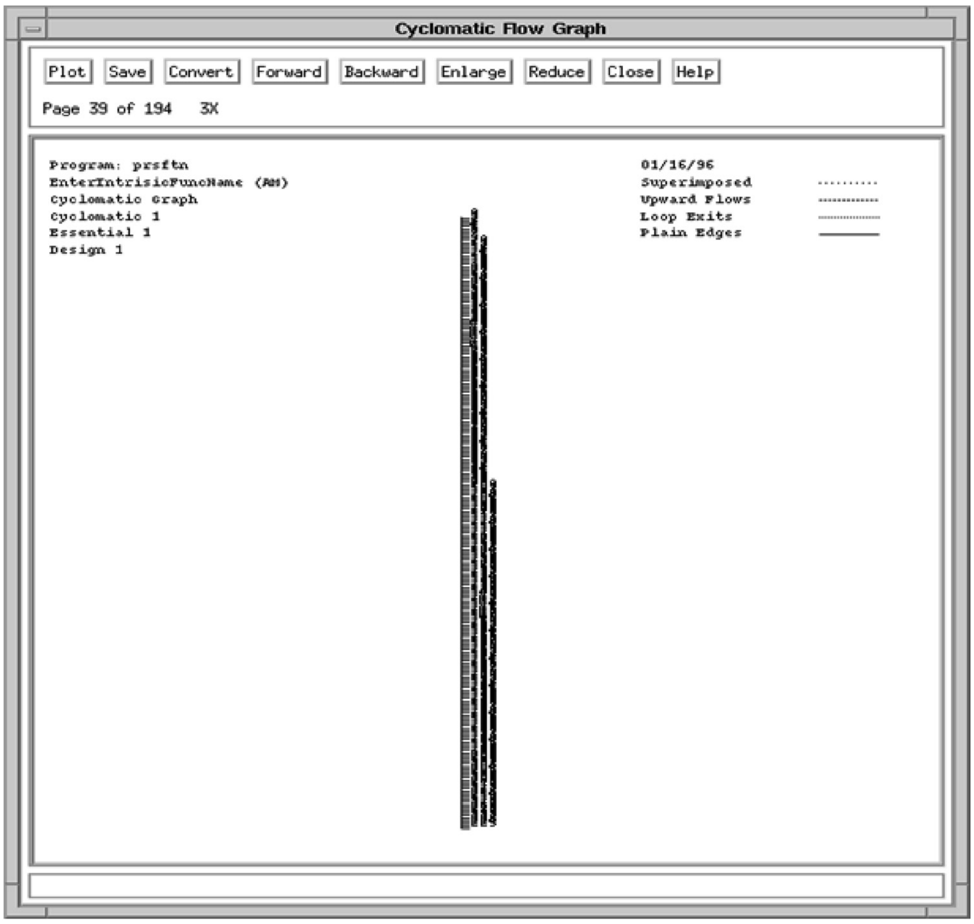
Notes

as 10 is a common limit for cyclomatic complexity, 60 is a common limit for the number of lines of code, the somewhat archaic rationale being that each software module should fit on one printed page to be manageable. Although the number of lines of code is often used as a crude complexity measure, there is no consistent relationship between the two. Many modules with no branching of control flow (and hence the minimum cyclomatic complexity of one) consist of far greater than 60 lines of code, and many modules with complexity greater than ten have far fewer than 60 lines of code. For example, Figure 9.4 has complexity 1 and 282 lines of code, while Figure 9.5 has complexity 28 and 30 lines of code. Thus, although the number of lines of code is an important size measure, it is independent of complexity and should not be used for the same purposes.

Several flow graphs and their complexity

Several actual control flow graphs and their complexity measures are presented in through, to solidify understanding of the measure. The graphs are presented in order of increasing complexity, in order to emphasize the relationship between the complexity numbers and an intuitive notion of the complexity of the graphs. (See Figure 9.6 and Figure 9.7)

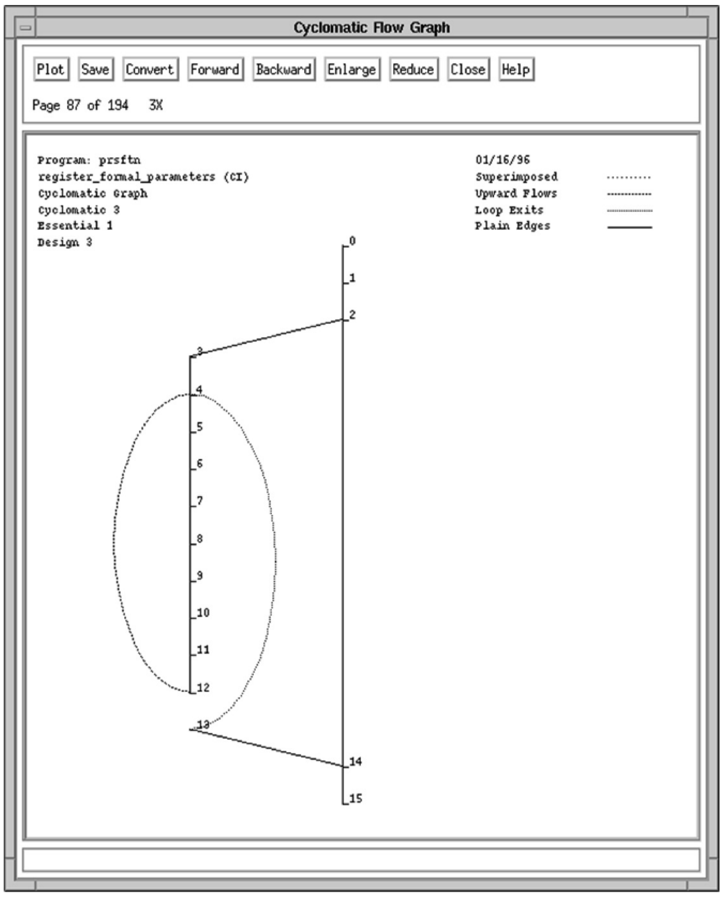
Figure 9.4: Control Flow Graph with Complexity 1



```
Page 39 of 194   3X

Program: prsftn                                01/16/96
EnterIntrisicFuncName (AM)                     Superimposed  ..........
Cyclomatic Graph                               Upward Flows
Cyclomatic 1                                   Loop Exits
Essential 1                                    Plain Edges
Design 1
```

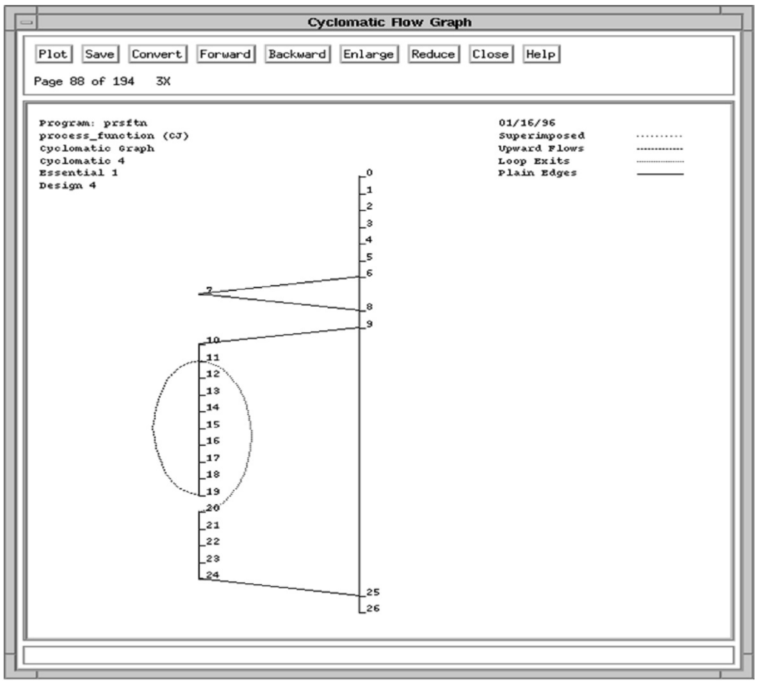
Notes

Figure 9.5: Control Flow Graph with Complexity 3



```
Program: prsftn                              01/16/96
register_formal_parameters (CI)              Superimposed  ..........
Cyclomatic Graph                             Upward Flows  ------------
Cyclomatic 3                                 Loop Exits    ------------
Essential 1                                  Plain Edges   ________
Design 3
```

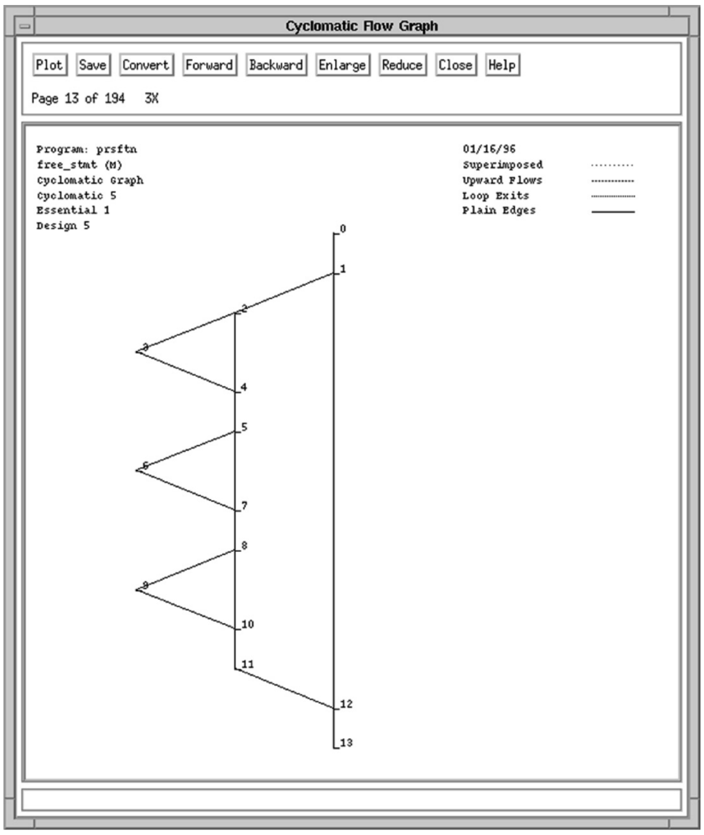
Figure 9.6: Control Flow Graph with Complexity 4



```
Program: prsftn                              01/16/96
process_function (CJ)                        Superimposed  ..........
Cyclomatic Graph                             Upward Flows  ------------
Cyclomatic 4                                 Loop Exits    ------------
Essential 1                                  Plain Edges   ________
Design 4
```

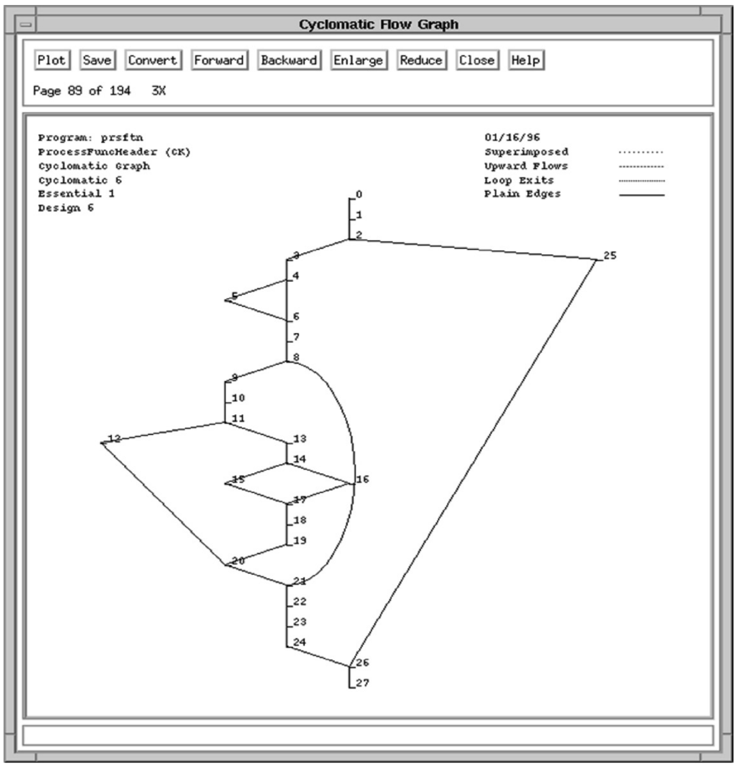
Notes

Figure 9.7: Control Flow Graph with Complexity 5



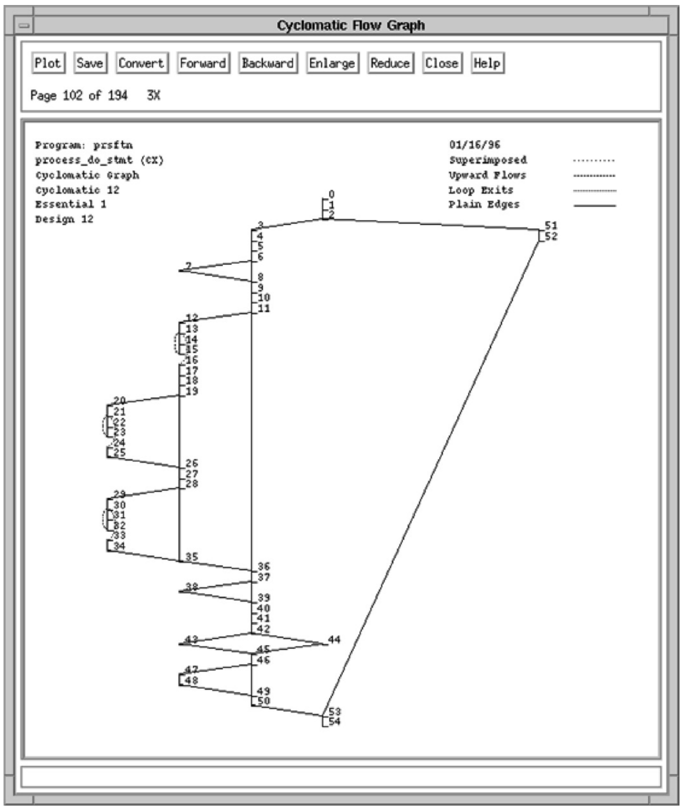
```
Page 13 of 194   3X

Program: prsftn                                   01/16/96
free_stmt (M)
Cyclomatic Graph
Cyclomatic 5
Essential 1
Design 5
```

Figure 9.8: Control Flow Graph with Complexity 6



```
Page 89 of 194   3X

Program: prsftn                                   01/16/96
ProcessFuncHeader (CK)
Cyclomatic Graph
Cyclomatic 6
Essential 1
Design 6
```

Figure 9.9: Control Flow Graph with Complexity 12

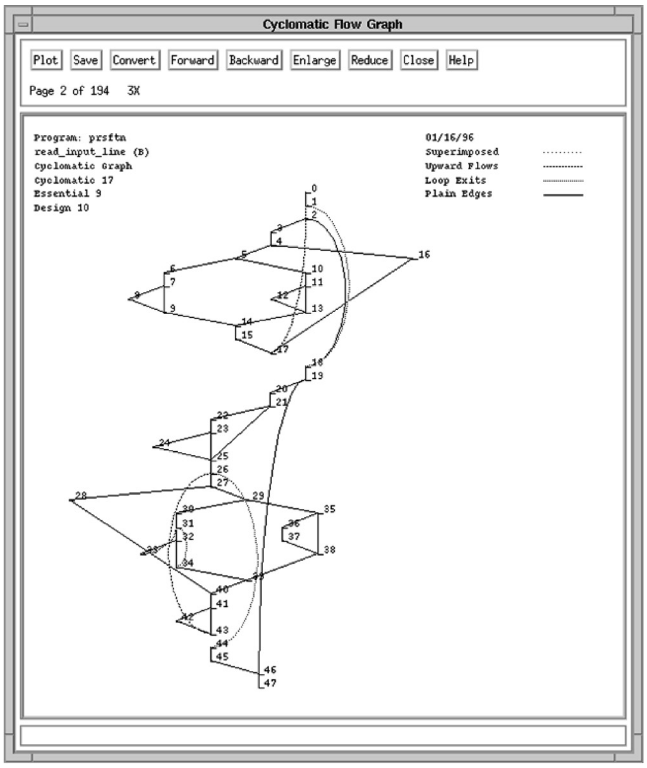


Page 102 of 194 3X

Program: prsftn
process_do_stmt (CX)
Cyclomatic Graph
Cyclomatic 12
Essential 1
Design 12

01/16/96

Figure 9.10: Control Flow Graph with Complexity 17



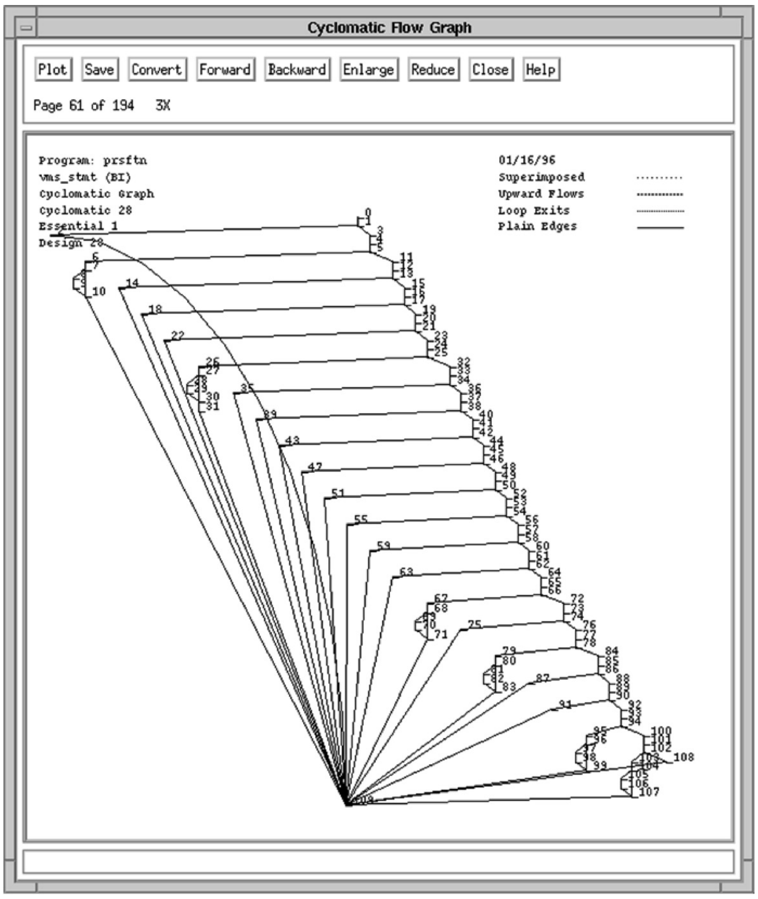
Page 2 of 194 3X

Program: prsftn
read_input_line (B)
Cyclomatic Graph
Cyclomatic 17
Essential 9
Design 10

01/16/96

Notes

Figure 9.11: Control Flow Graph with Complexity 28



One essential ingredient in any testing methodology is to limit the program logic during development so the program can be understood, and the amount of testing required verifying the logic is not overwhelming. A developer who, ignorant of the implications of complexity, expects to verify a module such as that of with a handful of tests is heading for disaster. The size of the module in size of several of the previous graphs exceeded 60 lines, for example the module in. In practice, large programs often have low complexity and small programs often have high complexity. Therefore, the common practice of attempting to limit complexity by controlling only how many lines a module will occupy is entirely inadequate. Limiting complexity directly is a better alternative. (See Figure 9.8 and Figure 9.9)

Caution It is difficult to define or measure software qualities and quantities and to determine a valid and concurrent measurement metric.

9.2 Data Binding

We have seen that coupling and cohesion are significant concepts for evaluating a design. However, to be really effective, metrics are wanted to "measure" the coupling between modules or the cohesion of a module. During system design, we tried to quantify coupling based on information flow between modules. Now that the logic of modules is also available, we can come up with metrics that also consider the logic. One metric that attempts to capture the module-level concept of coupling is data binding. Data bindings are events that capture the data interaction across portions of a software system. In other words, data bindings try to specify how strongly coupled different modules in a software system are. Different types of data bindings are possible [90]. A potential data binding is defined as a triplet (p,x,q), where p and q are modules and x is a variable within the static scope of both p and q. This reflects the possibility that the modules p and q may communicate with each other through the shared variable x. This binding does not consider the internals of p and q to determine if the variable x is actually accessed in any of the modules. This binding is based on data declaration. A used data binding is a potential binding where both p and q use the variable x for reference or assignment. This is harder to compute than potential data binding and requires more information about the internal logic of a module.

An actual data binding is a used data binding with the additional restriction that the module p assigns a value to x and q references x. It is the hardest to compute, and it signifies the situation where information may flow from the module p to module q through the shared variable x. Computation of actual data binding requires detailed logic descriptions of modules p and 4.

All of these data bindings attempt to represent the strength of interconnections among modules. The greater the number of bindings between two modules, the higher the interconnection between these modules. For a particular type of binding, a matrix can be computed that contains the number of bindings between different modules. This matrix can be used for further statistical analysis to determine the interconnection strength of the system.

Caution Always software metrics are based on code syntax, quantifying the complexity or cohesion of an Implementation by performing calculations based on counting code structures.

Self Assessment Questions

1. .... describe the logic structure of software modules.

   (*a*) Flow graph notation (*b*) Cyclomatic complexity

   (*c*) Graph matrices (*d*) Control flow graphs

2. .... can be characterized as the number of elements of a basis set of control flow paths through the module.

   (*a*) Flow graph notation (*b*) Cyclomatic complexity

   (*c*) Graph matrices (*d*) Control flow graphs

3. .... are measures that capture the data interaction across portions of a software system.

   (*a*) Graph matrices (*b*) Potential data binding

   (*c*) Data bindings (*d*) Control flow graphs

4. Rewriting a single multiway decision to cross a module boundary is not a clear violation of structured design.

   (*a*) True (*b*) False

5. The cyclomatic number in graph theory is defined as n + 1.

   (*a*) n + 1 (*b*) n – 1

   (*c*) (n + 1)2 (*d*) n(n + 1)

9.3 Cohesion Metric

Metrics can assist software developers and managers assess the quality of software and pinpoint trouble areas in their systems. For instance, a metric may indicate that a class lacks cohesion. A cohesive class is one in which all of the members are closely related, focused on a single task. A class that lacks cohesion is poorly designed, and therefore is more likely to be error-prone. If metrics are used to quickly and automatically find out which classes in software system lack cohesion, the programming team can take steps to check those classes and improve them before the software has entered the integration and testing stages. Since changes are less expensive the earlier in the development lifecycle they are made this can save the project considerable time and money.

Most software metrics are based on code syntax, quantifying the complexity or cohesion of an Implementation by performing calculations based on counting code structures. In contrast, semantic metrics, introduced by, quantify the meaning within a domain of the task being performed. To collect semantic metrics, first a program understanding system performs understanding; in that way the operation of the software, or what the software does, is Represented in a general knowledge-based format. In our semMet tool, a mature program understanding engine.

The structure of the code itself is not considered for semantic metrics, so it is not necessary for a system to be implemented in order to calculate semantic metrics.

```
if (balance < withdrawal) {
bounce = true;
}
else {
bounce = false;
}
```

Compare that code sample with the following:

```
bounce = (balance < withdrawal) ? true: false;
```

Traditional metrics such as lines of code produce

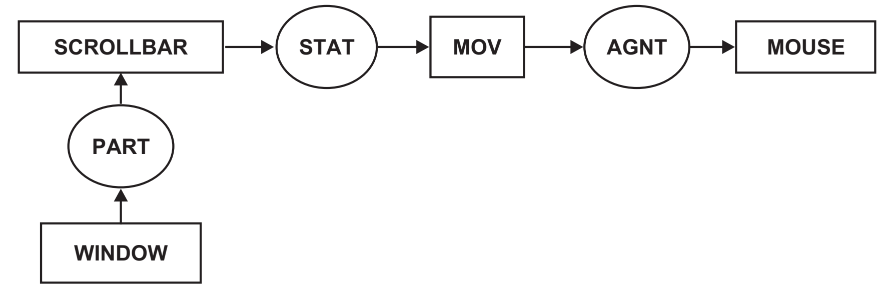
Different values for these two code samples, even though they do exactly the same thing. In contrast, semantic metrics would flag the concepts of balance, withdrawal, and bounce in the banking domain for both samples, regardless of the implementation details.

Semantic metrics are especially useful for measuring cohesion, which is hard to capture based on program syntax, according However, before we can use any metric, we must make sure it is a valid measure of the attribute of interest. Many of the metrics that have been defined have never been adequately validated theoretically or empirically, making their use questionable. Found that many metrics are invalid or poorly defined. They suggested that any new metrics be valid and Unambiguous.

Others have also studied existing metrics and concluded that metrics must be valid and well defined, and too many existing metrics fail in at least one of these are as cohesion metrics and coupling metrics from ten different sources. Found problems with all of these metrics except these studies point to a clear need for valid, well-defined metrics.

PSI (percentage of shared ideas), a knowledge-based semantic cohesion metric. We validate PSI theoretically and empirically, and we compare its performance to six variations on a well-known cohesion metric, Chidambaram and Kemmerer's LCOM. (See Figure 9.12)

Figure 9.12: A Conceptual Graph Example



The Patrician system (and semMet by extension) performs natural language processing on identifiers and comments from code in order to match these words with keywords and concepts in a knowledge base. The Patrician system program understanding engine was originally applied to identifying reusable components in object- oriented software.

SemMet currently consists of two parts: The source code interface and the main processing module. A design specification interface will also be added to facilitate the calculation of semantic metrics from design specifications. The source code interface performs the following steps:

- Count concepts and keywords related to each class and each method of each class.
- Use class- and function-level concept and keyword information to calculate metrics.

The knowledge base used by the semMet system uses the same structure as the knowledge base in the Patricia system This structure consists of two layers: A layer of keywords tagged with part of speech information, and a layer of conceptual graphs Conceptual graphs are a knowledge representation format that can be used to show ideas and the relationships among them In semMet, conceptual graphs are used to represent the relationships among the ideas in the knowledge base. Conceptual graphs are made up of concepts, which represent entities, attributes, states, and events; and conceptual relations, which show how concepts are interconnected For instance, to show "the mouse moves the scrollbar, which is part of the window," we might make a conceptual graph such as the one in. This conceptual graph is read as follows: the scrollbar is part of a window, the state of the scrollbar is moving, and the agent of the scrollbar's moving is the mouse.

Conceptual graphs make up one layer of the knowledge base of the semMet system. The other layer is an interface layer of weighted keywords, which have been tagged with parts of speech. Inference occurs from the interface layer to the conceptual graph layer, and further inference can occur between concepts in the conceptual graph layer.

To calculate semantic metrics using the semMet system, a knowledge base with this structure is created for the domain in which a piece of software is written. The words appearing in the identifiers and comments of a piece of code are compared to concepts and keywords in the knowledge base. Whenever a word from the code matches a keyword in the knowledge base, that keyword is associated with the class or member function.

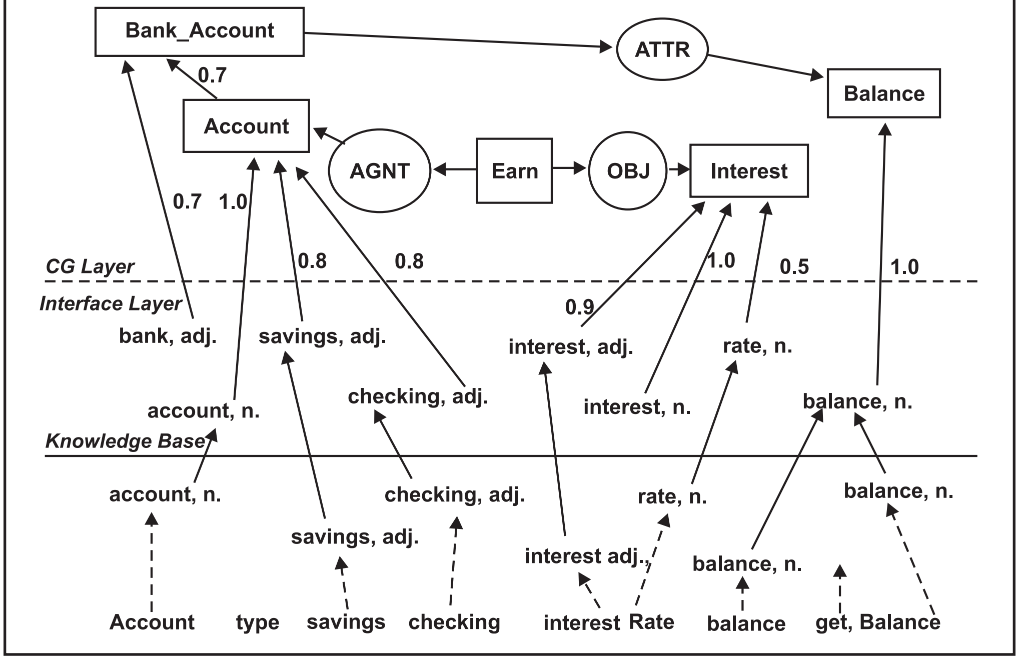
Furthermore, inference is performed from the keyword layer in the knowledge base to the conceptual graph layer. If a class or member function contains keywords which trigger a concept in the conceptual graph layer of the knowledge base, that concept is also associated with the class or member function. As in the Patricia system, semMet's knowledge base and inference engineered implemented in the CLIPS expert system shell Once the appropriate concepts and keywords from the Knowledge base have been associated with each class Retrieve the inheritance hierarchy and each class's attribute variables and member functions. Extract all comments at both class and function levels. Use natural language processing to try to determine the part of speech for each identifier. For example, the function name "getBalance" would become get (verb) and balance (noun). Perform sentence-level natural language processing on comments to determine the part of speech of each word. This task can be accomplished with a high degree of accuracy because comments have their own to illustrate this, consider the abbreviated bank account class example, the identifiers Account, balance, type, interestRate, and getBalance, as well as the comment associated with the class definition, are processed. First multiple-word identifiers such as "interestRate" are split into their component words. Each word is assigned a part of speech. For example, in this case "account" is a noun. Then, the words with their parts of speech are compared against the part-of-speech-tagged keywords in the interface layer of the knowledge base. In this example, the following keywords are matched: Account (noun), Savings (adjective), Checking (adjective), Interest (adjective), Rate (noun), and Balance (noun). The keywords bank (adjective) and interest (noun) are not matched. All of the matched keywords

Notes

are associated with the class Account. In addition, the keyword balance (noun) is associated with the functiongetBalance (), because that function's name contained a match to the keyword. Class. (See Figure 9.13)

```
class Account {
private:
int type; // 0=savings, 1=checking
float interestRate;
float balance;
public:
float getBalance();
```

Figure 9.13: Illustration of semMet Processing a Bank Account Code Example



Then, inference occurs from the interface layer of keywords to the conceptual graph layer. In this case, the weighted links from the account (noun), savings (adjective), and checking (adjective) keywords meet the threshold of 1.0 for the Account concept to be matched. Also, the interest (adjective) and rate (noun) keywords meet the threshold for the Interest concept and the balance (noun) keyword meets the threshold for the Balance concept to be matched. Therefore, the Account, Interest, and Balance concepts are associated with the class Account. Similarly, the Balance concept is associated with the getBalance function, since its name matched the balance (noun) keyword, which met the threshold for the Balance concept. From there, further differencing can occur within the conceptual graph layer. In this case, the link from the Account concept is fired, but it does not have enough weight to meet the threshold for the Bank Account concept, so the Bank Account concept is not matched.

The end result of this example is that the Account class is associated with the Account, Interest, and Balance concepts, and the account (noun), savings (adjective), checking (adjective), interest (adjective), rate (noun), and balance (noun) keywords. The getBalance () function is associated with the Balance concept and the balance (noun) keyword. This information is then used to calculate semantic metrics.

9.3.1 Definition of PSI (Percentage of Shared Ideas)

Although the semantic metrics proposed by are not subject to distortions due to programming language or programmer style as traditional metrics are, they still have one obstacle: they require a knowledge base with a conceptual graph structure in order to be calculated. Not all knowledge bases have such a structure. Therefore, proposed semantic metrics, including PSI, that can be calculated using any knowledge base that associates.

Concepts or keywords with classes and their member functions, regardless of how the knowledge base is implemented.

The PSI is the number of concepts or keywords shared by at least two member functions of a class, divided by the number of concepts or keywords belonging to any member function in the class. PSI for class C*a* is defined as follows.

$$\text{PSI} = \frac{\left|\{x \mid x \in I_a \wedge \exists i, j(x \in I_{ai} \wedge x \in I_{aj})\}\right|}{\left|\{y \mid y \in I_a \wedge \exists k(y \in I_{ak})\}\right|}$$

For 1_ i, j, k _ $|Fa|$, or 0 if no ideas are associated with any function of the class.

For example, define class *Ca* to contain four member functions and a total of ten ideas (concepts or keywords from the knowledge base) associated with member functions to make up the following sets: *Ia1* = {*i3*, *i4*, *i5*}, *Ia2* = {*i4*, *i5*, *i6*}, *Ia3* = {*i1*, *i2*}, and *Ia4* = {*i6*, *i7*, *i8*, *i9*, *i10*}.

In this example, ideas *i4*, *i5*, and *i6* are common to at least two member functions; the others are not. Therefore, PSI for class *Ca* = 3/10 = 0.30.

9.3.2 Syntactic Cohesion Metrics

One of the majority usually cited suites of metrics is that proposed by Chidamber and Kemerer Chidamber and Kemerer proposed these metrics, many people have analyzed, criticized, and proposed their own versions of these metrics. We will compare the performance of PSI to six versions of LCOM (lack of PSI= (1). (See Table 9.2)

Table 9.2: Definition of Various LCOM Matrices

	Definition	Reference
LCOM	LCOM = the set of pairs of member functions with no instance variable used by both members in the pair	[9]
LCOM1	[13] added the constraint that a pair of member functions containing a member function and itself should not be counted. Inherited instance variables are not counted.	[13]
LCOM2	[8] specified a pair of member functions could not include the member function and itself. Inherited instance variables do not count.	[8]
LCOM3	An undirected graph has edges that are pairs of member functions with at least one attribute variable in common. LCOM3 = number of connected components of the graph.	[23] [19]
LCOM4	A variation of LCOM in which there is an edge in the graph for each function that calls another function in addition to the edges for functions that share attribute variables.	[19]
LCOM5	LCOM 5 is specified from the perspective of the number of functions accessing each attribute.	[13]

Notes

9.3.3. Criteria for Evaluating Cohesion Metrics

In this framework, they described the structure of any measure as containing the entities being analyzed, such as classes or modules; the attribute being measured, such as size; the unit used, such as lines of code; and the data scale: nominal, ordinal, interval, or ratio. Units are valid only for interval or ratio data, but they can be adapted for use with ordinal data. In order for a value to have any meaning, the entity, the attribute being measured, and the units must be specified. The measure must be defined over a specified set of permissible values.

In order to be valid, a measure must have:

- Attribute validity: The entity being analyzed has the attribute.
- Unit validity: The unit is appropriate for the attribute.
- Instrumental validity: The underlying model is valid and the instrument was calibrated.
- Protocol validity: The protocol used for the measurement was valid and prevented errors such as double-counting .

Furthermore, in order to be theoretically valid, a direct measure must have the following properties:

- The attribute has different values for different entities.
- The measure works in a way that makes sense with respect to the attribute and its values for different entities.
- Any of the attribute’s units can be used if the attribute is part of a valid measure.
- The attribute can have the same value for different entities.

For an indirect measure, the following properties apply:

- A model of relationships among entities’ attributes is the basis for the measure.
- No improper use of dimensionality occurs in the measure.
- No unexpected discontinuities occur in the measure.
- The units used are appropriate for the scale of data available.

These criteria were proposed to apply specifically to complexity metrics, but some of them are more generally applicable. Of these rejected most but incorporated properties 1, 3, and 4 into their framework. These properties are:

1. There exist different entities with different values.
2. There exist different entities with the same values.
3. There exist entities that perform the same function in different ways and have different values also proposed a set of criteria for metrics. Included in their criteria are some specific properties that cohesion metrics should have. These properties are:
   - Non-negativity and normalization: The value falls in a defined range [0, max]
   - Null value: The value is zero if there are no relations within a module
   - Monotonicity: Adding relations within a module never decreases the value
   - Cohesive modules: The module created by merging two unrelated modules has a value less than or equal to the cohesion value of the more cohesive original module.

Case Study

Software Metrics in an Agile Organization

For any software organization the ability to accurately estimate its projects attributes, such as time, effort, and cost, is a priceless advantage in a highly-competitive global market. In order to attain useful estimates of these attributes, a set of software metrics must be implemented by the organization. For this reason, the appropriate use of quality and productivity metrics embodies one of the most important quality management tools any software organization can have.

On the other hand, over the last decade organizations of all types have replaced their traditional software development life cycles with a new set of so-called agile methodologies, whose main principle is to return to the origins of software engineering where the development tasks were mostly empirical and without much planning and documentation. After almost a decade of the Agile Manifesto's publication and much experimentation, agile methods have gained acceptance as effective software development methodologies and are now widely used in many different types of organizations. For them, however, the problem of successfully implementing software metrics is still present.

From the software measurement standpoint, there is nothing in agile methodologies opposite to measuring quality and performance. By the contrary, the short-iteration, rapid-change-response, rolling-planning, test-driven development nature of these methodologies require constant measurement of projects, processes, and products as to ascertain progress. If "working software is the primary measure of progress" as stated in the Agile Manifesto, then agile organizations need a method to measure completion of its products and determine progress.

Measurement is a primary tool for managing software life cycle tasks, including planning, controlling and monitoring of project plans. This is especially true in agile organizations, where these are "every day" activities. However, not all metrics processes and standards developed over the years for traditional "non-light" lifecycle models can be straightforwardly implemented by agile organizations without some adaptation. They need to follow a different approach in implementing a successful metrics program, particularly if they are small organizations (less than 100 people).

We describe the experience of a small agile software development Organization in starting a metrics program. We provide a description of the design made and the results of the implementation effort. Details are provided on how the ISO/IEC 15939 for software measurement was used as a guideline in an organization that uses Scrum. The contents of this might interest agile organizations.

Software Metrics Standards

There are many standards available to help organizations starting a measurement program. In our case, as our main guideline. This standard, based on the Practical Software Measurement (PSM) methodology developed by McGarry et al. describes a four-phase process for implementing a measurement program. its main outcomes. The process starts by establishing Management commitment and assigning roles and responsibilities to organizational units and individuals. Then, the metrics program is planned by deriving the set of metrics based on business goal. Next, in step three the program is implemented and the first results are obtained. Finally, in step 4 the program is evaluated by validating the implemented metrics and making changes accordingly.

We followed a streamlined version of this process to implement our project since the 15 steps proposed in this standard are too burdensome for small organizations. Besides, the organization was not specifically interested in complying with the standard. In addition,

Contd...

for activity 5.2 Plan the measurement process, we used the Goal/Question/Metric (GQM) methodology originally developed by Basili and Weiss since it has been widely used over more than two decades, it is simple to use, and very effective.

Phase described in the ISO/IEC 15939 standard

Clause number and name	Main outcome
5.1: Establish and sustain measurement commitment	• Management commitment and responsibility, team roles
5.2: Plan the measurement process	• Selected metrics definition
5.3: Perform the measurement process	• Metrics program implementation results
5.4: Evaluate measurement	• Metrics analysis and validation

We also took a look at the IEEE Standard for a Software Quality Metrics Methodology that describes a very similar process to the one defined in the ISO/IEC 15939 standard except that it consists of five phases instead of four. Finally, we also for Software Productivity Metrics as a guideline to identify some productivity metrics.

Related Work

The academic literature is abundant on case studies about implementing metrics programs in software organizations. However, for the purpose of this, our particular interest focused on recent works associated with applying metrics in agile organizations. Many works have been published on this issue. Most recently proposes a quality model and a set of metrics for agile software development organizations and discusses the problem of trying to implement conventional metrics models in agile organizations. In the present a similar experience but using object-oriented metrics as part of their agile methodology. In a similar line of work, presents an exercise in validating three different sets of object-oriented Experience of incorporating software metrics practices into an iterative commercial.

The Organization

Round Box Global (RBX) originated in 2003 in Atlanta, Georgia, USA. Since 2006, the company has a 90-person development team in San Jose, Costa Rica. The organization develops mainly Java web applications for corporate education and training programs. The several development teams are supported by a project management infrastructure along with a quality assurance group.

RBX uses Scrum, one of the best known agile methodologies. Scrum proposes a process model comprising a set of practices and roles that can be used as a guide line to define a software development process. There are three main roles: The Scrum Master maintains the process and acts as project manager, The Product Owner represents the stakeholders and The Team includes all the participating developers.

At each sprint, which lasts between 15 and 30 calendar days, the team develops deliverables software increment. The Product Backlog, a prioritized list of high-level software requirements, defines the functionality to be implemented in each increment.

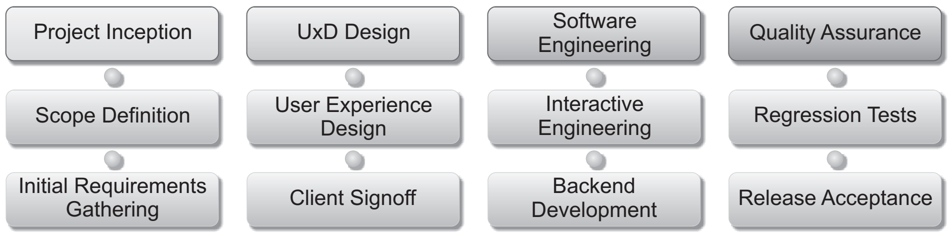
For each sprint, and during the Sprint Planning meeting, the Product Owner requests requirements from this list to the team, whose members determine the quantity of work that can be developed during the sprint? During a sprint, the Sprint Backlog can no be changed thus a requirements baseline can be defined.

Agile organizations that use Scrum generally would have a matrix structure where each project team would have the mixture of people from various Departments with the skills required for each sprint (e.g., project manager, developer, programmer, quality assurance engineer). Each individual in a team would belong to a different but work under a single Scrum Master for a particular project. The organizational process, depicted.

Contd...

The process consists of 4 main phases:

1. Project Inception: Elicits the software requirements and determines the project's goals and scope. Sprints for executing iterations of the following three phases are defined based on this information.
2. UxD Design: Designs the application's human-computer interface.
3. Software Engineering: Consists of implementing the application interface and backend (business and data management layers). This also called develop mentor implementation.
4. Quality Assurance: Performs regression and acceptance testing before customer delivery.



Metrics Program Design

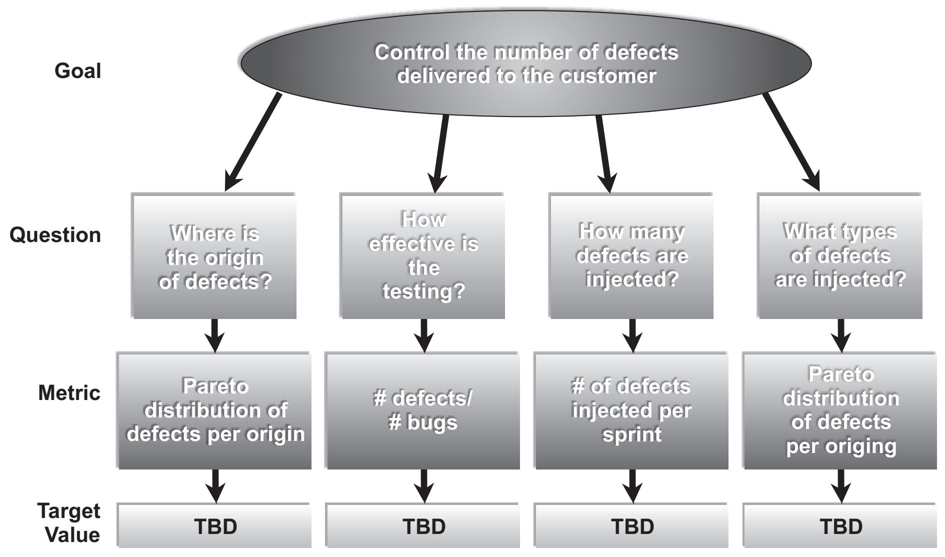
Team members are off course responsible for collecting the data. The Scrum Master is the person responsible for summarizing the metrics data and makes it visible to all team members. Team members are the most important *stakeholders* of the metrics program, thus, the metrics results should be presented to the team members the same way as the user stories (requirements) are discussed and analyzed by the team.

From the metrics standpoint, Management identified two main organizational objectives:

1. Control the number of defects delivered to the customer.

2. Control project time and cost.

Based on these two goals, we used the GQM methodology to build up the Goal/Question/Metrics trees and determine the software metrics based on organizational objectives. Figure 2 shows the ensuing metrics for the defects goal.


Goal
Control the number of defects delivered to the customer
Question
Where is the origin of defects?
How effective is the testing?
How many defects are injected?
What types of defects are injected?
Metric
Pareto distribution of defects per origin
# defects/ # bugs
# of defects injected per sprint
Pareto distribution of defects per origing
Target Value
TBD
TBD
TBD
TBD

Questions

1. How can deign metrics program

2. Describe the related work of matrices

Self Assessment Questions

6. Which one is the category of software metrices?

   (*a*) product matrics (*b*) project metrics

   (*c*) both (*a*) and (*b*) (*d*) None of these

7. Cyclomatic complexity is software metric that provides a ………….. measure of the logical complexity of a program.

   (*a*) quantitative (*b*) qualitative

   (*c*) Both (*a*) and (*b*) (*d*) None of these.

8. Cyclomatic complexity, V (G), for a flow graph, G is defined as

   (*a*) V (G) = E+N+2P (*b*) V (G) = E-N+2P

   (*c*) V (G) = E+N-2P (*d*) None of these.

9. Cyclomatic complexity can be characterized as the number of elements of a basis set of …………..

   (*a*) control flow (*b*) data flow

   (*c*) management flow (*d*) product flow

10. The flow graph depicts logical control flow using a diagrammatic notation

    (*a*) True (*b*) False

11. Cyclomatic complexity, V (G) for a flow graph, G is also defined as V (G) = Pie-1.

    (*a*) True (*b*) False

12. PSI (percentage of shared ideas), a knowledge-based semantic coupling metric.

    (*a*) True (*b*) False

13. Class- and function-level concept and keyword information to calculate metrics.

    (*a*) True (*b*) False

14. Software metrics are based on code syntax, quantifying the complexity or cohesion of an Implementation by performing calculations based on counting code structures.

    (*a*) True (*b*) False

15. Balance concept is associated with the getBalance function, since its name matched the balance (noun) keyword, which met the threshold for the Balance concept.

    (*a*) True (*b*) False

9.4 Summary

- Network metrics for design focus on the structure chart (mostly the call graph component of the structure chart) and define some metrics of how “good” the structure or network is in an effort to quantify the complexity of the call graph.

- In one of the earliest work on information flow metrics, the complexity of a module is considered as depending on the intramodule complexity and the intermodule complexity.
- Software quality metrics focus on the quality aspects of the product, process, and project.
- Implement the defect arrival metric during the last phase of testing, augmented by a qualitative indicator of the number of critical problems (or show stoppers) and the nature of these problems.
- The product is shipped, a natural metric is the number of defects coming in from the field over time.
- When a product size metric is available, the above metrics can be normalized to form other metrics such as defect density curves over time, during testing or when the product is in the field, and overall defect rate of the product, for a specific duration or for the maintenance life of the product.
- Discussion of a few simple and useful metrics as the beginning of a good metrics practice is from the quality perspective. For overall project management.

9.5 Keywords

Cohesion Metrics: Metrics can help software developers and managers assess the quality of software and pinpoint trouble areas in their systems.

Control Flow Graphs: Control flow graphs describe the logic structure of software modules. A module corresponds to a single function or subroutine in typical languages, has a single entry and exit point.

Control Flow Paths: Cyclomatic complexity can be characterized as the number of elements of a basis set of control flow paths through the module.

Cyclomatic Complexity: Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program.

Flow Graph Notation: The flow graph depicts logical control flow using a diagrammatic notation. Each structured construct has a corresponding flow graph symbol.

Case Study

1. Draw the control flow graph with complexity
2. Solve the problem of cohesion metrics with example.

9.6 Review Questions

1. What are the matrices?
2. Explain the cohesion metrics.
3. How many bugs can be expected?
4. Define control flow with complexity.
5. What do you understand by control flow with complexity 12?
6. Differentiate between control flow with complexity 6 and 12.
7. Discuss about control flow with complexity 17 and 27.
8. What are the network metrics?

9. What are the information flow metrics?
10. Differentiate between network metrics and information flow metrics.

Answers for Self Assessment Questions

1.	(*d*)	2.	(*b*)	3.	(*c*)	4.	(*b*)	5.	(*a*)
6.	(*c*)	7.	(*c*)	8.	(*b*)	9.	(*b*)	10.	(*b*)
11.	(*a*)	12.	(*a*)	13.	(*c*)	14.	(*a*)	15.	(*a*)

9.7 Further Reading

Books

Software Engineering and Knowledge Engineering: *Trends for the Next Decade* by William D. Hurley

Online link

http://books.google.co.in/books?id=bDlZa4KvFGgC&pg=PA253&dq=metrics+in+software+engineering&hl=en&sa=X&ei=ze8LUM-yF8O4rAebtYzICA&ved=0CDYQ6AEwAQ#v=onepage&q=metrics%20in%20software%20engineering&f=false

Unit 10: Coding Standards

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the common errors
- Describe the structured programming
- Define programming practices

Introduction

Programmers use far more time reading code than writing code. Over the life of the code, the spends a considerable time reading it throughout debugging and enhancement. People spend considerable attempt in reading code because the code is often maintained by someone. In short, it is of prime importance to write code in a manner that it is easy to read and understand. Coding standards provide method and guidelines for some aspects of programming in order to make code easier to read. Most organizations that develop software regularly develop their own standards. In general, coding standards provide guidelines for programmers regarding naming, file organization, statements and declarations, and layout and comments. To give an idea of coding standards (often called conventions or style guidelines), we discuss some guidelines for Java, based on publicly available standards.

Naming Conventions

- Names of variables in mixed case, beginning with lowercase.

- Function names and type names (classes, structs, enum, and typedef) in mixed case, beginning with uppercase.
- Names of #defines, constants, and enum values in all uppercase, separated with underscores.
- Global variables start with "g".
- Class member variables start with "m_".

Regular Variables

Use mixed case, starting with lowercase. Avoid names that are too short and cryptic, but also avoid extremely long names. Do not use any prefix (e.g. Hungarian notation), except for the global variables and pointers.

Global Variables

Global variables should start with 'g'. The reason is that it is always important to know the scope of a variable, especially for global variables that have dangerously large scope.

Example gParser, gSimulationClock

Class Names

Use mixed case starting with uppercase. There is no reason to prefix class names with "c" or "C" to indicate that they are classes, since in a C++ program most data types should be classes, anyway. Even in the case that you have some non-class types (e.g. struct or enum types), you want to be able to change them to classes easily, without having to modify their names and consequently the client code that uses them. Also note that you do not gain any real programming benefit by embedding in the name the information that something is a class vs. struct or enum type. It is OK (and very often, advisable) to declare variables with similar name as the class:

Example: FluidSet fluidSet;

Try to limit the number of words you use in class names. Compound names of over three words are a hint that your design may be confusing various entities in your system. Revisit your design and try to see if your objects have more responsibilities than they should.

Class Member Variables

Class member variables should start with "m_". This increases code readability as it makes it clear what variables are member variables vs. parameters or locals. Also it allows you to reuse member variable names as function parameters. The 'm_' should always precede other name modifiers like 'p' for pointer.

Example:

```
class Foo {
public:
void Transform(Color color, Flavor flavor) {
m_color = color;
m_flavor = flavor;
}
…
private:
Color m_color;
Flavor m_flavor;
```

Notes

```
String m_pName;
…
};
```

10.1 Common Errors

Software errors (we will use the terms errors, defects and bugs interchange-ably in our discussion here;) are a reality that all programmers have to deal with. Much of effort in developing software goes in identifying and removing bugs. There are various practices that can reduce the occurrence of bugs, but regardless of the tools or methods we use, bugs are going to occur in programs. Though errors can occur in a wide variety of ways, some types of errors are found more commonly. Here we give a list of some of the commonly occurring bugs. The main purpose of discussing them is to educate programmers about these mistakes so that they can avoid them.

```
char* foo(int e)
{
char *output;
if (s>0)
output (char*) malloc (size);
if (s==1) return NULL; /* if s==1 then mem leaked
return(output);
}
```

Similar to NULL dereference is the error of accessing uninitialized memory. This often occurs if data is initialized in most cases, but some cases do not get covered, they were not expected. An example of this error is:

```
switch( i )
{
case 0: s=OBJECT_1; break;
case 1: s=OBJECT_2:break;
}
return (s); /* s not initialized for values other than 0 or 1 */
```

Lack of Unique Addresses

Aliasing creates many problems, and among them is violation of unique addresses when we expect different addresses. For example in the string concatenation function, we expect source and destination addresses to be different. If this is not the case, as is the situation in the code segment, it can lead to runtime errors.

```
strcat(src, destn);
/* In above function, if src is aliased to destn,
* then we may get a runtime error */
```

Task Write a program reduce the common error.

10.1.1 Synchronization Errors

In a parallel program, where there are multiple threads possibly accessing some common resources, then synchronization errors are possible. These errors are very difficult to find as they do not manifest easily. But when they do manifest, they can cause serious damage to the system.

There are different categories of synchronization errors, some of which are:

1. Deadlocks
2. Race conditions
3. Inconsistent synchronization

Deadlock is a situation in which one or more threads mutually lock each other. The most frequent reason for the cause of deadlocks is inconsistent locking sequence the threads in deadlock wait for resources which are in turn locked by some other thread. Race conditions occur when two threads try to access the same resource and the result of the execution depends on the order of the execution of the threads. Inconsistent synchronization is also a common error representing the situation where there is a mix of locked and unlocked accesses to some shared variables, particularly if the access involves updates. Some examples of these errors are given.

Array Index Out of Bounds

Array index often goes out of bounds, leading to exceptions. Care needs to be taken to see that the array index values are not negative and do not exceed their bounds.

Arithmetic Exceptions

These include errors like divide by zero and floating point exceptions. The result of these may vary from getting unexpected results to termination of the program.

10.1.2 Enumerated Data Types

Overflow and underflow errors can easily occur when working with enumerated types, and care should be taken when assuming the values of enumerated data types. An example of such an error is:

```
typedef enum {A, B,C, D:} grade;
void foo(grade x)
{
int 1,m;
1=GLOBAL_ARRAY[x–1]; /* Underflow possible */
m=GLOBAL_ARRAY(x+1); /* Overflow possible */
}
```

10.1.3 String Handling Errors

There are a number of ways in which string handling functions like strcpy, sprintf, gets etc can fail. Examples are one of the operands is NULL, the string is not NULL terminated, or the source operand may have greater size than the destination. String handling errors are quite common.

Buffer overflow

Though buffer overflow is also a frequent cause of software failures, in today's world its main impact is that it is a security flaw that can be exploited by a malicious user for executing arbitrary code. When a program takes an input which is being copied in a buffer, by giving a large (and malicious) input, a malicious user can overflow the buffer on the stack. By doing this, the return address can get rewritten to what-ever the malicious user has planned. So, when the function call ends, the control goes to where the malicious user has planned, which is typically some malicious code to take control of the computer or do some harmful actions. Basically, by exploiting the buffer overflow situation, a malicious user can execute arbitrary code. The following code fragment illustrates buffer overflow:

```
void mygets (char *str) {
        int ch;
        while (ch = getchar () ! ='\n' && ch !='\0')
                        *(str++) = ch;
        *str ='\0';
}
main () {
        char s2 [4] ;
}       mygets (s2) ;
```

10.2 Structured Programming

Structured (or modular) programming techniques shall be second-hand. GOTO statements shall not be old as they lead to "spaghetti" code, which is hard to read and uphold, except as outlined in the FORTRAN Standards and Guidelines.

Structured programming techniques assist the programmer in writing effectual error free programs.

The elements of structured of programming include:

- Top-down development
- Modular design.

The Structure Theorem:

It is possible to write any computer program by using only three (3) basic control structures, namely:

- Sequential
- Selection (if-then-else)
- Repetition (looping, DoWhile)

10.2.1 Algorithms

An algorithm is a sequence of precise instructions for solving a problem in a finite amount of time.

Properties of an Algorithm:

- It must be precise and unambiguous
- It must give the correct solution in all cases
- It must eventually end.

Algorithms and Humans

Algorithms are not a natural way of stating a problem's solution, because we do not normally state our plan of action.

Notes

- We tend to execute as we think about the problem. Hence, there are inherent difficulties when writing an algorithm.
- We normally tailor our plans of action to the particular problem at hand and not to a general problem (i.e. a nearsighted approach to problem solving).
- We usually do not write out our plan, because we are usually unaware of the basic ideas we use to formulate the plan. We hardly think about it – we just do it.
- Computer programmers need to adopt a scientific approach to problem solving, i.e. writing algorithms that are comprehensive and precise.
- We need to be aware of the assumptions we make and of the initial conditions.
- Be careful not to overlook a step in the procedure just because it seems obvious.
- Remember, machines do not have judgment, intuition or common sense!

Developing an Algorithm

- Understand the problem (Do problem by hand. Note the steps)
- Devise a plan (look for familiarity and patterns)
- Carry out the plan (trace)
- Review the plan (refinement)

10.2.2 Understanding the Algorithm

Possibly the simplest and easiest technique to understand the steps in an algorithm, is by means of the flowchart method. This algorithm is composed of block symbols to represent each step in the solution process as well as the directed paths of each step. The most common block symbols are:

Symbol	Representation		Symbol	Representation
▢	Start/Stop		◇	Decision
▭	Process		○	Connector
▱	Input/Output		↓	Flow Direction

There are many other block symbols, used in flow charting, but for the purposes of this course, we will restrict our usage to the symbols described above. They are sufficient to illustrate the steps in developing solutions to the simple problems we will be dealing with.

10.2.3 Top-Down Design Approach (Modularization)

The modularization move toward involves breaking a problem into a set of sub-problems, followed by breaking every sub-problem into a set of tasks, then breaking each task into a set of actions.

Example:

Problem 1

Add 23 and 35

No further refinement is required.

Notes

Problem 2

Turn on a light bulb

Sub-problem 1: locate bulb (one task, one action)

Sub-problem 2: depress switch

Example:

Given a list of students' test scores, find the highest and lowest score and the average score.

Sub-problem 1: read students' scores

Sub-problem 2: find highest score

Sub-problem 3: find lowest score

Sub-problem 1 can be considered as one action and therefore wants no further refinement. Sub problems 2 and 3 however can be further divided into a group of actions. This is left as an exercise for the student.

Advantages of the Top-Down Design Method

- It is easier to comprehend the solution of a smaller and less complicated problem than to grasp the solution of a large and complex problem.
- It is easier to test segments of solutions, rather than the entire solution at once. This method allows one to test the solution of each sub-problem separately until the entire solution has been tested.
- It is often possible to simplify the logical steps of each sub-problem, so that when taken as a whole, the entire solution has less complex logic and hence easier to develop.
- A simplified solution takes less time to develop and will be more readable.
- The program will be easier to maintain.

Did u know? A partial formalization of the concept of algorithm began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928.

Self Assessment Questions

1. .... errors are very difficult to find as they do not manifest easily.

   (*a*) Software (*b*) Synchronization

   (*c*) String handling (*d*) Warning

2. An ....is a sequence of precise instructions for solving a problem in a finite amount of time.

   (*a*) syntax (*b*) program

   (*c*) algorithm (*d*) none of these

3. Advantages of the top-down design method:

   (*a*) The program will be easier to maintain

   (*b*) It must be precise and unambiguous (*c*) It must eventually end

(*d*) None of these

4. ……………..techniques assist the programmer in writing effective error free programs.

   (*a*) Computer programming (*b*) Modular programming

   (*c*) Numerous programming (*d*) Structured programming

5. Top-down design method is easier to comprehend the solution of a smaller and less complicated problem than to grasp the solution of a large and complex problem.

   (*a*) True (*b*) False

10.3 Programming Practices

Experienced developers go after numerous programming practices or method of thumb, which typically derived from hard-learned. The practices listed are not all-inclusive, and should not be used without due consideration. Veteran programmers deviate from these practices on occasion, but not with no careful consideration of the potential repercussions. Using the best programming practice in the wrong context can cause more harm than good.

- To conserve resources, be selective in the choice of data type to ensure the size of a variable is not excessively large.
- Keep the lifetime of variables as short as possible when the variables represent a finite resource for which there may be contention, such as a database connection.
- Keep the scope of variables as small as possible to avoid confusion and to ensure maintainability. Also, when maintaining legacy source code, the potential for inadvertently breaking other parts of the code can be minimized if variable scope is limited.
- Use variables and routines for one and only one purpose. In addition, avoid creating multipurpose routines that perform a variety of unrelated functions.
- When writing classes, avoid the use of public variables. Instead, use procedures to provide a layer of encapsulation and also to allow an opportunity to validate value changes.
- When using objects pooled by MTS, acquire resources as late as possible and release them as soon as possible. As such, you should create objects as late as possible, and destroy them as early as possible to free resources.
- When using objects that are not being pooled by MTS, it is necessary to examine the expense of the object creation and the level of contention for resources to determine when resources should be acquired and released.
- Use only one transaction scheme, such as MTS or SQL Serve, and minimize the scope and duration of transactions.
- Be wary of using ASP Session variables in a Web farm environment. At a minimum, do not place objects in ASP Session variables because session state is stored on a single machine. Consider storing session state in a database instead.
- Stateless components are preferred when scalability or performance are important. Design the components to accept all the needed values as input parameters instead of relying upon object properties when calling methods. Doing so eliminates the need to preserve object state between method calls. When it is necessary to maintain state, consider using alternative methods, such as maintaining state in a database.

- Be specific when declaring objects, such as ADODB. Recordset instead of just Recordset, to avoid the risk of name collisions.
- Require the use Option Explicit in Visual Basic and VBScript to encourage forethought in the use of variables and to minimize errors resulting from typographical errors.
- Avoid the use of variables with application scope.
- Use RETURN statements in stored procedures to help the calling program know whether the procedure worked properly.
- Use early binding techniques whenever possible.
- Use Select Case or Switch statements in lieu of repetitive checking of a common variable using If…Then statements.
- Explicitly release objects references.

Data-Specific

- Never use SELECT *. Always be explicit in which columns to retrieve and retrieve only the columns that are required.
- Refer to fields implicitly; do not reference fields by their ordinal placement in a Recordset.
- Use stored procedures in lieu of SQL statements in source code to leverage the performance gains they provide.
- Use a stored procedure with output parameters instead of single-record SELECT statements when retrieving one row of data.
- Verify the row count when performing DELETE operations.
- Perform data validation at the client during data entry. Doing so avoids unnecessary round trips to the database with invalid data.
- Avoid using functions in WHERE clauses.
- If possible, specify the primary key in the WHERE clause when updating a single row.
- When using LIKE, do not begin the string with a wildcard character because SQL Server will not be able to use indexes to search for matching values.
- Use WITH RECOMPILE in CREATE PROC when a wide variety of arguments are passed, because the plan stored for the procedure might not be optimal for a given set of parameters.
- Stored procedure execution is faster when you pass parameters by position (the order in which the parameters are declared in the stored procedure) rather than by name.
- Use triggers only for data integrity enforcement and business rule processing and not to return information.
- After each data modification statement inside a transaction, check for an error by testing the global variable @@ERROR.
- Use forward-only/read-only recordsets. To update data, use SQL INSERT and UPDATE statements.
- Never hold locks pending user input.
- Use uncorrelated subqueries instead of correlated subqueries. Uncorrelated subqueries are those where the inner SELECT statement does not rely on the outer SELECT statement for information. In uncorrelated subqueries, the inner query is run once instead of being run for each row returned by the outer query.

Notes

ADO-Specific

- Tune the RecordSet.CacheSize property to what is needed. Using too small or too large a setting will adversely impact the performance of an application.
- Bind columns to field objects when looping through recordsets.
- For Command objects, describe the parameters manually instead of using Parameters. Refresh to obtain parameter information.
- Explicitly close ADO Recordset and Connection objects to insure that connections are promptly returned to the connection pool for use by other processes.
- Use adExecuteNoRecords for non-row-returning commands.

Superior coding techniques and programming practices are hallmarks of a professional programmer. The bulk of programming consists of making a large number of small choices while attempting to solve a larger set of problems. How wisely those choices are made depends largely upon the programmer's skill and expertise.

The document addresses some fundamental coding techniques and provides a collection of coding practices from which to learn. The coding techniques are primarily those that improve the readability and maintainability of code, whereas the programming practices are mostly performance enhancements.

The readability of source code has a direct impact on how well a developer comprehends a software system. Code maintainability refers to how easily that software system can be changed to add new features, modify existing features, fix bugs, or improve performance. Although readability and maintainability are the result of many factors, one particular facet of software development upon which all developers have an influence is coding technique. The easiest method to ensure that a team of developers will yield quality code is to establish a coding standard, which is then enforced at routine code reviews.

- Coding Standards and Code Reviews
- Coding Techniques
- Best Practices
- Conclusion
- Suggested Reading

10.3.1 Coding Standards and Code Reviews

A comprehensive coding standard encompasses all aspects of code construction and, while developers should exercise prudence in its implementation, it should be closely followed. Completed source code should reflect a harmonized style, as if a single developer wrote the code in one session. At the inception of a software project, establish a coding standard to ensure that all developers on the project are working in concert. When the software project will incorporate existing source code, or when performing maintenance upon an existing software system, the coding standard should state how to deal with the existing code base.

Although the primary purpose for conducting code reviews throughout the development life cycle is to identify defects in the code, the reviews can also be used to enforce coding standards in a uniform manner. Adherence to a coding standard can only be feasible when followed throughout the software project from inception to completion. It is not practical, nor is it prudent, to impose a coding standard after the fact.

Notes

Caution Avoid the use of forced data conversion, sometimes referred to as variable coercion or casting, which may yield unanticipated results.

10.3.2 Coding Techniques

Coding techniques incorporate many facets of software development and, although they usually have no impact on the functionality of the application, they contribute to an improved comprehension of source code. For the purpose of this document, all forms of source code are considered, including programming, scripting, mark-up, and query languages.

The coding techniques defined here are not proposed to form an inflexible set of coding standards. Rather, they are meant to serve as a guide for developing a coding standard for a specific software project.

The coding techniques are divided into three sections:

- Names
- Comments
- Format

Names

Perhaps one of the most influential aids to understanding the logical flow of an application is how the various elements of the application are named. A name should tell "what" rather than "how." By avoiding names that expose the underlying implementation, which can change, you preserve a layer of abstraction that simplifies the complexity. For example, you could use GetNextStudent() instead of GetNextArrayElement().

A tenet of naming is that difficulty in selecting a proper name may indicate that you need to further analyze or define the purpose of an item. Make names long enough to be meaningful but short enough to avoid being wordy. Programmatically, a unique name serves only to differentiate one item from another. Expressive names function as an aid to the human reader; therefore, it makes sense to provide a name that the human reader can comprehend. However, be certain that the names chosen are in compliance with the applicable language's rules and standards.

Following are recommended naming techniques:

Routines

- Avoid elusive names that are open to subjective interpretation, such as Analyze() for a routine, or xxK8 for a variable. Such names contribute to ambiguity more than abstraction.
- In object-oriented languages, it is redundant to include class names in the name of class properties.
- Use the verb-noun method for naming routines that perform some operation on a given object, such as CalculateInvoiceTotal().
- In languages that permit function overloading, all overloads should perform a similar function. For those languages that do not permit function overloading, establish a naming standard that relates similar functions.

Variables

- Append computation qualifiers (Avg, Sum, Min, Max, Index) to the end of a variable name where appropriate.
- Use customary opposite pairs in variable names, such as min/max, begin/end, and open/close.

- Since most names are constructed by concatenating several words together, use mixed-case formatting to simplify reading them. In addition, to help distinguish between variables and routines, use Pascal casing (CalculateInvoiceTotal) for routine names where the first letter of each word is capitalized. For variable names, use camel casing (documentFormatType) where the first letter of each word except the first is capitalized.
- Boolean variable names should contain Is which implies Yes/No or True/False values, such as fileIsFound.
- Avoid using terms such as Flag when naming status variables, which differ from Boolean variables in that they may have more than two possible values. Instead of documentFlag, use a more descriptive name such as documentFormatType.
- Even for a short-lived variable that may appear in only a few lines of code, still use a meaningful name. Use single-letter variable names, such as i, or j, for short-loop indexes only.
- If using Charles Simonyi's Hungarian Naming Convention, or some derivative thereof, develop a list of standard prefixes for the project to help developers consistently name variables. For more information, see "Hungarian Notation."
- For variable names, it is sometimes useful to include notation that indicates the scope of the variable, such as prefixing a g_ for global variables and m_ for module-level variables in Microsoft Visual Basic.
- Constants should be all uppercase with underscores between words, such as NUM_DAYS_IN_WEEK. Also, begin groups of enumerated types with a common prefix, such as FONT_ARIAL and FONT_ROMAN.

Tables

- When naming tables, express the name in the singular form. For example, use Employee instead of Employees.
- When naming columns of tables, do not repeat the table name; for example, avoid having a field called EmployeeLastName in a table called Employee.
- Do not incorporate the data type in the name of a column. This will reduce the amount of work needed should it become necessary to change the data type later.

Microsoft SQL Server

- Do not prefix stored procedures with sp_, because this prefix is reserved for identifying system-stored procedures.
- In Transact-SQL, do not prefix variables with @@, which should be reserved for truly global variables such as @@IDENTITY.

Miscellaneous

- Minimize the use of abbreviations. If abbreviations are used, be consistent in their use. An abbreviation should have only one meaning and likewise, each abbreviated word should have only one abbreviation. For example, if using min to abbreviate minimum, do so everywhere and do not later use it to abbreviate minute.
- When naming functions, include a description of the value being returned, such as GetCurrentWindowName().
- File and folder names, like procedure names, should accurately describe what purpose they serve.

- Avoid reusing names for different elements, such as a routine called ProcessSales() and a variable called iProcessSales.
- Avoid homonyms when naming elements to prevent confusion during code reviews, such as write and right.
- When naming elements, avoid using commonly misspelled words. Also, be aware of differences that exist between American and British English, such as color/colour and check/cheque.
- Avoid using typographical marks to identify data types, such as $ for strings or % for integers.

Comments

Software documentation exists in two forms, external and internal. External documentation is maintained outside of the source code, such as specifications, help files, and design documents. Internal documentation is composed of comments that developers write within the source code at development time.

One of the challenges of software documentation is ensuring that the comments are maintained and updated in parallel with the source code. Although properly commenting source code serves no purpose at run time, it is invaluable to a developer who must maintain a particularly intricate or cumbersome piece of software.

Following are recommended commenting techniques:

- When modifying code, always keep the commenting around it up to date.
- At the beginning of every routine, it is helpful to provide standard, boilerplate comments, indicating the routine's purpose, assumptions, and limitations. A boilerplate comment should be a brief introduction to understand why the routine exists and what it can do.
- Avoid adding comments at the end of a line of code; end-line comments make code more difficult to read. However, end-line comments are appropriate when annotating variable declarations. In this case, align all end-line comments at a common tab stop.
- Avoid using clutter comments, such as an entire line of asterisks. Instead, use white space to separate comments from code.
- Avoid surrounding a block comment with a typographical frame. It may look attractive, but it is difficult to maintain.
- Prior to deployment, remove all temporary or extraneous comments to avoid confusion during future maintenance work.
- If you need comments to explain a complex section of code, examine the code to determine if you should rewrite it. If at all possible, do not document bad code—rewrite it. Although performance should not typically be sacrificed to make the code simpler for human consumption, a balance must be maintained between performance and maintainability.
- Use complete sentences when writing comments. Comments should clarify the code, not add ambiguity.
- Comment as you code, because most likely there would not be time to do it later. Also, should you get a chance to revisit code you have written, that which is obvious today probably would not be obvious six weeks from now.
- Avoid the use of superfluous or inappropriate comments, such as humorous sidebar remarks.
- Use comments to explain the intent of the code. They should not serve as inline translations of the code.

- Comment anything that is not readily obvious in the code.
- To prevent recurring problems, always use comments on bug fixes and work-around code, especially in a team environment.
- Use comments on code that consists of loops and logic branches. These are key areas that will assist the reader when reading source code.
- Separate comments from comment delimiters with white space. Doing so will make comments stand out and easier to locate when viewed without color clues.
- Throughout the application, construct comments using a uniform style, with consistent punctuation and structure.

Format

Formatting makes the logical organization of the code stand out. Taking the time to ensure that the source code is formatted in a consistent, logical manner is helpful to yourself and to other developers who must decipher the source code.

Following are recommended formatting techniques:

- Establish a standard size for an indent, such as four spaces, and use it consistently. Align sections of code using the prescribed indentation.
- Use a monospace font when publishing hard-copy versions of the source code.
- Except for constants, which are best expressed in all uppercase characters with underscores, use mixed case instead of underscores to make names easier to read.
- Align open and close braces vertically where brace pairs align, such as:

  ```
  for (i = 0; i < 100; i++)
  {
   …
  }
  ```

  You can also use a slanting style, where open braces appear at the end of the line and close braces appear at the beginning of the line, such as:

  ```
  for (i = 0; i < 100; i++)
  {
   …
  }
  ```

  Whichever style is chosen, use that style throughout the source code.
- Indent code along the lines of logical construction. Without indenting, code becomes difficult to follow, such as:

  ```
  If … Then
  If … Then
  …
  Else
  …
  End If
  Else
  …
  End If
  ```

Indenting the code yields easier-to-read code, such as:

```
If … Then
    If … Then
        …
    Else
        …
    End If
Else
    …
End If
```

- Establish a maximum line length for comments and code to keep away from having to scroll the source code window and to allow for clean hard-copy presentation.
- Use spaces before and after most operators when doing so does not alter the intent of the code. For example, an exception is the pointer notation used in C++.
- Put a space after each comma in comma-delimited lists, such as array values and arguments, when doing so does not alter the intent of the code. For example, an exception is an ActiveX Data Object (ADO) Connection argument.
- Use white space to provide organizational clues to source code. Doing so creates "paragraphs" of code, which aid the reader in comprehending the logical segmenting of the software.
- When a line is broken across several lines, make it obvious that the line is incomplete without the following line.
- Where appropriate, avoid placing more than one statement per line. An exception is a loop in C, C++, Visual J++, or JScript, such as for (i = 0; i < 100; i++).
- When writing HTML, establish a standard format for tags and attributes, such as using all uppercase for tags and all lowercase for attributes. As an alternative, adhere to the XHTML specification to ensure all HTML documents are valid. Although there are file size trade-offs to consider when creating Web pages, use quoted attribute values and closing tags to ease maintainability.
- When writing SQL statements, use all uppercase for keywords and mixed case for database elements, such as tables, columns, and views.
- Divide source code logically between physical files.
- In ASP, use script delimiters around blocks of script rather than around each line of script or interspersing small HTML fragments with server-side scripting. Using script delimiters around each line or interspersing HTML fragments with server-side scripting increases the frequency of context switching on the server side, which hampers performance and degrades code readability.
- Put each major SQL clause on a separate line so statements are easier to read and edit, for example:

```
SELECT FirstName, LastName
FROM Customers
WHERE State = 'WA'
```

- Do not use literal numbers or literal strings, such as For i = 1 To 7. Instead, use named constants, such as For i = 1 To NUM_DAYS_IN_WEEK, for ease of maintenance and understanding.

Notes

- Break large, complex sections of code into smaller, comprehensible modules.

Caution Do not open data connections using a specific user's credentials. Connections that have been opened using such credentials cannot be pooled and reused, thus losing the benefits of connection pooling.

Case Study

Redwood Software Inc. expands its Report2Web Burster to integrate PDF document processing

Brief Portrait

Redwood Software Inc. was founded in 1993. Its product range includes software solutions in the field of business intelligence and controlling, among others. Redwood Software provides its customers with direct added value in the form of a high level of automation in reporting processes.

Report2Web is an innovative, web-based solution that provides safe and professional content reporting and archiving within the company. It enables any system to make any kind of electronic document available throughout the corporation safely, speedily, reliably and on time.

Customer Application Requirements

The highly flexible and reliable functionality of the Report2Web solution splits large numbers of related, corporation-wide reports in PDF format and merges them with new content to form new documents. The program must be able to handle incoming reports with different formats.

It should be possible to extract individual reports from the overall reporting system and make them available to executive staff in individual areas of the company as part of the management process. The program should also guarantee a smooth internal and external document flow, thus forming the basis for safely archiving the reports in the archiving system used by the customer.

Solution Concept

PDF Prep Tool Suite and the PDF Library Software Development Kit (SDK) enable highly flexible PDF document splitting. The solution ensures that no content is lost and that readability and availability is guaranteed at all times.

Benefit

By utilizing PDF Prep Tool Suite and the PDF Library SDK from PDF Tools AG, Redwood Software was able to significantly increase the performance scope of Report2Web Burster for processing PDF documents. The technology from PDF Tools AG enables the customer to use the intelligent processing ability of Report2Web Burster for reports not only for ASCII texts but also for PDF formats. This ability has made Report2Web Burster a central processing module in the Reporting product family of Redwood Software.

Questions

1. Explain the customer application requirements.

2. Explain the Report2Web.

Self Assessment Questions

6. The XHTML specification to ensure all ……….. documents is valid.

   (*a*) HTML (*b*) XML

   (*c*) Java Script (*d*) None of these.

7. All forms of source code are considered in the …………. including programming, scripting, mark-up, and query languages.

   (*a*) database (*b*) networking

   (*c*) software project (*d*) None of these.

8. The primary purpose for conducting code reviews throughout the development life cycle is to identify defects in the ………………..

   (*a*) design (*b*) code

   (*c*) document (*d*) design

9. The application, construct comments using a uniform style, with consistent punctuation and……………..

   (*a*) structure (*b*) software development life cycle

   (*c*) programming language (*d*) None of these.

10. File and folder names, like procedure names, should accurately describe what purpose they serve.

    (*a*) True (*b*) False

11. The comments are used to explain the intent of the code.

    (*a*) True (*b*) False

12. An abbreviation should have only one meaning and likewise, each abbreviated word should have many abbreviation.

    (*a*) True (*b*) False

13. A name of the project should tell "what" rather than "how."

    (*a*) True (*b*) False

14. Stored procedure execution is faster when you pass parameters by position rather than by name.

    (*a*) True (*b*) False.

15. Use a stored procedure with output parameters instead of single-record SELECT statements when retrieving one row of data.

    (a) True (b) False.

10.4 Summary

- Coding standards provide rules and guidelines for some aspects of programming in order to make code easier to read.
- Software errors are a reality that all programmers have to deal with.

- A software engineer's responsibility is to produce software that is a business asset and is going to last for many years.
- Algorithms are not a natural way of stating a problem's solution, because we do not normally state our plan of action.
- Possibly the simplest and easiest method to understand the steps in an algorithm, is by using the flowchart method.
- Possibly the simplest and easiest method to understand the steps in an algorithm, is by using the flowchart method.
- The modularization approach involves breaking a problem into a set of sub-problems.
- The bulk of programming consists of making a large number of small choices while attempting to solve a larger set of problems.
- The coding techniques defined here are not proposed to form an inflexible set of coding standards.

10.5 Keywords

Algorithms: These are not a natural way of stating a problem's solution, because we do not normally state our plan of action.

Bulk of Programming: It consists of making a large number of small choices while attempting to solve a larger set of problems.

Class Member Variables: It should start with "m_" and increases code readability as it makes it clear what variables are member variables vs. parameters or locals.

Coding Techniques: In the incorporate many facets of software development and, although they usually have no impact on the functionality of the application, they contribute to an improved comprehension of source code.

Comprehensive Coding: The standard encompasses all aspects of code construction and, while developers should exercise prudence in its implementation.

Deadlock: It is a situation in which one or more threads mutually lock each other.

Overflow and Underflow Errors: It can easily occur when working with enumerated types, and care should be taken when assuming the values of enumerated data types.

Lab Exercise

1. Write the method of program structured.
2. Draw the structure of program practices.

10.6 Review Questions

1. What is the naming convention?
2. Define regular variables and coding techniques.
3. Explain the global variables.
4. What are the class member variables?
5. What do you mean by synchronization errors?
6. What are the string handling errors?
7. Describe the structured programming.

8. Define algorithms and the procedure of developing an algorithm.
9. Explain the top-down design approach (modularization).
10. What do you understand by programming practices and coding standards and code reviews?

Answers of Self Assessment Questions

1. (*b*)	2. (*c*)	3. (*a*)	4. (*d*)	5. (*a*)
6. (*a*)	7. (*c*)	8. (*b*)	9. (*a*)	10. (*a*)
11. (*a*)	12. (*b*)	13. (*b*)	14. (*a*)	15. (*a*)

10.7 Further Readings

Books “*Software Engineering*” By-Sommerville

“*Software Engineering: Theory and Practice*” By- Shari Lawrence Pfleeger, Joanne M. Atlee

Online link http://www.perlmonks.org/?node=919075

Unit 11: Coding Process

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define incremental
- Discuss the test driven
- Explain the pair programming

Introduction

Software coding standards are language-specific programming system that greatly decrease the probability of introducing errors keep on your applications, regardless of which software development model (iterative, waterfall, and so on) is being used to create that application.

Software coding standards originated from the intensive learn of industry experts who analyzed how bugs were generated when code was written and connected these bugs to specific coding practices. They took these correlations between bugs and coding practices and came up with a set of rules that when used prevent coding errors from occurring. Coding standards offer incredible value to software development organizations because they are pre-packaged automated error prevention practices; they close the criticism loop between a bug and what must be done to prevent that bug from reoccurring.

In a squad environment or group collaboration, coding standards ensure uniform coding practices, reducing oversight errors and the time spent in code reviews. When work is outsourced to a third-party contractor, having a set of coding principles in place ensures that the code produced by the contractor meets all quality guidelines mandated by the client company. Coding Standards are not merely a way of enforcing naming conventions on your code.

Coding Standards Enforcement IS static analysis of source code for:

- Certain rules and patterns to detect problems automatically
- Based on the knowledge collected over many years by industry experts
- Virtual code review or peer review by industry respected language experts – automatically.

Standards enforcement includes SEI - CMM and ISO 9001. These efforts failed to deliver on their promise because they created stacks upon stacks of bureaucratic documents. There was no automation of processes– because of this the cost of implementation overwhelms the benefit of process implementation.

The coding process of the software life-cycle is concerned with the development of code that will implement the design. This code is written is a formal language called a programming language. Programming languages have evolved over time from sequences of ones and zeroes directly interpretable by a computer, through symbolic machine code, assembly languages, and finally to higher-level languages that are more understandable to humans.

The goal of the coding phase is to translate the design of the system into code in a given programming language. For a given design, the aim of this phase is to implement the design in the best possible manner. The coding phase affects both testing and maintenance profoundly. A well written code reduces the testing and maintenance effort. Since the testing and maintenance cost of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort. Hence, during coding the focus should be on developing programs that are easy to write. Simplicity and clarity should be strived for, during the coding phase.

The different notations used to communicate algorithms to a computer. A computer executes a sequence of instructions (a program) in order to perform some task. In spite of much written about computers being electronic brains or having artificial intelligence, it is still necessary for humans to convey this sequence of instructions to the computer before the computer can perform the task. The set of instructions and the order in which they have to be performed is known as an algorithm. The result of expressing the algorithm in a programming language is called a program. The process of writing the algorithm using a programming language is called programming, and the person doing this is the programmer.

Machine Language

For the first machines in the 1940s, programmers had no choice but to write in the sequences of digits that the computer executed. For example, assume we want to calculate the absolute value of $A + B - C$, where A is the value at machine address 3012, B is the value at address 3013, and C is the value at address 3014, and then store this value at address 3015.

It should be clear that programming in this manner is difficult and fraught with errors. Explicit memory locations must be written, and it is not always obvious if simple errors are present. For example, at location 02347, writing 101 instead of 111 would compute $|A + B + C|$ rather than what was desired. This is not easy to detect.

Assembly Language

Since each component of a program stands for an object that the programmer understands, using its name rather than numbers should make it easier to program. By naming all locations with easy-to-remember names, and by using symbolic names for machine instructions, some of the difficulties of machine programming can be eliminated. A relatively simple program called an assembler converts this symbolic notation into an equivalent machine language program.

The symbolic nature of assembly language greatly eased the programmer's burden, but programs were still very hard to write. Mistakes were still common. Programmers were forced to think in terms of the computer's architecture rather than in the domain of the problem being solved.

High-level Language

The concept was that if we want to compute |A + B − C|, and store the result in a memory location called D, all we had to do was write D = |A + B − C| and let a computer program, the compiler, convert that into the sequences of numbers that the computer could execute. FORTRAN (an acronym for Formula Translation) was the first major language in this period.

The FORTRAN statements were patterned after mathematical notation. In mathematics the = symbol implies that both sides of the equation have the same value. However, in FORTRAN and some other languages, the equal sign is known as the assignment operator. The action carried out by the computer when it encounters this operator is, 'Make the variable named on the left of the equal sign have the same value as the expression on the right.' Because of this, in some early languages the statement would have been written as −D → D to imply movement or change, but the use of → as an assignment operator has all but disappeared.

11.1 Incremental

An incremental approach postpones detail in some or all phases to produce working software earlier in the project development timescale. The basic idea is to develop the system in a vertical slice rather than a horizontal slab.

Incremental Development: Incremental Development is the growth of a system in a sequence of incomplete products (increments) throughout the project timescale.

Incremental Delivery: Incremental Delivery is the delivery of increments to the customer/users at intervals throughout the project timescale.

Increment: An Increment is a self-contained functional unit of software, together with all supporting material, including:

- Requirements specification
- Design documentation
- Test plans, cases and results
- User manuals and training
- Estimates, plans, schedules, resourcing
- Quality assurance information (e.g. Review reports)
- Configuration management information.

An increment produces (or alters) a cross-section of the final system deliverables, connected to a functional piece of the final system. Incremental development is the construction of a software system in a series of small mini-life-cycles, rather than construction in one large monolithic life cycle.

11.1.1 Incremental Life Cycle Models

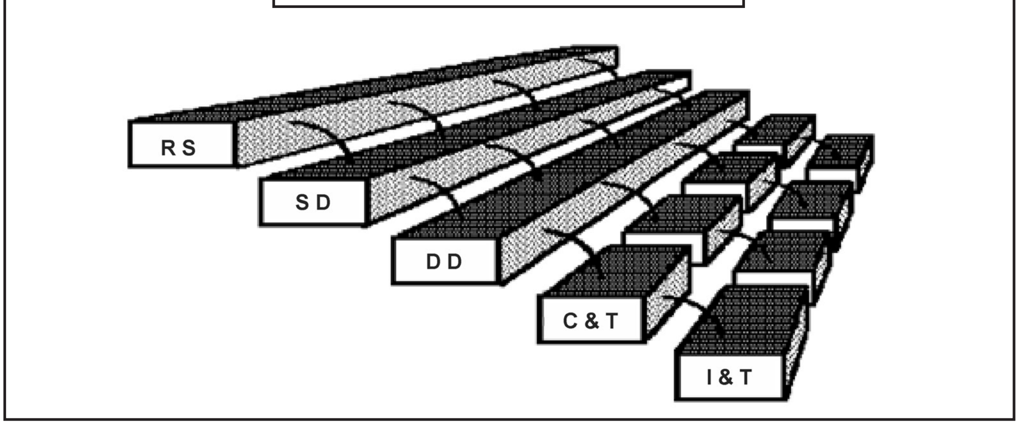
Incremental Build and Test

The incremental build and test approach begins the incremental development in the coding phase, with the phases being monolithic. Many developers go some way toward this approach casually, although often without the complete put of life cycle documentation. Since this is not

Notes

what is recommended by the monolithic waterfall model, the developers may feel culpable that they are not following the model correctly; better results will be obtained from following the incremental build and test approach intentionally rather than accidentally. (See Figure : 11.1)

Figure 11.1: Incremental Build and Test



Evolutionary Delivery

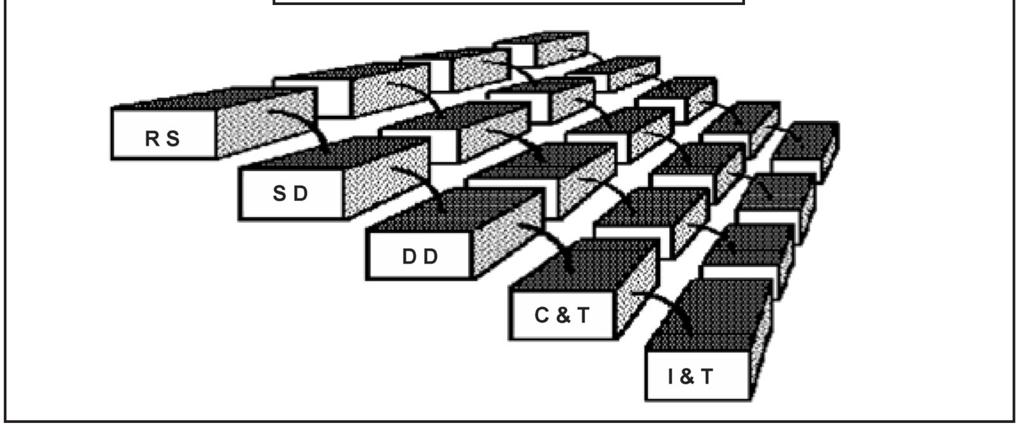
Evolutionary delivery is shown in Figure: 11.2, and is the most extreme incremental approach, defining the increments from the top of the life cycle. Gilb's method includes incremental delivery as well as incremental development, and therefore has helpful working facilities available to the customers much earlier than other life cycle models.

The diagram shown actually does not do justice to the evolutionary delivery method, however, since there is a higher level procedure which precedes the incremental steps, consisting of setting system and business objectives, open architecture design, planning and quality assurance.

The evolutionary deliveries are made at frequent intervals (possibly as small as a week), and consist of some function, facility, or organizational modify which is useful to the customer and relatively easy to produce. In fact that ratio is used to determine the order of the increments.

A major effect of evolutionary delivery is to elicit requests for change, mainly from users. However, these change requests may be 'folded back' into the development process at significantly less cost than for monolithic models, for two reasons. First, change is expected and planned for, so it does not come as an unwelcome surprise. Second, when requirements have been completely detailed and designed (in the monolithic approach), changes which are requested will affect the work already invested in the frozen specification.

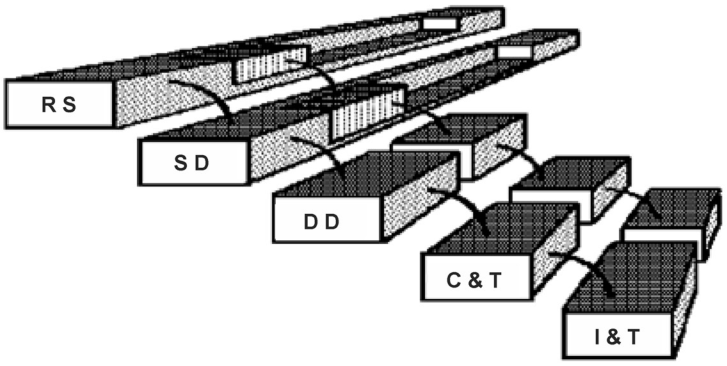
Figure 11.2: Evolutionary Deliveries



Framework Incremental Life Cycle

Just as one extreme being wrong does not imply that the other extreme is right, a framework incremental approach may be the 'best of both worlds', by providing a compromise between the monolithic waterfall and Gilb's evolutionary delivery. Enough of the initial requirements specification and architectural design is done so that the direction and arrangement of the system produced is clear sufficient to direct the software development process. This approach can still give useful products very early in the development timescale. An example of this type of approach is the specification of the structure and interfaces for a database, with detailed facilities to be specified later. (See figure: 11.3)

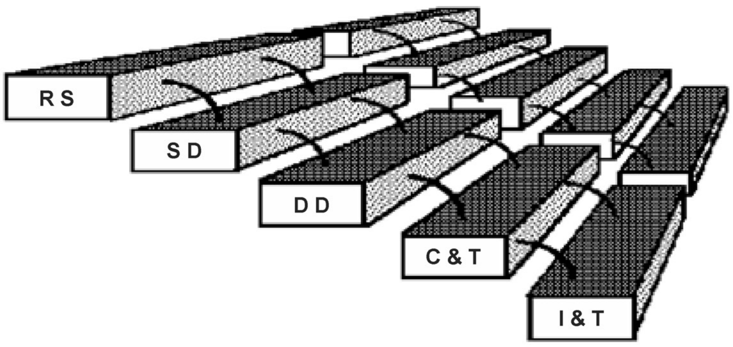
Figure 11.3: Framework Incremental Life Cycle



Phased Development

Phased development has frequently been use in the development of large systems, and is a step in the right direction. The difference between a very small phase and a large increment is not distinct. However, phases tend to be large and growing; there is a tendency to put as much as possible into the current phase. There is also a temptation to compensate for timescale slippage by bringing forward the later phases to overlap the earlier ones. This approach can result in severe incompatibility between successive phase products. The emphasis with incremental development is to include as little as possible, i.e. as little as would be useful into each increment. (See figure: 11.4)

Figure 11.4: Phased Development



Prototyping

Prototyping is usually regarded as the building of a small system (or two) before building the big one. The knowledge gained in either building or using the prototype is then used in building the 'real' system. A good description of the prototyping approach, combined with risk analysis.

'Rapid prototyping' is used to describe the building of systems using software tools such as code generators, report generators, or Fourth Generation Languages (4GL's).

Prototypes are used to reduce risk in the applications area (a novel design, function or performance), or to reduce risk in the user interface area. (The users should then recognise the final system as what they had approved as a prototype.)

However, in a long development timescale, with several years between deliveries of prototypes, the requirements may change extensively, so that it is difficult to relate new requirements to the old prototypes. New hardware capacity, newer user interface styles and graphical capabilities seen on personal computers can make a prototyped user interface seem very old-fashioned.

- Throw-Away Prototype
- Incorporated Prototype
- Incremental Prototype

Did u know? The first programming languages were developed in the late 1950s.

11.2 Test Driven

Test Driven, also known as test first programming or test first development as a new move toward to software development where the programmer have to first write a test that fails before writing any functional code. This practice became popular with the birth of extreme programming (XP).

Test driven development is a programming technique which is applied by following a series of steps repetitively. The first step is to quickly add a test, basically just enough code to fail. next, the tests are run, often the complete test suite although for sake of speed only a subset of the test suite may be chosen, to ensure that the new test does in detail fail. Then the functional code is updated to make it pass the new tests. The next step is running the tests again. If they fail, the functional code needs to be updated to pass the tests. Once the tests pass the next step is to start over.

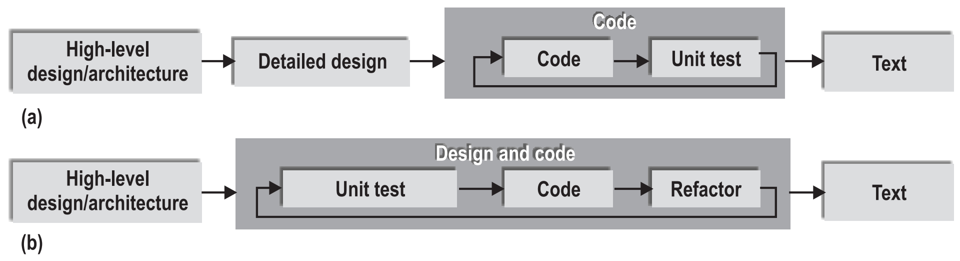
11.2.1 Test Driven Development Methodology

Test Driven Development (TDD) is a very simple methodology that relies on two main concepts: unit tests and refactoring. TDD is basically composed of the following steps:

- Writing a test that defines how a small part of the software should act.
- Making the test run as easily and quickly as possible. Design of the code is not a concern; the sole aim is just getting it to work.
- Cleaning up the code. A step back is taken and any duplication or any other problems that were introduced to get the test to run is refectories and removed.

The TDD is an iterative process, and these steps are repeated a number of times until satisfaction with the new code are achieved. TDD does not rely on a lot of up-front design to determine how the software is structured. The way TDD works is that requirements, or use cases, are decomposed into a set of behaviour that are needed to fulfill the requirement. For each behaviour of the system, the first obsession done is to write a unit test that will test this behaviour. The unit test is written first so that a well-defined set of criteria is formed that can be used to tell when just enough code to implement the behaviour has been written. One of the benefits of writing the test first is that it actually helps better define the behaviour of the system and answer some design questions. (TDD process is shown in figure: 11.5)

Figure 11.5: Development flow: (a) Traditional Test Last and (b) Test-driven Development/test-first Flow



11.2.2 Benefits of Test Driven Development

Test Driven Development contributes to software development practice from many aspects such as requirements definition, writing clean and well designed code, and change and configuration management. The promises of TDD can be summarized as follows;

Simple, Incremental Development: TDD takes a straightforward, incremental approach to the development of software. One of the main benefits to this approach is having a operational software system almost immediately. The first iteration of this software system is very simple and does not have much functionality, but the functionality will get improved as the development continues. This is a less risky approach than trying to build the entire system all at once, hopeful it will work when all the pieces are put together.

Simpler Development Process: Developers who use TDD are more focused. The only thing that a TDD developer has to worry about is getting the next test to pass. The goal is focusing the attention on a small piece of the software, getting it to work, and moving on rather than trying to create the software by doing a lot of up-front design. Thousands of decisions have to be made to create a piece of software. To make all those decisions correctly before starting writing the code is a complex challenge to undergo many times. It is much easier to make those decisions as developing the code.

Constant Regression Testing: The domino effect is well known in software development. Sometimes a simple change to one module may have unforeseen consequences throughout the rest of the project. This is why regression taxing is important. Regression testing is like self-defense against bugs. It is usually done only when a new release is sent to quality assurance (QA). By then it is sometimes hard to trace which code change introduced a particular bug and makes it harder to fix. TDD runs the full set of unit tests every time a change is made to the code, in effect running a full regression test every time a minor change is made. This means any change to the code that has an undesired side effect will be detected almost immediately and be corrected, which should prevent any regression surprises when the software is handed over to QA.

Improved Communication: Communicating the ideas needed to explain how a piece of software should work is not always easy with words or pictures. Words are often imprecise when it comes to explaining the complexities of the function of a software component. The unit tests can serve as a common language that can be used to communicate the exact behaviour of a software component without ambiguities.

Improved Understanding of Required Software Behaviour: The level of requirements on a project varies greatly. Sometimes requirements are very through and other times they are vague. Writing unit tests before writing the code helps developers focus on understanding the required behaviour of the software. As writing a unit test, pass/fail criteria for the behaviour of the software is being added. Each of these pass/fail criteria adds to the knowledge of how the software must behave. As more unit tests are added because of new features or new bugs, the set of unit tests come to represent a set of required behaviours of higher and higher fidelity.

Centralization of Knowledge: Humans all have a collective consciousness that stores ideas they all have in common. Unfortunately, programming is mostly a solitary pursuit. Modules are usually developed by a single individual, and a lot of the knowledge that went into designing the module is usually stuck in the head of the person who wrote the code. Even if it is well documented, clean code, it is sometimes hard to understand some of the design decisions that went into building the code. With TDD, the unit tests constitute a repository that provides some information about the design decisions that went into the design of the module. Together with the source code, this provides two different points of view for the module. The unit tests provide a list of requirements for the module. The source code provides the implementation of the requirements. Using these two sources of information makes it a lot easier for other developers to understand the module and make changes that would not introduce bugs.

Better Encapsulation and Modularity: Encapsulation and modularity help managing the chaos of software development. Developers cannot think about all the factors of a software project at one time. A good design will break up software into small, logical, manageable pieces with well defined interfaces. This encapsulation allows developers concentrate on one thing at a time as the application is built. The problem is that sometimes during the fog of development one may stray from the ideas of encapsulation and introduce some unintended coupling between classes. Unit tests can help detect unencapsulated a module. One of the principles of TDD says that the unit tests should be easy to run. This means that the requirements needed to run any of the unit tests should be minimized. Focusing on making testing easier will force a developer making more modular classes that have fewer dependencies.

Simpler Class Relationships: A well designed piece of software will have well defined levels that build upon each other and clearly defined interfaces between the levels. One of the results of having software that has well defined levels is that it is easier to test. The corollary to this is also true. If code is designed by writing tests, the focus will be very narrow, so the tests will tend not to create complex class relationships. The resulting code will be in the form of small building blocks that fit neatly together. If a unit test is hard to write, then this usually means there is a problem in the design of the code. Code that is hard to test is usually bad code. Since the creation of the unit tests help point out the bad code, this allows correcting the problem and produce better designed, more modular code.

Reduced Design Complexity: Developers try to be forward looking and build flexibility into software so that it can adapt to the ever-changing requirements and requests for new features. Developers are always adding methods into classes just in case they may be needed. This flexibility comes at the price of complexity. It is not that developers want to make the software more complex, it is just that they feel that it is easier to add the extra code up front than make changes later. Having a suite of unit tests allows to quickly tell if a change in code has unforeseen consequences. This will give the developer the confidence to make more radical changes to the software. In the TDD process, developers will constantly be refactoring code. Having the confidence to make major code changes any time during the development cycle will prevent developers from overbuilding the software and allow them to keep the design simple. The approach to developing software using TDD also helps reduce software complexity. With TDD the goal is only adding the code to satisfy the unit tests. This is usually called developing by intention. Using TDD, it is hard to add extra code that is not needed. Since the unit tests are

Notes

derived from the requirements of the system, the end result is just enough code to have the software work as required.

Did u know? Test driven development is the NASA Project Mercury in the 1960's.

Caution Any unmanaged change to the code may be the cause of an undesired side effect in the designed model.

Self Assessment Questions

1. .... is the development of a system in a series of partial products throughout the project timescale.

   (*a*) Incremental delivery (*b*) Increment

   (*c*) Incremental development (*d*) None of these

2. A major effect of ....is to elicit requests for change, mainly from users.

   (*a*) Evolutionary delivery (*b*) Increment test

   (*c*) Incremental build (*d*) Incremental life cycle

3. ………………..has frequently been used in the development of large systems, and is a step in the right direction.

   (*a*) Increment test (*b*) Evolutionary delivery

   (*c*) Incremental builds (*d*) Phased development

4. TDD is basically composed of the steps:

   (*a*) Writing a test that defines how a small part of the software should behave.

   (*b*) Simple, incremental development

   (*c*) Simpler development process

   (*d*) Constant regression testing

5. Encapsulation and modularity are not help managing the chaos of software development.

   (*a*) True (*b*) False

11.3 Pair Programming

Pair programming refers to the practice whereby two programmers work together at one computer, collaborating on the similar design, algorithm, code, or test. The couple is made up of a driver, who actively types at the computer or records a design; and a navigator, who watches the work of the driver and carefully identifies problems, asks clarifying questions, and makes suggestions. Both are also continuous brainstorming associates. Pair programming has been shown to have many of the benefits of reviews while also eliminating the programmer's distaste for reviews so that at least one form of review is actually performed.

Pair programming is a method of programming in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test. In industry, the practice of pair programming has been shown to improve product quality, improve team spirit, aid in knowledge management, and reduce product risk. In education, pair programming also improves student morale, helps students to be more successful, and improves student retention in an information technology major. This topic provides an overview and history of pair programming followed by a summary of the use of pair programming in industry and academia. It is also provides insight into the principles that make pair programming successful, the economics of pair programming, and the challenges in the adoption of pair programming.

11.3.1 The Driver and Navigator

One of the pair, called the driver, types at the computer or writes down a design. The other partner, called the navigator, has many jobs. One of these is to observe the work of the driver—looking for tactical and strategic defects in the driver's work. Some tactical defects might be syntax errors, typos, and calling the wrong method. Strategic defects occur when the driver is headed down the wrong path—what driver and navigator are implementing just would not accomplish what it needs to accomplish. The navigator is the strategic, longer-range thinker of the programming pair. Because the navigator is not as intensely involved with the design, algorithm, code or test, he or she can have a more objective point of view and can better think strategically about the direction of the work.

Another benefit of pair programming is that the driver and the navigator can think at any time the situation calls for it. An effective pair programming relationship is very lively. In an effective pairing relationship, the driver and the navigator continually communicate. Periodically, it is also very important to switch roles between the driver and the navigator.

Pairing During All Phases of Development

The name of the technique, pair programming can lead people to incorrectly assume that we should only pair during code development. However, pairing can occur during all phases of the development process, in pair design, pair debugging, duo testing, and so on. Programmers could pair up at any time during development, in particular when they are working on something that is complex.

Task Draw the incremental life cycle models.

11.3.2 Necessity of Pair Programming

Some people think that having two people sit down to develop one artifact must be a big waste of resources. Managers are especially concerned about this since they think they will have to pay two programmers to do the work one could do. Even students are concerned about this because they think they might have to spend twice as long on their homework. However, some research results show that these concerns do not materialize.

Higher-quality Code

Pairs developed higher quality code faster with only a minimal increase in total time spent in coding. For instance, if one student finished a project in ten hours, the pair might work on it for five and a half hours (for eleven total hours of time between the two). The code produced by the pairs in the study also passed 15% more of the automated test cases, demonstrating that the pairs produce code of higher quality.

Enhanced Morale, Teamwork, and Learning

Pair programming offers additional benefits, including the following:

Increased Morale: Pair programmers are happier programmers. Several surveys were taken of pair programmers in the North Carolina study discussed above. Ninety-two percent of them indicated that they enjoyed programming more when they worked with a partner. Ninety-six percent of them indicated they felt more confident in their product when they worked with a partner.

Increased Teamwork: Pair programmers get to know their classmates much better because they work so closely together.

Enhanced learning: Pairs continuously learn by watching how their partners approach a task, how they use their language capabilities, and how they use the development tools.

11.3.3 How Does Pair Programming Work?

It may seem odd that two people can sit down at one computer and finish in about half the time, with higher quality code, and enhanced morale, teamwork, and learning. But studies have shown that pairing makes work differently.

Pair Pressure

- Programmers say they work harder and smarter on programs because they do not want to let their partner down.
- Programmers say they work very intensively because they are highly motivated to complete the task at hand during the session.

Pair Negotiation

The second of the pair programming behaviours we discuss is one we call pair-negotiation. The term pair negotiation is describe how two pair programmers arrive at the best solution together, when pairing is working at its best:

- Each person brings to the partnership his or her own set of skills, abilities, and outlooks.
- Both share the same goal for completing the task.
- Each person has a suggested alternative for attacking a joint problem.
- The partners must negotiate how to jointly approach the problem. In this negotiation, they evaluate more alternatives than either one would have considered alone, whereas, a person working alone tends to pursue the first approach that comes to mind. Together, the partners consider and include each other’s suggestions and determine the best plan of attack.

Brainstorming

Pair brainstorming describes the give-and-take procedure in which two people work together to resolve a problem. They share their knowledge and energy, chipping steadily away at the problem, evolving a solution to the problem. An effective technique of brainstorm is to build upon the ideas of others. A subtle difference in pair brainstorming though, is that the driver might actually be working out a design or implementing part of the problem, realizing that he or she may ultimately come to a dead end in their problem resolution. The navigator, while watching the driver’s partial design or implementation, also begins thinking about the next step. When the driver hits the dead end, the navigator is often prepared to take over and lead the way. Often, the cycle continues until the problem is determined.

Pair Debugging

Every person has experienced problems that can be determined simply through the act of explaining the problems to another person. Others have cited the need just to have a cardboard cut-out of their favorite guru or a rubber duck. When they have a problem, they must explain the problem to the inanimate helper, often finding the elusive flaw in the logic. Doubtlessly, it is better to have an animate person to debug with than an inanimate helper. A person will ask questions and will likely cause you to frequently explain your reasoning.

Pair Learning

Notes

Knowledge is constantly being approved between partners, from tool practice tips to programming language rules to design and programming techniques. The partners take turns being the teacher and the student on a minute-by-minute basis. Even unspoken skills and habits cross partners.

Case Study

Computer Assisted Coding for Inpatients

Computer-assisted coding (CAC) has often been marketed for its ability to increase productivity. We undertook two studies to determine if productivity for inpatient coding was increased using CAC software. A number of significant problem areas were identified and it was determined that the CAC software did not increase coder productivity.

We have been testing computer-assisted coding (CAC) software since 2002 in order to increase the productivity of our remote coding staff. This topic describes several of our initiatives and the results.

'The term computer-assisted coding is currently used to denote technology that automatically assigns codes from clinical documentation for a human…to review, analyze, and use,' explains Rita Scichilone, MHSA, RHIA, CCS, CCS-P, director of coding products and services at AHIMA.1 We became familiar with this technology and sought a way to harness its potential. We did recognize that the development of such software was not mature, as results of other projects have been reported in the literature. One example of the issues surrounding computer assisted coding (CAC) was provided by Beth Friedman: 'More often, questions were raised about CAC quality and reliability. Results indicated that 29% of respondents are concerned that CAC systems would result in high productivity but poor quality, and 38% expressed concern that these systems were not yet proven.

There are a variety of methodologies employed by developers of CAC software to 'read' text and assign codes. The software can use structured input (SI) or natural language processing (NLP). Even within the NLP range of products, there are a variety of approaches with varying levels of sophistication. The methodology used has a tremendous impact on data transmission and the output reviewed by the coders.

Study Method

Our general study method consisted of selecting transcribed files for 100 patients. The files consisted of the History and Physical report, any consultation report or operative report, and discharge summary.

Not all reports were available for all patients. Both studies focused on inpatients because we felt that if a company had developed software sophisticated enough to provide accurate code assignment for inpatient records, they could easily adapt the software to handle outpatient records.

We reviewed the software product on several criteria:

Accuracy of code assignment

Ease of use (including transmission of transcribed data, initial coder instruction, and functionality of output information). Ability of the software and review process to enhance coder productivity.

Contd...

Pilot 1

Our first pilot involved a vendor whose product assigned codes for inpatients and outpatients. The software used SI coding. In order for the software to 'read' the clinical terms to be codes, the diagnostic or procedural phrase had to start on the same line and at the same column in each different transcribed document. It was not possible for us to meet this requirement with transcription coming from multiple clients with varying report formats. The only alternative was to place delimiters around the text to be read for coding. This proved tedious and time-consuming. It negated any productivity increases by the coders. In addition, the requirement for the strict identification of phrases meant that any diagnostic or procedural information embedded in the body of the report could not be 'read' by the software.

Other problems became apparent during the study:

The software was unable to assign Current Procedural Terminology (CPT) codes, so this eliminated its use for outpatients.

The software was fairly accurate in ICD-9-CM code assignment (approximately 70% accuracy rate), but the list of assigned codes included extensive notes to physicians regarding possible documentation improvement and extensive notes to coders about possible additional or different codes. While interesting the first time they appeared, the notes became overwhelmingly repetitive. Since each note was interspersed with its associated code in the list of codes assigned by the software, it was not possible for a coder to quickly review the codes for accuracy or completeness. The code listing spanned several screens or, if printed out, several pages. There was no way to turn off this feature other than totally eliminating all edits. Because of these operational issues—the pilot was discontinued.

Pilot 2

Our second pilot involved a product that only assigned codes for inpatients. It truly 'read' the text and assigned ICD-9-CM codes for both diagnoses and procedures. It did not sequence codes, but left that task to the coder. Human intervention was required to determine the principal diagnosis. Once a coder made that selection, the software determined the diagnosis related group (DRG). With the selection of additional codes for secondary diagnoses and procedures, the software updated the DRG assignment.

The computer assigned codes appeared in a window. If coders clicked on the code, they were taken to the report and text that served as the basis for the code assignment. If there were multiple text references, the coder would be led to each one for review.

An encoder and grouper were embedded in the CAC product, so coders could 'recode' any diagnostic or procedural term. All activity occurred on one screen through various windows. The CAC software analyzed documentation for 100 patients as called for in our study method.

Results of Pilot 2

1. Coding Accuracy

Codes assigned were accurate, but not always appropriate. The coders did not accept 75% of the diagnosis (dx) codes and 90% of the procedure (px) codes. In 58% of the cases, coders added diagnosis codes. In 45% of the cases, procedure codes were added. The codes required to determine the correct DRG were present in 48% of the cases.

Specific Problem Areas

- The codes assigned by software were listed in numeric order, making review and resequencing tedious.

Contd...

Notes

- The software had difficulty distinguishing between status codes and follow-up codes (V-codes).
- For obstetric codes or other code sections in which the 5th digit needed to be determined as a distinct thought process, the software selected code with an 'x' for the 5th digit or reported back a range of codes.
- The software had difficulty distinguishing between poisonings and adverse drug events, although even human coders have trouble with this. There was no E code (External Cause of Injury Code) selection by the software.
- The software had a problem selecting E codes for accidents. No 'Place of Occurrence' E Codes were assigned.
- The determination of 'history of' conditions versus 'active' conditions also proved problematic for the software.
- The software had difficulty in determining context of 'cervical'—neck versus cervix. It coded Cervical Spine Exam to Examination of cervix.
- There were some misses on code assignment—chronic lower leg edema was coded to unequal leg length.
- Chest pain with radiation was coded as chest pain with radiotherapy.
- Some procedures were not coded at all (such as biventricular pacemaker).
- Some grouping issues were noted (certain principal diagnosis codes would not group to the correct DRG).
- The system did not allow for disposition code effect on DRG. It assumed all discharges were to be assigned to the status of 'home.'
- The system defaulted birth defects to 'congenital' codes.
- The software missed the code for the specific organisms for many infections.
- The software had difficulty dealing with some acronyms and abbreviations or distinguishing an abbreviation from letters imbedded in a word—for example, CAD (academic).
- The software had great difficulty dealing with procedures with multiple components (which parts to bundle and which to code separately).

2. Ease of Use

The software was very intuitive. It required minimal training (less than one/half day) and the coders liked using it.

The software had no problems importing the transcribed documents. No data manipulation required.

3. Enhanced Coder Productivity

Productivity was not enhanced due to the coder needing to research each of the numerous codes returned by the program (average of 20 codes per case). Coder A reported coding two and a half charts per hour. Coder B reported coding four charts per hour. Our productivity range for inpatient is three to five charts coded per hour. Clearly, this software did not enhance productivity.

Contd...

Of note is that the key documents were not present for all patients. The History and Physical report was handwritten or came from another source (physician office, clinic). The Discharge Summary was handwritten or not required (in cases of OB, NB, short-stay patients). A key issue for CAC will be the presence of complete documentation in a form that the software can read.

Discussion

The CAC products we tested did reflect improvements in the technology from software that required structured text or parsed data in order to recognize the phrases to be coded to software that could select key phrases within text to be coded. The software tested second also reflected improved ease in use and sophistication of code selection.

Because of the vast number of codes presented to the coder, the software did not improve productivity as expected. Future versions of CAC software for inpatient documentation need to reduce the number of codes presented to the coder. The software can do this by removing symptoms when related to specific diagnoses and by eliminating the unspecified and specified forms of the same code. It needs to be 'smart' enough to assign fifth digits, V-codes and E codes.

Correct coding is more than just following the rules in the code. Billing guidelines also determine correct selection and sequencing of codes. At this point, CAC is not a panacea for poor coding. It cannot reduce under coding. In order to do this, the software must recognize all terms and assign the correct code. Currently, the software we tested often missed subsequent documentation or failed to link separated diagnostic or procedural statements that could result in a higher level of ICD-9-CM or CPT code.

The software also cannot reduce over coding. In order to do this, the software must incorporate rules related to unbundling of codes that are included with the main diagnostic or procedural code. It must also eliminate codes that are not being treated (such as history or rule outs).

Questions

• Explain the computer-assisted coding.

• Describes the coding accuracy.

Self Assessment Questions

6. During software life cycle, which activity generally consumes the maximum effort?

   (*a*) Design (*b*) Maintenance

   (*c*) Testing (*d*) Coding

7. Which is not a software life cycle model?

   (*a*) Spiral Model (*b*) Waterfall Model

   (*c*) Prototyping Model (*d*) Capability maturity Model

8. SRS stands for ………..

   (*a*) Software requirement specification (*b*) software requirement solution

   (*c*) system requirement specification (*d*) None of these.

9. structured programming codes includes?

   (*a*) sequencing (*b*) alteration

   (*c*) iteration (*d*) All of these.

10. An important aspect of coding is
    (*a*) readability (*b*) productivity
    (*c*) to use as small memory space (*d*) None of these.
11. Which of the following is not the characteristic of software?
    (*a*) Software does not wear out (*b*) Software is flexible
    (*c*) Software is not manufactured (*d*) Software is always correct.
12. Requirements can be refined using
    (*a*) the waterfall model (*b*) prototyping model
    (*c*) the evolutionary model (*d*) the spiral mode.
13. Pseudo code can replace:
    (*a*) flowcharts (*b*) structure charts
    (*c*) decision tables (*d*) cause-effect graphs.
14. Which phase is not available in software life cycle?
    (*a*) Coding (*b*) Testing
    (*c*) Maintenance (*d*) Abstraction
15. Rapid prototyping is used to describe the building of systems using software tools such as code generators, report generators, or Fourth Generation Languages.
    (*a*) True (*b*) False

11.4 Summary

- The coding process of the software life-cycle is concerned with the development of code that will implement the design.
- An incremental approach postpones detail in some or all phases to produce working software earlier in the project development timescale.
- An increment produces (or alters) a cross-section of the final system deliverables, related to a functional portion of the final system.
- Test Driven is a new approach to software development where the programmer must first write a test that fails before writing any functional code.
- Pair programming refers to the practice whereby two programmers work together at one computer, collaborating on the same design, algorithm, code, or test.

11.5 Keywords

FORTRAN: It is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing.

Waterfall Model: It is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.

Fourth-Generation Programming Language (4GL): It is a programming language or programming environment designed with a specific purpose in mind, such as the development of commercial business software.

Extreme Programming (XP): It is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements.

Quality Assurance (QA): It refers to the planned and systematic activities implemented in a quality system so that quality requirements for a product or service will be fulfilled.

Lab Exercise

1. Write a program to reverse the position of 'n' integers using C.
2. Write a program to arrange 10 numbers in increasing order.

11.6 Review Questions

1. What is the coding process in software designing?
2. What is the incremental? Define also development of incremental.
3. Explain the incremental life cycle models.
4. What is the evolutionary delivery?
5. Discuss about framework incremental life cycle.
6. What is the test driven? And also define its methodology.
7. What are the benefits of test driven development?
8. Discuss about pair programming in brief.
9. Why pair programming is essential for programmer?
10. Define working of pair programming.

Answers for Self Assessment Questions

1.	(*c*)	2.	(*a*)	3.	(*d*)	4.	(*a*)	5.	(*b*)
6.	(*c*)	7.	(*d*)	8.	(*a*)	9.	(*d*)	10.	(*a*)
11.	(*d*)	12.	(*b*)	13.	(*a*)	14.	(*d*)	15.	(*a*)

11.7 Further Readings

Books

Software Engineering: A Concise Study, by Kelkar

Software engineering: a practitioner's approach, by Roger S. Pressman

Online link

http://www.nws.noaa.gov/oh/hrl/developers_docs/General_Software_Standards.pdf

Unit 12: Refactoring

CONTENTS

Objectives

After studying this unit, you will be able to:

- Define meaning of refactoring
- Understand verification of refactoring
- Define software metrics
- Explain the size of metrics
- Define complexity metrics analysis

Introduction

Refactoring, as a software engineering method used to incrementally get better the design of existing code, is being increasingly adopted in industrial software development.

Software refactoring is a key to agile software maintenance. It not only helps to locate bugs additional easily, but, what is more important, to keep it readable and ready for changes. If applied regularly, it benefits in shorter learning curve and simple accommodation of possible changes. However, the necessary condition for effective refactoring is to ensure its correctness. Any modification that may introduce new bugs to the software is more dangerous than untidy code: it requires additional time wasted on debugging the formerly working code. Unit testing is commonly suggested as the primary method of verification for refactoring. Typical test cases are used to cheek if domain-specific relations in the code persist and bug has been re-injected. Refactoring, however, aim at different goal. Although they have to preserve the software functionality (and the regression tests as well) hut they also introduce new, independent from the business domain properties that must be satisfied or break some of some existing ones. The considerable effort required to adjust the testing suite at every single refactoring indicates that unit tests are inappropriate for refactoring. There is a need for tests suited exclusively for refactoring that would fix the deficiencies of unit tests. In the chapter we present a concept of refactoring tests, which are intended to ease the process of testing the refactoring. These tests are created exclusively for the refactoring purposes and check the properties specific for the transformation being applied. In subsequent sections we describe the requirements for them and their suggested implementation.

12.1 Meaning of Refactoring

Refactoring is typically applied at the level of programs (i.e., source code). A program refactoring is a program transformation that improves the design of a program while preserving its behaviour. Martin Fowler defines a Refactoring as:

- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour

He then continues to define the verb to Refractor as:

- To restructure software by applying a series of Refactoring without changing its observable behaviour.

Refactoring is an significant component of the Extreme Programming software engineering methodology. In this context, Refactoring are often associated with:

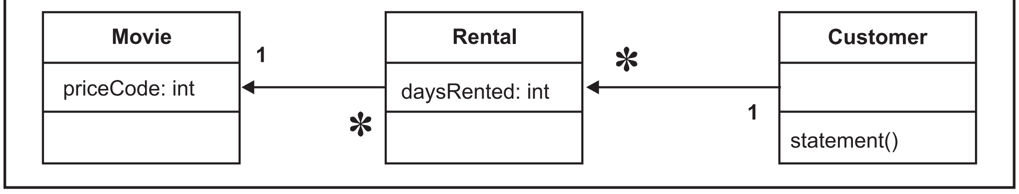
- Many small changes applied repeatedly
- A catalogue of widely discussed changes
- Explicit unit tests applied before and after each minimal change
- interactive, perhaps tool-supported, but not automated.

Refactoring is the procedure of converting existing “bad” code into “good” code. It is done via the parser, the relational model, and the language module. Refactoring can rebuild the damaged structure of the code, and can even fix errors made by developers, as An interesting concept. Worth exploring is incremental Refactoring where one small portion of code is refectory at a time, leaving the rest unchanged or with only the minimum changes needed to accommodate the “bad” parts. A development method with incremental refactoring would have developers always working with hilly refectory code. That is, with code that has been machine-generated from the relational model. The developers add some new code, and immediately submit their work to the model. The model refractors the new code into the existing code and reports the result. If the new code is compatible, then the step is complete and development can continue. If not the model should explain why not and suggest alternatives, or offer an option to reactor the entire program. The key issue is that the model knows the canonical logic and uses it as a reference point before moving into new ground, in contrast with blank development where there is no stable reference point and the structure becomes a moving target.

12.1.1 Refactoring Basic Examples

The sample program is very easy. It is a program to computer and print a statement of a customer's charges at a video store. The program is told which movies a customer rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type movie. There are three kinds of movies: regular, children's, and new releases. In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release. Several classes represent various video elements. Here is a class diagram to show them Figure 12.1.

Figure 12.1: Class Diagram (Movie+Rental+Customer)



The code for each of these classes in turn is shown Movie

Movie is just a simple data class.

```
public class Movie (
public static final in CHILDRENS = 2;
public static final nit REGULAR = 0;
public static final int NEW RELEASE = I;
private String _title
private int _priceCode;
public Movie(String title, int priceCode)
{__price Code = priecCode;
public int getPriceCode() { return _priceCode;
public void setPriceCode(intarg)
{ _priecCode = arg: title = title;
public String get Title return _title:
```

Rental

The rental class represents a customer renting a movie:

```
class Rental
 private Movie _movie;
 private int claysRented;
public Rental(Movie movie, int daysRented)
{ movie = movie;
clavsRented = daysRented;
 public int getDaysRentedu {
 return _davsRented,
public Movie getMovieu()
{ return _movie,
} };
```

Notes

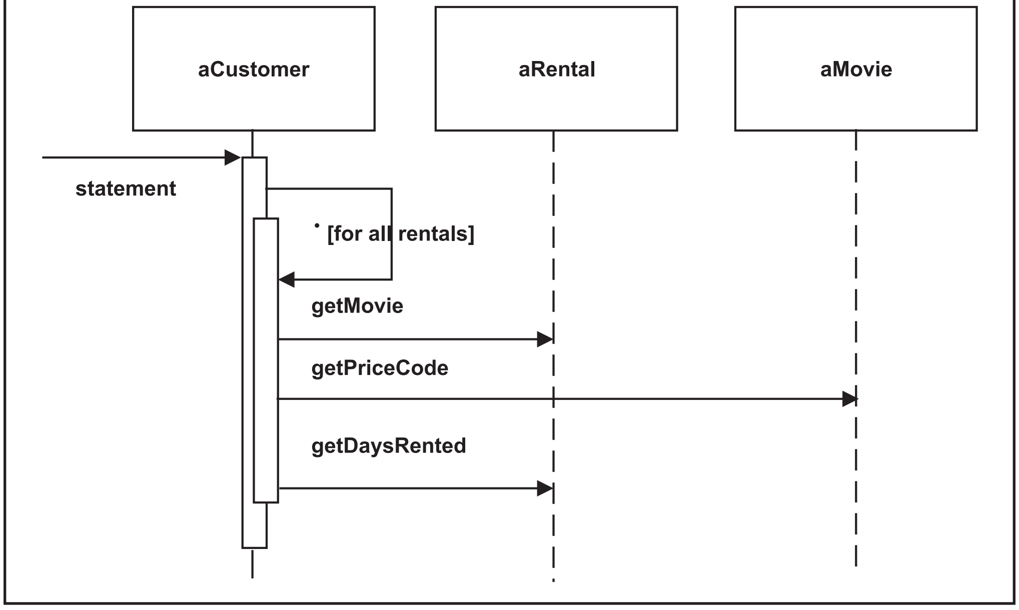
Customer

The customer class represents the customer of the store. Like the other classes it has data and assessors:

```
class Customer{
private String _name;
private Vector _rentals = new Vector();
public Customer (String name)
{ name = _name;
public void addRental(Rental arg) {
_ rentals.addElement(arg);
public String getName ( )
{ return name:
};
```

Customer also has the method that produces a statement. Figure 12.2 shows the interactions for this method. The body for this method is on the facing page.

Figure 12.2: Interactions for the Statement Method



```
public String statement()
{double total Amount = ();
int frequentRenterPoints = ();
Enumeration rentals = Jentals.elements ();
String result = "Rental Record for" + getName () + "\n";
While (rentals.hasMoreElements ())
double thisAmount = 0;
Rental each = (Rental) rentalsmeNtElcment ();
//determine amounts for each line switch
```

```
(each.getMovieagetPriceCode ()){
case Movie.REGUI,AR:
thisAmount += 2;
if (each.getDaysRented()> 2) thisAmount += (each.getDaysRented()- 2)*
1.5: break; case Movie.NEW RELEASE: thisAmount += each.
getDaysRented() * break,
 case Movie.CHILDRENS;
 thisAmount += 1.5;
if (each.getDaysRented() > 3)
thisAmount += (each. getDaysRented() - 3) * 1.5: break;
// add frequent renter points frequentRenterPoints ++;
 // add bonus for a two day new release rental if ((each.
getMovieagetPriceCode()
==Movie.NEW_RELEASE) && each.getDaysRented() > I)
frequentRenterPoints ++;
//show figures for this rental result += "\t" + each.getMovieagetTitle0+
"\t" + String.value0f (thisAmount) + "1n";
totalAmount += thisAmount;
//add footer lines result += "Amount owed is" + String.
value0f(totalAmount) + "111";
Result += "You earned" + String.valueof(frequentRenterPoints) + "
frequent renter points";
return result;
};
```

12.1.2 Bad Smells in Code

What qualities do we expect in good software? It has been suggested that we should aim at developing programs that are easy to read, that have all logic specified in only one place, that allow modifications without endangering existing behaviour, and whose conditional logic is expressed as easily as possible Programs that do not have those qualities smell bad.

The mother of all sins in programming is Duplicated Code. It is easy to see why it makes software maintenance a nightmare: you need to make (nearly) the same modifications to many places and it is hard to know when you are done with them. Naturally code duplication increases also the amount of code making systems harder to understand and maintain.

Another major source for bad smells is the organization of classes and methods. They can be too big and complex (Large class, Long Method, Long Parameter List) or too small and trivial (Lazy Class, Data Class). Lack of classic modularity qualities of loose coupling between structures and cohesion within them may also cause bad smells (Inappropriate Intimacy, Feature Envy, and Data Clumps). Other sources for bad smells include using too much or too little delegation (Message Chains, Middle Man) and using non-object-oriented control or data structures (Switch Statements, Primitive Obsession).

If you think about the original version of the movie rental example, you may notice several bad smells. There are instances of Data Class (Movie, Rental), Long Method (statement) and Switch Statements just to name a few.

Did u know? The term appears to have been coined by Kent Beck on WardsWiki in the late 1990s.

12.1.3 Documenting Refactoring

The power of refactoring as a method lies in systematically organized documents, which describe proven techniques for enhancing software safely. In refactoring: improving the Design of existing code. A refactoring catalogue was used a standard format to represent over 70 frequently needed refactoring.

Each refactoring has five parts:

- ***Name:*** identifies the refactoring and helps to build a common vocabulary for software developers.
- ***Summary:*** tells when and where you need the refactoring and what it does. Summary helps you to find a relevant refactoring in a given situation. It also includes source code excerpts or UML diagrams to show a simple before and after scenario.
- ***Motivation:*** describes why the refactoring should be done and lists circumstances in which it should not be used.
- The mechanics part provides a step-by-step description of how to carry out the refactoring. The steps are as brief as possible to make them easy to follow.

Examples illustrate how the refactoring can be employed in real programs. The catalogue includes refactoring for composing methods and handling local variables (e.g. Extract Method, Inline Method, Inline Temp, Replace Temp with Query) as well as for moving features and organizing data (Move Method, Move Field, Hide Delegate, Replace Data Value with Object, Replace Type Code with State/Strategy). The catalogue also discusses conditional expressions (Replace Conditional with Polymorphism, Introduce Null Object), object creation (Replace Constructor with Factory Method) and generalization (Replace Inheritance with Delegation, Form Template Method).

Most of the refactoring are quite simple and their name pretty much reveals their intention. In general, most refactoring are tiny steps that transform traditional procedural designs into more object-oriented ones. The many insightful discussions are the main substance of the catalogue, in addition to the concrete instructions on how to refractor in various situations.

As the names of some refactoring suggest, refactoring are quite closely related to design patterns. A design pattern tells you how to solve a recurring design problem in a disciplined manner in a given context. Refactoring, on the other hand, guides you to enhance your existing implementation so that it reflects a better design. Often this "better design" is a design pattern. So, in many cases, you end up applying a set of refactoring to turn a piece of ad hoc code into an instance of a design pattern.

Refactoring are very useful in developing efficient and flexible application frameworks and they fit well to the iterative framework development process. Refactoring as a technique plays also a major part in extreme programming.

12.1.4. Basic Techniques behind Refactoring

How do we introduce high-quality qualities to our software and remove bad smells? One of the basic procedures of refactoring (besides eliminating duplication) is adding indirection.

Indirection in its most fundamental form mean defining structures (e.g. classes and methods) and giving them names. Using named structures makes code easy to read because it gives you a way to explain intention (class and method names) and implementation (class structures and method bodies) separately. The same technique enables sharing of logic (e.g., methods invoked in different places or a method in super class shared by all subclasses). Sharing of logic, in turn, helps you to manage change in systems. Finally, polymorphism (another form of indirection) provides a flexible, yet clear way to express conditional logic.

Like in most techniques, the key to success with indirection is to put just the right amount of it in the right spot. Too much or badly placed indirection results in a fragmented system. Useless indirection is often found in a component that used to be shared or polymorphic, but is not anymore due to changes made during the development process. A refactoring catalog gives you the starting point for deciding where and how much indirection should be used. Basically, if you encounter indirection that is not paying for itself, you need to take it out.

12.1.5 Refactoring in Software Development Process

Refactoring essentially means improving the design after it has been implemented. It is an inherently iterative method, which implies that it does not fit very well to the traditional waterfall model of software engineering process. With refactoring design occurs continuously during development.

Refactoring can be thought of as an alternative to careful upfront design. This kind of speculative design is an attempt to put all the good qualities into the system before any code is written. The problem with this process is that it so easy to guess wrong. Sometimes extreme programming is regarded as a paradigm which reacts to that observation by skipping the design phase altogether.

Refactoring can, however, be used also in a more conservative way. Instead of abandoning the design phase completely, you move from overly flexible and complex think-about-everything-beforehand designs to simpler ones. This is sensible because you do not need to anticipate all changes in advance. With refactoring, you can favour simplicity in design because design changes are inexpensive.

When you develop software using refactoring, you divide your time between two distinct activities: Refactoring and adding function. When you add function you should not change existing code. If there is a turn in your development that seems difficult because your implementation does not support the new feature very well, you need to first refractor your system to accommodate the modification.

After adding a function you must add tests to see that the function was implemented correctly. You can even consider tests as explicit requirement documentation for the feature and write them before you add it. Use existing tests to ensure that refactoring did not change behaviour or introduce bugs.

Testing should be as easy as possible. That is why all tests should be made automatic and they should check their own results. This will enable you to test as often as you compile. When you encounter a failure, you can concentrate your debugging in a narrow area of code you added after the last successful tests.

The testing technique that works best with refactoring is class-level white-box testing. A good way to write such unit tests for an object-oriented system is to have a separate test class for each important production class. You can also use a test framework (e.g. Joint) to handle the test case management and reporting in a standardized way. The test framework should provide flexible ways to combine tests so that they can be run in any order.

Testing should be risk driven. This means that you test those parts of a class that are complex and most valuable. Remember to concentrate your testing on the boundaries of the input range and other special conditions (e.g. missing input, null values).

Besides testing, code reviews are known to be valuable in verifying software quality. Refactoring process can be an integral part of reviews, especially if they are conducted in small groups. Refactoring gives you a chance to see the concrete effect of suggested corrections.

Caution Do not try to test everything or you might end up testing nothing.

12.1.6 Benefits of Refactoring

Refactoring can be used for quite a few purposes. First of all, refactoring helps the code to retain its shape. Without refactoring the design of the program will decay. As people change code (usually without fully understanding the design objectives behind the implementation) it gradually begins to lose its structure. Once the structure gets cluttered, the code becomes harder to understand and so the chances of cluttering the design further increase.

Refactoring makes your code more readable. This is essential for conveying the intention of the code to others. It also makes the code easier to read for you. That is equally important since it is unrealistic to assume that you can remember your intentions for more than few weeks.

We can also use refactoring to grasp the intention of unfamiliar code. When looking at a fragment of code you try to understand what it does. When you find out how the code works, you refractor it to better reflect your understanding of its purpose. After that you can test if the system still behaves as it should. If everything goes well, you have understood and processed a part of the system correctly. If not, you need to get a better understanding of the code fragment at hand.

The main advantage of refactoring is that it helps you develop software more quickly. It is rather easy to believe that refactoring improves quality and readability of code, but how does it speed up development? You would think that all modifications and iterative nature of refactoring, not to mention the big effort put into testing, would make your development slow.

The secret is that adding new functions and finding bugs is very efficient when you work on a system with a solid design you understand well. Without refactoring your system will begin to decay from the very beginning. That is why it does not take long for the benefits of keeping your implementation in line with the design to overweight the overhead of refactoring and associated testing. If you ignore refactoring you end up very soon in a situation where you have trouble inserting new functions because of unwanted and unexpected side effects. Similarly, modifications will become harder because you begin to have instances of duplicated code.

12.1.7 Problems with Refactoring

Although very beneficial in many cases, refactoring is not always easy or even useful. For example, systems based on database access (or persistency in general) are known to be hard to change. Most business applications are very closely coupled to the database schema that supports them making it hard to modify either the database or the system built on it. Even if you have a sufficient layering separating the objects from the database, you are still required to migrate your data when the database schema changes. Data migration is also a very familiar nuisance for anyone who has ever tried to modify a Java application using serialization.

Changing an interface in an object-oriented system is always a major change when compared to changing implementation. Unfortunately most refactoring change interfaces. This problem is easy to deal with if you have access to the code that uses the changed interfaces you just modify those parts too.

The problem is much more severing with published interfaces (i.e. interfaces that are used by people outside your organization or by code that you cannot change). If you must refractor a published interface, you should keep both the old and the new version (at least for some time) and the old version should be tagged deprecated. These complications imply that you should not publish your interfaces prematurely.

Sometimes you should not refractor a system to begin with. The most common reason for this is that the system needs to be written again from scratch. This is usually due to the fact that the system simply does not work and is so full of bugs that it cannot easily be fixed.

12.1.8 A Critical View on Refactoring

The most obvious argument against refactoring would be that there really is not much new to it. All the techniques applied in refactoring have been around for years. On the other hand, there clearly is a demand for an easy-to-use handbook for software transformations and maintenance in general. Refactoring catalogues can well serve that purpose.

More serious disappointment is that, after all, refactoring seems to offer very little support for adaptation and maintenance of large legacy systems. Fowler emphasizes that refactoring should be an integral part of the development process, but gives almost no indication on how to work with complex systems that have not been constructed to enable modifications. Some concrete tips on where to begin refactoring and how to proceed would have been helpful.

Fowler claims that refactoring makes redesign inexpensive. This seems rather an exaggerated statement if anything else but the lowest level of design is concerned. Refactoring provide ways to safely switch between implementation mechanics with different characteristics. There is, however, much more to design than choosing a class structure or an object interaction scheme. Fowler's refactoring does not deal with higher level matters such as implementation environment selection, distribution strategies, or user interface characteristics.

Task Search more about the refactoring.

12.2 Verification

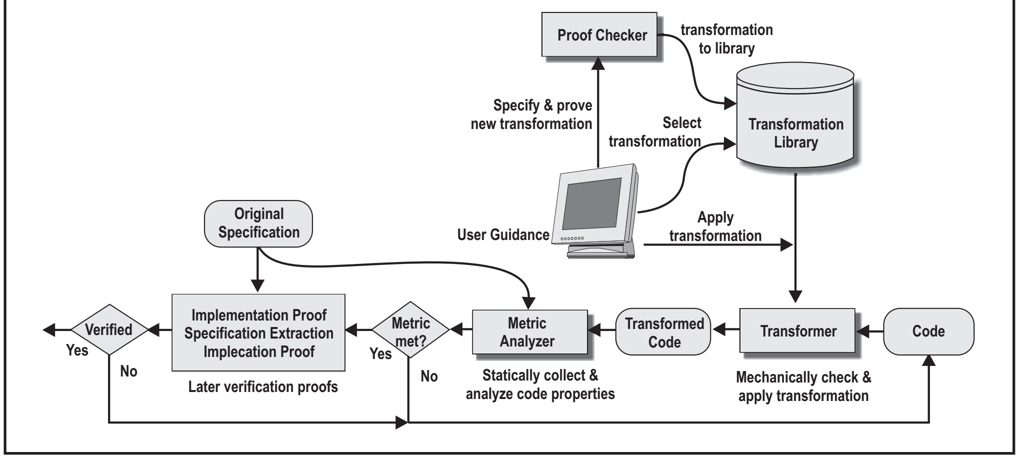
Our process for applying verification refactoring in practice is shown in Figure 12.3. A semantics-preserving transformation from the library is chosen by the user (or suggested automatically), and the transformer then checks the applicability of the selected transformation mechanically and applies it mechanically if it is applicable. When all of the selected transformations have been applied, a metrics analyzer collects and analyzes the code properties of the transformed code, and presents the complexity metrics to the user. If the metric results are not acceptable, or if they are acceptable but later verification proofs cannot be established, the process goes back to refactoring and more transformation are performed. The role of the source-code metrics is to give the user insight into the likely success of the two Echo proofs. We hypothesize that the metrics we use are an indication of relative complexity and therefore of likely verification difficulty, and we present some support for this hypothesis in the case study. Verification refactoring cannot be fully automatic in the general case, because recognizing effective transformation requires human insight except in special cases.

Furthermore, some software, especially domain specific applications might require transformations that do not exist in the library. In such circumstance, the user can specify and prove a new semantics-preserving transformation using the proof template we provide and add it to the library.

To facilitate exploration with transformations, if the user has confidence in a new transformation, the semantics-preserving proof can be postponed until the transformation has been shown to be useful or even until the remainder of the verification is complete. In most cases; the order in which transformations are applied does not matter. Clearly, however, when two transformations are interdependent, they have to be applied in order. A general heuristic is that those transformations that change the program structure and those that can vastly reduce the code size should be applied earlier.

Notes

Figure 12.3: The Verification Refactoring Process


Proof Checker
transformation to library
Specify & prove new transformation
Select transformation
Transformation Library
Original Specification
User Guidance
Apply transformation
Verified
Yes
No
Implementation Proof Specification Extraction Implecation Proof
Later verification proofs
Metric met?
Yes
No
Metric Analyzer
Statically collect & analyze code properties
Transformed Code
Transformer
Mechanically check & apply transformation
Code

We are not aware of any transformations or circumstances of their application in which a transformation would have to be removed, and we make no explicit provision for removal in the current tools and process. In the event that it becomes necessary, removing a transformation is achieved easily by recording the software's state prior to the application of each transform. All the user activities, especially the design and selection of transformations, have to be mechanically checked, and these two activities need to support by automation to the extent possible. The transformer is implemented using the Stratego/XT toolset. Stratego checks the applicability of the selected transformation, and carries it out mechanically using term rewriting. We use the PVS theorem proven as the transformation proof checker and provide a proof template. When the user specifies a new transformation, an equivalence theorem will be generated automatically, and the user can discharge it interactively in the theorem proverb.

Did u know? The term "bug" was used in an account by computer pioneer Grace Hopper (in 1946), who publicized the cause of a malfunction in an early electromechanical computer.

12.3 Software Metrics

Software cannot be seen nor touched, but it is essential to the successful use of computers. It is necessary that the reliability of software should be measured and evaluated, as it is in hardware. IEEE 610.12-1990 defines reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time." IEEE 982.1-1988defines Software Reliability Management as "The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance." Using these definitions, software reliability is comprised of three activities:

1. Error prevention
2. Fault detection and removal
3. Measurements to maximize reliability, specifically measures that support the first two Activities there have been extensive work in measuring reliability using mean time between failure and mean time to failure.

Successful modelling has been done to predict error rates and Software metrics have been proved to reflects the software quality, and thus has been widely used in software quality evaluation methods. The results of these evaluation methods can be used to indicate which parts of a software system need to be reengineered. The reengineering of these parts is usually performed using refactoring. Refactoring is defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" .The abundance of available tools for collecting software metrics and performing automated refactoring on source code simply maintenance. This means that more developers and software engineers use refactoring consciously as a means to improve the quality of their software. However, as this study suggests, the expected and actual results often differ. Although people use refactoring (or claim to do so) in order to improve the quality of their systems, the metrics indicate that this process often has the opposite results.

Self Assessment Questions

1. A development method with incremental refactoring would have developers always working with hilly.............

   (*a*) existing code (*b*) refectory code

   (*c*) duplicated code (*d*) None of these

2. Data migration is also a very familiar nuisance for anyone who has ever tried to modify a Java application using serialization.

   (*a*) True (*b*) False

3. ....refactoring cannot be fully automatic in the general case, because recognizing effective transformation requires human insight except in special cases.

   (*a*) Documenting (*b*) Relevant

   (*c*) Verification (*d*) Fowler

4. .... does not deal with higher level matters such as implementation environment selection, distribution strategies, or user interface characteristics.

   (*a*) Documenting (*b*) Relevant

   (*c*) Verification (*d*) Fowler's refactoring

5. The abundance of available tools for collecting software metrics and performing automated refactoring on source code simply maintenance.

   (*a*) True (*b*) False

12.4 Size of Metrics

Size of Matrices has verification complexity metric available that could guide the user in selection of transformations, and so we present a hybrid of metrics to the use the SPARK Examiner, and our own analyzer. The metrics include:

Element Metrics

Lines of code, number of declarations, statements, and subprograms, average size of subprograms, logical SLOC, unit nesting level, and construct nesting level.

Complexity Metrics

McCabe cyclamate complexity, Essential complexity, Statement complexity, Short-circuits complexity, and Loop nesting level.

Verification Condition Metrics

The number and size of verification condition maximum length of verification conditions and the time that the Spark tools take to analyze the verification conditions.

Specification Structure Metrics

The user is presented with a summary of the architectures of the original and the extracted specifications. This allows comparison so as to get an idea of the difficulty of implication proof.

All the metrics are subjective, and we do not have specific values that would give confidence in the ability of the PVS theorem proves to complete the implication proof.

We developed the following heuristics to both select transformations and determine the order of application:

- Dependent transformations are applied in order.
- Transformations that impact the major sources of difficulty, such as code and VC size, are applied first.
- Transformations that affect global structure are applied earlier and those that affect local structure are applied later.
- Refactoring proceeds until all proofs are possible.
- Refactoring proceeds until all subprograms can easily annotate.
- And the program structure "matches" the specifications. In practice, if specification extraction or either of the proofs fails to complete, or if either proof is unreasonably difficult, the user returns to refactoring and applies additional transformations.

12.4.1 Verification Refactoring

The Advanced Encryption Standard (AES) implementation employs various optimizations (including implementing functions using table lookups, fully or partially unrolling loops, and packing four 8-bit bytes into a 32-bit word) that improved performance but also created difficulties for verification. For instance, the SPARK tools ran out of resources on the original program because the unrolled loops created verification conditions that were too large.

We applied a total of 50 refactoring transformations in eight categories. Of those 50, the following 38 transformations from six categories were selected from the prototype Echo refactoring library (the number after the category name is the number of transformations applied in that category): rerolling loops (5); reversing inline functions or cloned code (11); splitting procedures (2); moving statements into or out of conditionals (3); adjusting loop forms (4); modifying redundant or intermediate computations (2); and modifying redundant or intermediate storage (11). The rationale and use of these transformations are discussed in the next section.

In addition to the transformations above, we also added two new categories of transformations specifically for AES:

Adjusting Data Structures

The 32-bit words were replaced by arrays of four bytes, and sets of four words were packed into blocks or states as defined by the specification. Constants and operators on those types were also redefined accordingly to reflect the transformations.

Reversing Table Lookups

The table lookups were replaced with explicit computations based on the original documentation and the precompiled tables removed. Both of these two added transformation types were driven by the goal of reversing documented optimizations and matching the extracted specification to the original specification. The final refectories AES program contained 25 functions and was 506 lines long.

12.5 Complexity Metrics Analysis

Using the heuristics mentioned earlier, we selected and ordered transformations to use with AES. Rather than examining the effects of each transformation separately, we grouped the transformations into the following 14 blocks:

1. Loop rerolling for major loops in the encryption and decryption functions
2. Reversal of word packing to use four-byte arrays
3. Reversal of table lookups
4. Packing four words into a state/block
5. Reversal of inline subroutines for major the encryption and decryption functions
6. Reversal of inline functions for key expansion subprograms
7. Moving statements into conditionals to reveal three distinct execution paths followed by procedure splitting
8. Adjustment of loop forms
9. Reversal of additional inline functions
10. Loop rerolling for sequential state updates
11. Procedure splitting
12. Adjustment of intermediate variables
13. Adjustment of loop forms; and
14. Additional procedure splitting.

Blocks 7-11 were for the subprogram that set up the key schedule for encryption, and blocks 12-14 were for the subprogram that modified the key schedule for decryption. As well as the main transformations, each block of transformations involved smaller transformations that modified redundant or intermediate computations and storage.

As part of determining whether further refactoring was required, we periodically attempted the proofs and determined the source-code metrics. Some of the results of the effect of applying the transformations on the values of the metrics are (shown in Figure 12.4). The histograms show the values of different metrics after the application of the 14 blocks of transformations where block 0 the original code is:

As the transformations were applied, the primary element metric, code size, dropped. The non-comment, non-annotation lines dropped from over 1365 to 412 mainly because loops were rerolled and precompiled tables was removed. The other transformations had little additional effect on length. We hypothesize that fewer source code lines will usually result in shorter annotations and verification conditions.

The average McCabe cyclamate complexities also declined as transformations were applied, dropping from 2.4 to 1.48. Statement complexity, essential complexity, etc. also declined. There is no evidence that these complexity metrics are related to the difficulty of verification, but their reduction suggests that the refectories program might be easier to analyses.

Since we would not undertake full annotation until refactoring was complete, we had no way to assess the feasibility of the proofs. To gain some insight, we set the post conditions for all subprograms to *true* for each version of the refectories code, generated verification conditions

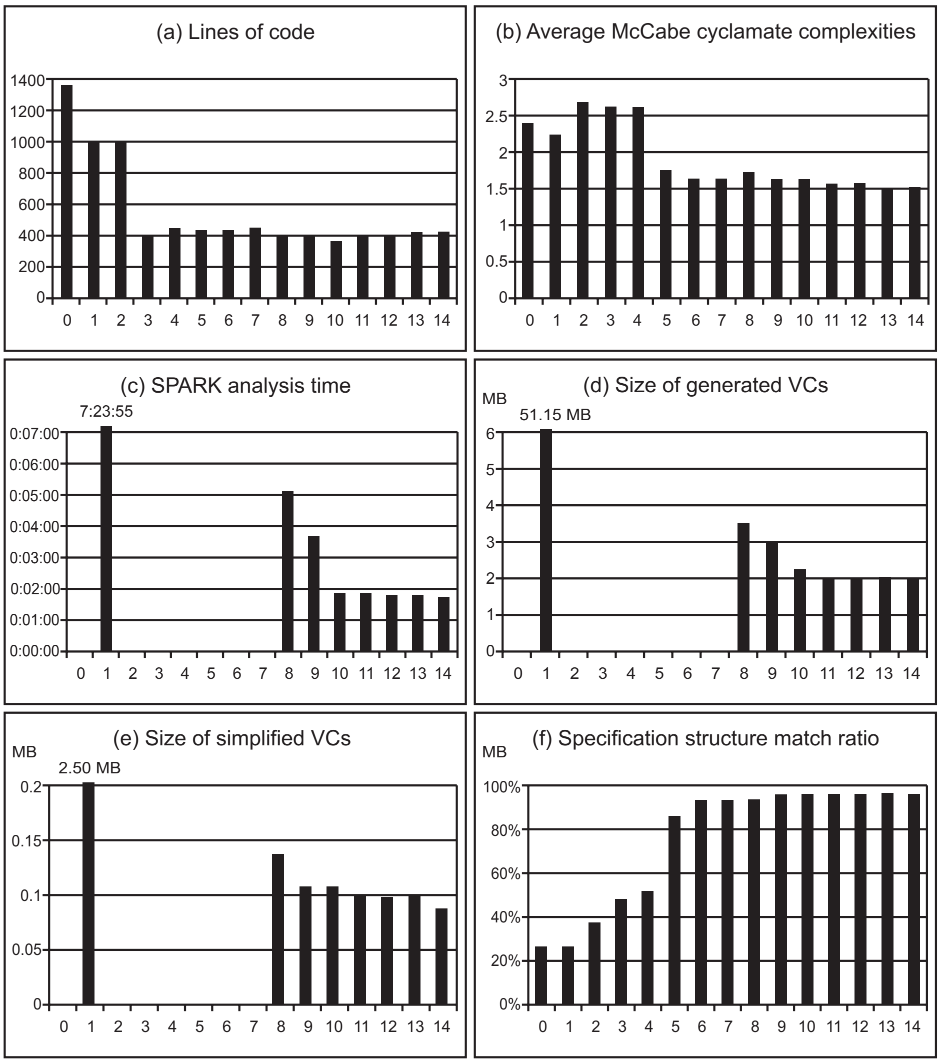
Notes

(VCs) using the SPARK examiner, and simplified the generated VCs using the SPARK simplifier. We measured the number of VCs, the size of VCs, the maximum length of VCs, and the time that the SPARK tools took to analyze the code. These data did not necessarily represent the actual proof effort needed for the implementation proof, but they were an indication.

The times required for analysis with the SPARK tools after the various refactoring are shown as Figure 12.4 (c). Some blocks are shown with no value because the VCs were too complicated to be handled by the SPARK tools. After the first loop rerolling at block 1, the tools completed the analysis but took 7 hours and 23 minutes on a 2.0 GHz machine. At block 2 with word packing reversed, the analysis again became infeasible. It was not until block 8, when we had adjusted the loop forms in the key schedule setup function that analysis by the SPARK tools became feasible again. The required analysis time gradually decreased and reached 1 minute 42 seconds for the final refectories program (a reduction of more than 99%).

The size of VCs showed the same declining trend. In block 1, 51.16 MB VCs were generated and 2.59 MB VCs were left after simplification. For the final refectories code, 1.90 MB VCs were generated and 86 KB were left after simplification (Figs 12.4(d) and 12.4(e)).

Figure 12.4: Metric Analysis with AES Verification Refactoring



The simplified VCs were those that needed human intervention to prove. After block 1, the maximum VC length was over 10,000 lines, far beyond what a human could manage. In the final refectories code, the maximum was 68 lines. When the implementation annotation was complete, the maximum length of VCs needing human intervention was only 126 lines.

We extracted a skeleton specification from the code after applying each block of transformations. These specifications were skeletons because they were obtained before the code had been annotated and so contained none of the detail from the annotations. We compared the structure of the skeleton extracted specification with that of the original specification by visually inspection and evaluated a match-ratio metric. This is defines as the percentage of key structural element data types, operators, functions and tables in the original specification that had direct counterparts in the extracted specification. We hypothesize that this measure is an indication of the likelihood of success fully establishing the implication proof.

The values of the match ratio are shown in Figure 12.4(f). The ratio increased gradually from 25.9% to 96.3% as the transformation blocks were applied. There is only a small increase in its value after the block 8 transformations were applied, and the implication proof could have been attempted at that point. However, although some metrics had stabilized after the block 8 transformations, the time required for the SPARK analysis was still declining.

Case Study

Impact of Refactoring On Quality and Productivity in an Agile Team

The object under study is a software project in an agile, close-to-industrial development environment ("close-to industrial" refers to an environment where the development team is composed of both professional software engineers and students). The result is a commercial software product developed at VTT in Oulu, Finland, to monitor applications for mobile, Java enabled devices. The programming language was Java (version 1.4) and the IDE was Eclipse 3.0. The project was a full business success in the sense that it delivered on time and on budget the required product. Four developers formed the development team. Three developers had an education equivalent to a BSc and limited industrial experience. The fourth developer was an experienced industrial software engineer. The development process followed a tailored version of the Extreme Programming practices, which included all the practices of XP but the "System Metaphor" and the "On-site Customer"; there was instead a local, on-site manager that met daily with the group and had daily conversations with the off-site customer. In particular, the team worked in a collocated environment and used the practice of pair programming. The project lasted eight weeks and was divided into five iterations, starting with a 1-week iteration, which was followed by three 2-weeks iterations, with the project concluding in final 1-week iteration. Throughout the project, mentoring was provided on XP and other programming issues according to the XP approach. Since the team was exposed for the first time to an XP-like process, a brief training of target XP practices was given before the start of the project. The total development effort per developer was about 192 hours (6 hours per day for 32 days). Since with PROM we monitored all the interactions of the developer with different applications, we are able to differentiate between coding and other activities: About 75% of the total development effort was spent for pure coding activities inside the IDE while the remaining 25% was spent for other assignments like working on text documents, reading and writing emails, browsing the web and similar tasks. The developed software consists of 30 Java classes and a total of about 1770 Java source code statements (LOC counted as number of semicolons in a Java program). During development two user stories have been explicitly written for refactoring activities: One at the end of iteration two

Contd...

with the title "Refractor Static Classes to Object Classes" and one at the end of iteration four with the title "Refractor Architecture". We refer to the implementation of these two user stories as explicit refactoring; we analyze changes of productivity and quality measures before and after their completion. Then we concern two methods are:

- H0A –Does productivity increase after "explicate factorings"?
- H0B –Cohesion, coupling and complexity: does refactoring improve code quality?

Although agile processes and practices are gaining more importance in the software industry there is limited solid empirical evidence of their effectiveness. This research focuses in particular on the practice of refactoring, which is one of the key practices of Extreme Programming and other Agile Methods. While the majority of software developers and researchers agree that refactoring has long-term benefits on the quality of a software product (in particular on program understanding) there is no such consensus regarding the development productivity. Available empirical results regarding this issue are very limited and not clear. This might refrain managers from adopting refactoring, as they might be scared of loosing resources. This work contributes to a better understanding of the effects of refactoring both on code qualities in particular on software maintainability - and development productivity in a close-to industrial, agile development environment. It provides new empirical, industrially based evidence that refactoring rather increases than decreases development productivity and improves quality factors, as measured using common internal quality attributes reduces code complexity and coupling; increases cohesion that was too limited to be taken as a reference. For internal quality metrics, our results are in accordance with the existing literature. Altogether, we believe that our findings are particularly relevant, as this work is a case study in a close-to-industry environment, a kind of empirical investigation that is rare for the research problem we discuss here. Clearly, this is a first work in the area. A real generalizable assessment of the implications of refactoring requires several repetitions of studies like this, possibly also including data on defects. The findings of this research have major implications for a widespread use of refactoring, as already mentioned by Beck in his first work on XP. Of course refactoring as any other technique is something a developer has to learn. First, managers have to be convinced that refactoring is very valuable for their business; this research should help them in doing so as it sustains that refactoring if applied properly intrinsically improves code maintainability and increases development productivity. Afterwards, they have to provide training and support to change their development process into a new one that includes continuous refactoring.

Questions

1. Define Agile refactoring process is maintain? Explain it.

2. Describe explicit refactoring.

Self Assessment Questions

6. A series of .... without changing its observable behaviour.

   (*a*) refactoring (*b*) factoring

   (*c*) configuring (*d*) All of these.

7. The power of refactoring as a method lies in ……………

   (*a*) systematically organized documents

   (*b*) rapped documents

   (*c*) simple

   (*d*) All of these.

8. The process of optimizing the reliability of .... through a program.
   (*a*) matrices (*b*) software
   (*c*) refactoring (*d*) All of these.
9. Refactoring is an activity crucial for evolutionary ....
   (*a*) Document process (*b*) right key process
   (*c*) Verification process (*d*) All of these.
10. Refactoring can be thought of as an alternative to:
    (*a*) Document upfront design (*b*) simple design
    (*c*) Careful upfront design (*d*) All of these.
11. Software Matrices Activates is:
    (*a*) Error prevention (*b*) complex prevention
    (*c*) Simple prevention (*d*) All of these.
12. Software metrics have been proved:
    (*a*) Bad quality (*b*) Right quality
    (*c*) To reflect software quality (*d*) All of these.
13. Refactoring is the process of converting existing
    (*a*) "Bad" code into "good" code (*b*) bugs code
    (*c*) Simple Code (*d*) All of these.
14. Matrices Complex city is the right process of ....
    (*a*) verification refactoring (*b*) factoring
    (*c*) bugs factoring (*d*) All of these.
15. Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.
    (*a*) True (*b*) False

12.6 Summary

- Refactoring is the process of converting existing "bad" code into "good" code. It is done via the parser, the relational model, and the language module.
- Refactoring is an activity crucial for evolutionary development processes.
- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour is called Refactoring.
- Metrics to measure software reliability do exist and can be used starting in the requirements phase.
- Refactoring helps the code to retain its shape. Without refactoring the design of the program will decay. As people change code.
- The power of refactoring as a method lies in systematically organized documents, which describe proven techniques for enhancing software.

12.7 Keywords

Documenting Refactoring: The power of refactoring as a method lies in systematically organized documents, which describe proven techniques for enhancing software safely.

Matrices: The process of optimizing the reliability of software through a program that emphasizes software error prevention, fault detection and removal, and the use of measurements to maximize reliability in light of project constraints such as resources, schedule and performance.

Matrices Size: Size of Matrices has verification complexity metric available that could guide the user in selection of transformations, and so we present a hybrid of metrics to the use the SPARK Examiner, and our own analyzer.

Refactoring: A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

Verification: It is the act of reviewing, inspecting or testing, in order to establish and document that a product, service or system meets regulatory or technical standards.

Lab Exercise

1. Prepare a structure to refactoring a program.
2. Create a flow chart for the verification process in refactoring.

12.8 Review Questions

1. What is the refactoring? Explain its documentation.
2. Is incremental refactoring better than wholesale refactoring?
3. What are the problems in refactoring?
4. Define verification? How it concern with refactoring?
5. What are the critical views on refactoring?
6. What are the matrices? Describe its size.
7. What are the benefits of refactoring in software engineering?
8. Explain the complexity metrics analysis.
9. Why refactoring is needed in software engineering?
10. Describe the relation between refactoring and matrices.

Answers for Self Assessment Questions

1.	(*b*)	2.	(*a*)	3.	(*c*)	4.	(*d*)	5.	(*a*)
6.	(*a*)	7.	(*a*)	8.	(*a*)	9.	(*a*)	10.	(*c*)
11.	(*a*)	12.	(*c*)	13.	(*a*)	14.	(*a*)	15.	(*a*)

12.9 Further Readings

Books

Knowledge-Based Software Engineering: Proceedings of the Sixth Joint, by Vadim Stefanuk, Kenji Kaijiri

Generative and Transformational Techniques in Software Engineering, by Ralf Lammel, Joao Saraiva, Joost

Online link

http://books.google.co.in/books?id=eWVWV8BjHeoC&pg=PA219&dq=software+engineering+in+Refactoring&hl=en&sa=X&ei=qVIKUKuCFo7prQfslbXJCA&ved=0CGUQ6AEwCA#v=onepage

Unit 13: Testing

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Explain the fundamentals of testing
- Explain the test oracles

Notes

- Understand the black box
- Explain equivalence class partitioning
- Explain white box
- Define control flow based

Introduction

The aspire of the testing process is to identify all defects obtainable in a software product. However, for most practical systems, even after satisfactorily carrying out the testing phase, it is not probable to guarantee that the software is fault free. This is because of the fact that the input data domain of most software products is very large. It is not practical to test the software exhaustively with respect to each value that the input data may assume. Even with this sensible limitation of the testing process, we should not underestimate the significance of testing. We must remember that testing does expose many defects existing in a software product. Therefore, we can safely conclude that testing provides a practical way of reducing defects in a system and increasing the users' confidence in a developed system. Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Although crucial to software quality and widely deployed by programmers and testers, software testing still remains an art, due to limited understanding of the principles of software. The difficulty in software testing stems from the complexity of software. We cannot completely test a program with moderate complexity. Testing is more than just debugging. The purpose of testing can be quality assurance, verification and validation, or reliability estimation. Testing can be used as a generic metric as well. Correctness testing and reliability testing are two major areas of testing. Software testing is a tradeoff between budget, time and quality.

Testing a program consists of subjecting the program to a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which failure occurs are noted for later debugging and correction.

The following are some commonly used terms associated with testing.

- A failure is a manifestation error (or defect or bug) but the mere presence of an error may not necessarily lead to a failure.
- A test case is the triplet [I, S, O], where we are is the data input to the system, S is the state of the system at which the data is input, and 0 is the expected output of the system.
- A test suite is the set of all test cases with which a given software product is to be tested.

13.1 Concept of Testing

Software testing is a field with no set "best practices" because so much of testing is based on the particular background of the test, it is often difficult to explain, categorize and dispense advice on aspects of software testing. However, there are fundamental rules around. We entire site is dedicated to the basics of software testing. However, we need to first master the basics of the basics before we begin. We are just preliminary the journey into the world of software testing. Regardless of the cause, once a software maker has decided to use official methods, it must address the question of which formal methods and metrics to adopt. Once methods or a course toward methods has been determined, everyone must be educated in the new methods. Moving an established culture from an informal method of doing something to a formal method of doing the same thing takes time, determination, and a good cost benefit ratio. It amounts to a cultural change, and introducing culture changes is risky business. Once the new methods are established, it still takes a continuing commitment from management to keep them alive and in use.

13.1.1 Verification and Validation

Verification

Verification Are we doing the job correct, the set of activities that ensure that software properly implements a specific function. (The process of determining whether or not foodstuffs of a given phase of the software development cycle fulfill the requirements established during phase)

Example: Technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis etc.

Validation

Validation Are we doing the right job, the set of activities that ensure that the software that has been built is traceable to customer requirements. (An attempt to find errors by executing the program in a real environment)

Example: Unit testing, system testing and installation testing etc.

Did u know? The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979.

13.1.2 Testing Quality

Software Quality

Learn how software quality is defined and what it means. Software quality is the degree of conformance to explicit or implicit requirements and expectations.

Dimensions of Quality

Learn the dimensions of quality. Software quality has dimensions such as Accessibility, Compatibility, Concurrency and Efficiency.

Software Quality Assurance

Learn what it means and what its relationship is with Software Quality Control. Software Quality Assurance is a set of activities for ensuring quality in software engineering processes.

Software Quality Control

Learn what it means and what its relationship is with Software Quality Assurance. Software Quality Control is a set of activities for ensuring quality in software products.

SQA and SQC Differences

Learn the differences stuck between Software Quality Assurance and Software Quality Control. SQA is process focused and prevention oriented but SQC is product focused and detection oriented.

Software Development Life Cycle

Learn what SDLC means and what activities a typical SDLC model comprises of Software Development Life Cycle defines the steps/stages/phases in the building of software.

Definition of Test

Learn the various definitions of the term 'test'. Merriam Webster defines Test as a critical examination, observation, or evaluation.

Notes

13.1.3 Errors

One common definition of a software error is a mismatch between the program and its specification. A mismatch between the program and its specification is an error in the program if and only if the specification exists and is correct. A software error is present for when the program does not do what its end user reasonability expects to do.

Categories of Software Errors

- User interface errors, such as output errors, incorrect user messages
- Function errors
- Defect hardware
- Incorrect program version
- Testing errors
- Requirements errors
- Design errors
- Documentation errors
- Architecture errors
- Module interface errors
- Performance errors
- Error handling
- Boundary related errors
- Logic errors
- calculation errors
- State based behaviour errors
- Communication errors
- Program structure errors, such as control flow errors

Most programmers are rather cavalier about scheming the quality of the software they write. They bang out some code, run it through some fairly obvious ad hoc tests, and if it seems okay, they are done. While this approach may work all right for small, personal programs, it does not cut the mustard for professional software development. Modern software engineering practices include considerable effort directed toward software quality assurance and testing. The idea, of course, is to produce completed software systems that have a high probability of satisfying the customer's needs. There are two ways to deliver software free of errors. The first is to prevent the introduction of errors in the first place. And the second is to identify the bugs lurking in our code, seek them out, and destroy them. Obviously, the first method is superior. A big part of software quality comes from doing a good job of defining the requirements for the system we are building and designing a software solution that will satisfy those requirements. Testing concentrates on detecting those errors that creep in despite our best efforts to keep them out.

13.1.4 Fault

A software fault is an erroneous portion of a software system which may cause failures to occur if it is run in a particular state, or with particular inputs. There are many universal statements that a tester will come across in the course of their career, some true Testing can only prove the presence of faults and not their absence, and some false "Developers must not test their own code", Such statements may well be paraphrased quotes, the original source may not be known or presented, and the quote may be taken out of its original context. But they will be one common form of the phrase that exists in the industry's shared consciousness. Each of these statements has an effect on the person hearing them or thinking about them. And each of the Development Lifecycle personnel, no matter what their role is, will re interpret the phrases based on their own experience. The benefit of such statements is that they provide something to think around. The danger is that the brain loves to simplify things, identify patterns, reduce and compartmentalise information and can therefore build a less than optimal reasoning chain.

- Testing can only prove the presence of faults and not their absence
- The objective of testing is to find faults
- A good test is one that has a high chance of finding a fault
- A successful test is one that finds a fault
- An unsuccessful test is one that does not find a fault
- Testing can only prove the presence of faults
- Testing cannot prove the absence of faults.

Practicing tester might well argue about statement as it may run contrary to their experience and the aims of testing as defined by their organization. One of the aims of their testing process is to validate the requirements, in effect, to prove the absence of faults. If a test runs to completion without a fault being identified, and on repeated runs (when it is run in exactly the same way) does not identify a fault. Possibly, but we cannot prove it. The test did not identify a fault but that does not mean that at a lower level of abstraction than the test was described in terms of, a fault did not occur. Perhaps the results were merely coincidentally correct.

13.1.5 Failure

If under certain environment and situation defects in the application or product get executed then the system will produce the wrong results causing a failure. Not all defects result in failures, some may stay inactive in the code and we may never notice them. Example Defects in dead code will never result in failures. It is not just defects that give rise to failure. Failures can also be caused because of the other reasons also like.

- Because of the environmental conditions as well like a radiation burst, a strong magnetic field, electronic field or pollution could cause faults in hardware or firmware. Those faults might prevent or change the execution of software.
- Failures may also arise because of human error in interacting with the software, perhaps a wrong input value being entered or an output being misinterpreted.
- Finally failures may also be caused by someone deliberately trying to cause a failure in the system.

The main premise is that scientists get so comfortable with accepted theory and the status quo that they do not recognize that failures might be breakthroughs instead of just mistakes in their own equipment or experimental method. There is certainly some value there gives the example of sensitive radio telescope static finally being accepted as cosmic background radiation and not a problem with the dish, and cites some serious ethnographic research of scientists and how they make discoveries.

Fault: It is a condition that causes the software to fail to perform its required function.

Error: Refers to difference between Actual Output and Expected Output.

Failure: It is the inability of a system or component to perform required function according to its specification.

IEEE Definitions

Failure: External behaviour is incorrect

Fault: Discrepancy in code that causes a failure

Error: Human mistake that caused fault

Note

Error: is terminology of Developer

Bug: is terminology of Tester.

13.2 Test Oracles

The oracle problem is addressed for chance testing and difficult of randomized software. The presented Statistical Oracle is a Heuristic Oracle using statistical methods, especially statistical tests. The Statistical Oracle is applicable in case there are explicit formulae for the mean, the distribution, and so on, of characteristics computable from the test result. As with the Heuristic Oracle, the decision of the Statistical Oracle is not always right. The Statistical Oracle has successfully been functional. To test any program, we need to have a description of its expected behaviour and a method of determining whether the observed behaviour conforms to the expected behaviour. For this we need a test oracle. A test oracle is a mechanism, different from the program itself, which can be used to check the correctness of the output of the program for the test cases. Conceptually, we can consider testing a process in which the test cases are given to the test oracle and the program under testing. The output of the two is then compared to determine if the program behaved correctly for the test cases. Test designers widely believe that the overall effectiveness and cost of software testing depends largely on the type and number of test cases executed on the software. Test oracle used during testing also contributes significantly to test effectiveness and cost. A test oracle is a mechanism that determines whether software executed correctly for a test case. We define a test oracle to contain two essential parts, oracle information that represents expected output, and an oracle procedure that compares the oracle information with the actual output. By varying the level of detail of oracle information and changing the oracle procedure, a test designer can create different types of test oracles.

13.3 Test Cases and Criteria

Test case and criteria for testing is given blow:

13.3.1 Test Cases

A test case is a documentation which specifies input principles, expected output and the preconditions for executing the test or a test case is a scenario made up of a sequence of steps and conditions or variables, where test inputs are provided and the program is run using those inputs, to see how it performs. An expected result is outlined and the actual result is compared to it. Certain working conditions are also present in the test case, to see how the program handles the conditions. Every requirement or objective that the program is expected to achieve, needs at least one test case. Realistically, it definitely takes more than one test case to determine the true functionality of the application being tested. The mechanism used to judge the result of the test case, i.e. whether the program has failed or passed the test, is called a test oracle. Test cases, at root level, are used to measure how a program handles errors or tricky situations such as if one input is incorrect or if both inputs are incorrect. They are also expected to expose hidden logical errors in the program's code that have gone undetected.

- Should be accurate and tests what it is intended to test.
- No unnecessary steps should be included in it.
- It should be reusable.
- It should be traceable to requirements.
- It should be compliant to regulations.
- It should be independent i.e. we should be able to execute it in any order without any dependency on other test cases.
- It should be simple and clear, any tester should be able to understand it by reading once.
- Now keeping in mind these characteristics we can write good and effective test cases.

13.3.2 Type of Test Case

A formal written test case can be divided into three main parts:

Information

Information consists of general information about the test case such as a case identifier, case creator info, test case version, formal name of the test case, purpose or brief description of the test case and test case dependencies. It should also include specific hardware and software requirements (if any) and setup or configuration requirements.

Activities

This part consists of the actual test case activities such as the environment that should exist during testing, activities to be done at the initialization of the test, activities to be done after test case is performed and step by step actions to be done while testing and the input data that is to be supplied for testing.

Results

Results are the outcomes of a performed test case. Result data consists of information about expected results, which is the criteria necessary for the program to pass the test and the actual recorded results.

13.3.3 Criteria

As a software organization grows and becomes more sophisticated by technology upgrades, it recognizes the need for Business Analysts and Quality Assurance to support their Information System functions. Often the practice of software changes released to the QA team is a matter of informal presentment. This presentment at best includes lists of modules, functional or design changes and timelines for moving the changes into production. We witness the roots of most test organizations planted firmly and integrated in the development department. This commonalty or kinship with the development organization often leads to biasness as the organization grows. The development of formal test organizations requires a formal interface with development teams. This interfacing serves as the Entry Criteria for the test organization. In addition, the output from the test organization can be labelled as Exit Criteria. The quality and effectiveness of software testing is primarily determined by the quality of the test processes used. The identification of Entry and Exit criteria in software testing are a critical step in further improving the test process.

Entry Criteria

Entry Criteria for testing is defined as "Specific conditions or ongoing activities that must be present before a process can begin". In the Systems Development Life Cycle it also specifies which entry criteria are required at each phase. Additionally, it is also important to define the time interval or required amount of lead time that an entry criteria item is available to the process.

Input can be divided into two categories. The first is what we receive from development. The second is what we produce that acts as input to later test process steps. The type of required input from development includes.

- Technical Requirements/Statement of Need
- Design Document
- Change Control
- Turnover Document

The type of required input from test includes

- Evaluation of available software test tools
- Test Strategy
- Test Plan
- Test Incident Reports

The Entry Exit Criteria matrix, we get the clarity of the deliverables expected from each phase. The matrix should contain "date required" and should be modified to meet the specific goals and requirements of each test effort based on size and complexity.

Exit Criteria

Exit Criteria is often viewed as a single document commemorating the end of a life cycle phase. Exit Criteria is defined as "The specific conditions or ongoing activities that must be present before a life cycle phase can be considered complete. The life cycle specifies which exit criteria are required at each phase". This definition identifies the intermediate deliverables, and allows us to track them as independent events. The type of output from test includes.

- Test Strategy
- Test Plan
- Test Scripts/Test Case Specifications
- Test Logs
- Test Incident Report Log
- Test Summary Report/Findings Report

By identifying the specific Exit criteria, we are able to identify and plan how these steps and processes fit into the life cycle. All of the Exit Criteria listed, less the Test Summary/Findings Report; act as Entry Criteria to alter process.

It is this level of process understanding that provides us with the tools we need to improve the overall test process.

Self Assessment Questions

1. .... is the degree of conformance to explicit or implicit requirements and expectations.

   (*a*) Software quality (*b*) Dimensions of quality

   (*c*) Software quality assurance (*d*) Software quality control

2. Categories of software error:

   (*a*) Fault (*b*) Failure

   (*c*) Error handling (*d*) None of these

3. …………………are often viewed as a single document commemorating the end of a life cycle phase.

   (*a*) Criteria (*b*) Exit criteria

   (*c*) Entry criteria (*d*) Entry and exit criteria

4. Which type of output from test includes:

   (*a*) Test plan (*b*) Design document

   (*c*) Turnover document (*d*) Change control

5. Exit criteria are not defined as the specific conditions or ongoing activities that must be present before a life cycle phase can be considered complete.

   (*a*) True (*b*) False

13.4 Black Box Testing

The black box approach is a testing method in which test data are resulting from the specified useful requirements without regard to the final program structure. It is also termed data driven; Input/Output driven or requirements based testing. Because only the functionality of the software module is of concern, black box testing also mainly refers to functional testing method emphasized on executing the functions and examination of their Input and Output data. The tester treats the software under test as a black box only the inputs, outputs and specification are visible, and the functionality is determined by observing the outputs to corresponding inputs. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation details of the code are considered.

It is understandable that the more we have covered in the input space, the more problems we will find and therefore we will be more confident about the quality of the software. Ideally we would be tempted to exhaustively test the input space. But as stated above, exhaustively testing the combinations of valid inputs will be impossible for most of the programs, let alone considering invalid inputs, timing, sequence, and resource variables. Combinatorial explosion is the major roadblock in functional testing. To make things worse, we can never be sure whether the specification is either correct or complete. Due to limitations of the language used in the specifications (usually natural language), ambiguity is often inevitable. Even if we use some type of formal or restricted language, we may still fail to write down all the possible cases in the specification. Sometimes, the specification itself becomes an intractable problem, it is not possible to specify precisely every situation that can be encountered using limited words. And people can seldom specify clearly, they usually can tell whether a prototype is, or is not, what they want after they have been finished operating system like Windows, a website like Google, a database like Oracle or even our custom application. Under Black Box Testing, we can test these applications by just focusing on the inputs and outputs without knowing their internal code implementation.

13.4.1 Black Box Testing Steps

Initially requirements and specifications of the system are examined. Tester chooses valid inputs to check whether SUT processes them properly. Also some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them. Tester determines expected outputs for all those inputs. Software tester constructs test cases with the selected inputs. The test cases are executed. Software tester compares the actual outputs with the expected outputs. Defects if any are fixed and re tested.

13.4.2 Types of Black Box Testing

Functional Testing

This black box testing type is related to functional requirements of a system; it is done by software testers.

Non Functional Testing

This type of black box testing is not related to testing of a specific functionality, but non functional requirements such as performance, scalability, usability.

Black Box Testing Strategy

Following are the prominent test strategy amongst the many used in Black box Testing.

Equivalence Class Testing

It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.

Boundary Value Testing

Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is mostly suitable for the systems where input is within certain ranges.

Decision Table Testing

A decision table puts causes and their effects in a matrix. There is unique combination in each column.

13.4.3 Black Box Testing and Software Development Life Cycle (SDLC)

Black box testing has its own life cycle called Software Test Life Cycle. it is relative to every stage of Software Development Life Cycle.

Requirement

This is the initial stage of and in this stage requirement is gathered. Software testers also take part in this stage.

Test Planning and Analysis

Test plan is created which determines possible project risks and their mitigation.

Design

In this stage Test cases/scripts are created on the basis of software requirement documents.

Test Execution

In this stage Test Cases prepared are executed. Bugs if any are fixed and re tested.

Task Draw the structure of white box testing.

13.4.4 Advantages and Disadvantages of Black Box Testing

In black box testing, the internals of the system are not taken into consideration. The testers do not have access to the source code. A tester who is doing black box testing generally interacts through a user interface with the system by giving the inputs and examining the outputs. The advantages of black box testing includes that it is very efficient for large segments of code. It clearly separates user's perspective from developer's perspective. The code access not required. It is very easy to execute. The disadvantage of black box testing includes limited code path coverage as limited number of inputs can be checked. It cannot control targeting code segments or paths which may be more error prone than others.

Advantages of Black Box Testing

- Efficient when used on Larger systems
- As the tester and developer are independent of each other, test is balanced and unprejudiced.
- Tester can be non technical.
- There is no need of having detailed functional knowledge of system to the tester.
- Tests will be done from a end user's point of view. Because end user should accept the system. (This is reason; sometimes this testing technique is also called as Acceptance testing).
- Testing helps to identify the vagueness and contradiction in functional specifications.
- Test cases can be designed as soon as the functional specifications are complete.

Disadvantages of Black Box Testing

- Test cases are tough and challenging to design, without having clear functional specifications.
- It is difficult to identify tricky inputs, if the test cases are not developed based on specifications.
- It is difficult to identify all possible inputs in limited testing time. So writing test cases is slow and difficult.
- Chances of having unidentified paths during this testing.
- Chances of having repetition of tests that are already done by programmer.

13.4.5 Equivalence Class Partitioning

Testing based on equivalence class analysis (synonyms equivalence partitioning, domain analysis) is a form of black box test analysis that attempts to reduce the total number of potential tests to a minimal set of tests that will uncover as many errors as possible It is a method that partitions the set of inputs and Outputs into a finite number of equivalence classes that enable the selection of a representative test value for each class. The test that results from the representative value for a class is said to be "equivalent" to the other values in the same class. If no errors were found in the test of the representative value, it is reasoned that all the other "equivalent" values would not identify any errors either. The power of Equivalence Classes lies in their ability to guide the tester using a sampling strategy to reduce the combinatorial explosion of potentially necessary tests. The technique provides logical bases by which a subset of the total conceivable number of tests can be selected. Here are some categories of problem areas for large numbers of tests that can be benefit from the consideration of equivalence classes:

- Combinations of independent variables
- Dependent variables based on hierarchical relationship
- Dependent variables based on temporal relationship
- Clustered relationships based on market exemplars
- Complex relationships that can be modeled.

Key Points

- Although in rare cases equivalence partitioning is also applied to outputs, typically it is applied to the inputs of a component being tested.
- This method divides the input domain of a program into classes of data from which test cases can be derived.
- It is based on an evaluation of equivalence classes for an input condition. An equivalence class represents a set of valid or invalid states for input conditions. An input has certain ranges which are valid and other ranges which are invalid.
- Supplemented by BVA(Boundary Value Analysis). Having determined the partitions of possible inputs the method of boundary value analysis has to be applied to select the most effective test cases out of these partitions.

13.4.6 Boundary Value Analysis

Boundary value analysis is a technique for test data assortment. A test engineer chooses values that lie along data extremes. Boundary values include maximum, minimum, just inside boundaries, just outside boundaries, typical values, and error values. The expectation is that, if a systems works correctly for these extreme or special values, then it will work correctly for all values in between. An effective way to test code is to exercise it at its natural boundaries. Boundary Value Analysis is a method of testing that complements equivalence partitioning. In this case, data input as well as data output are tested. The rationale behind BVA is that the errors typically occur at the boundaries of the data. The boundaries refer to the upper limit and the lower limit of a range of values or more commonly known as the "edges" of the boundary. BVA is a black box approach to identifying test cases. In black box testing, test cases are selected based upon the desired product functionality as documented in the specifications without consideration of the actual internal structure of the program logic. A fundamental assumption in BVA is that the majority of program errors will occur at critical Input (or Output) boundaries, places where the mechanics of a calculation or data manipulation must change in order for the program to produce a correct result. An example that illustrates the general concept of boundary values would be a program that calculates income tax for a given income. In a progressive income text scheme, the tax rate applied increases from low income brackets to high income brackets. In this case, the critical input boundaries would be the set of incomes at which the applied tax rate should change along with any minimum or maximum extremes of the income value. Thus, the set of boundary incomes defines the limits of each tax bracket.

The BVA also serves as an introduction to other test techniques. Discussions of BVA in the literature are often intermingled with a related black box technique known as Equivalence Partitioning (EP), which utilizes the boundary values in an attempt to define partitions or sets of test case that are equivalent in the sense that all test cases grouped within a particular partition would reveal the presence of the same set of defects and likewise fail to detect other defects. In its simplest form, once the partitions are identified, the set of test cases selected is one representative test case from each partition. A distinct advantage of the EP technique is that the total number of test cases is significantly smaller than the set of test cases identified through BVA. In fact, the set of test cases identified by EP can be a subset of those identified by BVA, and researchers have exploited this fact to reduce the total number of test cases identified in merged BVA EP schemes.

Boundary value analysis

It is widely recognized that input values at the extreme ends of input domain cause more errors in system. More application errors occur at the boundaries of input domain. 'Boundary value analysis' testing technique is used to identify errors at boundaries rather than finding those exist in centre of input domain. Boundary value analysis is a next part of Equivalence partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

13.5 White Box Testing

Contrary to black box testing, software is viewed as a white box, or glass box in white box testing, as the structure and flow of the software under test are noticeable to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. White box testing is also called glass box testing, logic driven testing or design based testing. There are many techniques available in white box testing, because the problem of intractability is eased by specific knowledge and attention on the structure of the software under test. The intention of exhausting some aspect of the software is still strong in white box testing, and some degree of exhaustion can be achieved, such as executing each line of code at least once (statement coverage), traverse every branch statements (branch coverage), or cover all the possible combinations of true and false condition predicates (Multiple condition coverage).

Control flow testing, loop testing, and data flow testing; all maps the corresponding flow structure of the software into a directed graph. Test cases are carefully selected based on the criterion that all the nodes or paths are covered or traversed at least once. By doing so we may discover unnecessary "dead" code that is of no use, or never get executed at all, which cannot be discovered by functional testing. In mutation testing, the original program code is perturbed and many mutated programs are created, each contains one fault. Each faulty version of the program is called a mutant. Test data are selected based on the effectiveness of failing the mutants. The more mutants a test case can kill, the better the test case is considered. The problem with mutation testing is that it is too computationally expensive to use. The boundary between black box approach and white box approach is not clear cut. Many testing strategies mentioned above, may not be safely classified into black box testing or white box testing. It is also true for transaction flow testing, syntax testing, finite state testing, and many other testing strategies not discussed in this text. One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad it may contain any requirement including the structure, programming language, and programming style as part of the specification content. We may be reluctant to consider random testing as a testing technique. The test case selection is simple and straightforward, they are randomly chosen. Study in indicates that random testing is more cost effective for many programs. Some very subtle errors can be discovered with low cost. And it is also not inferior in coverage than other carefully designed testing techniques. One can also obtain reliability estimate using random testing results based on operational profiles. Effectively combining random testing with other testing techniques may yield more powerful and cost effective testing strategies.

13.5.1 Basis Path Testing

Each independent path in the code is taken for testing:

Data Flow Testing (DFT)

In this approach we track the specific variables through each possible calculation, thus defining the set of intermediate paths through the code. DFT tends to reflect dependencies but it is mainly through sequences of data manipulation. In short each data variable is tracked and its use is verified. This approach tends to uncover bugs like variables used but not initialize, or declared but not used, and so on.

Path Testing

Path testing is where all possible paths through the code are defined and covered. Its a time consuming task.

Loop Testing

These strategies relate to testing single loops, concatenated loops, and nested loops. Independent and dependent code loops and values are tested by this approach.

13.5.2 Type of the white box testing

Unit Testing

Unit testing, which is testing of individual hardware or software units, groups of related units. A unit is a software component that cannot be subdivided into other components. Software engineers write white box test cases to examine whether the unit is coded correctly. Unit testing is important for ensuring the code is solid before it is integrated with other code. Once the code is integrated into the code base, the cause of an observed failure is more difficult to find. Also, since the software engineer writes and runs unit tests him or herself, companies often do not track the unit test failures that are observed making these types of defects the most "private" to the software engineer.

Integration Testing

Integration testing, which is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them Test cases are written which explicitly examine the interfaces between the various units. These test cases can be black box test cases, whereby the tester understands that a test case requires multiple program units to interact. Alternatively, white box test cases are written which explicitly exercise the interfaces that are known to the tester.

Regression Testing

Regression testing, which is selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements. As with integration testing, regression testing can be done via black box test cases, white box test cases, or a combination of the two White box unit and integration test cases can be saved and rerun as part of regression testing.

13.5.3 Advantages and Disadvantages White Box Testing

Advantages

Advantages of White Box Testing are mentioned below:

- All the features and functionality within the application can be tested
- Testing can be started at the very initial stage. Tester does not need to wait for interface or GUI to be ready for testing
- Can reduce to number of test cases to be executed during black box testing
- Helps in checking coding standards and optimizing code
- Extra code resulting in hidden defects can be removed
- Reason of failure can be known
- Identifying test data is easy because coding knowledge will be a pre requisite.

Disadvantages

Disadvantages of white box testing are mentioned below:

- Tester should be highly skilled because should have the knowledge of coding, implementation

- Cost of tester is very high
- White Box testing is very complex
- It is not possible to look into each piece of code to find out hidden errors
- Test cases maintenance can be tough if the implementation changes very frequently
- Since White Box Testing it closely tied with the application being testing, tools to cater to every kind of implementation/platform may not be readily available
- Exhaustive testing of larger system is not possible.

13.6 Data Flow Based Testing Techniques

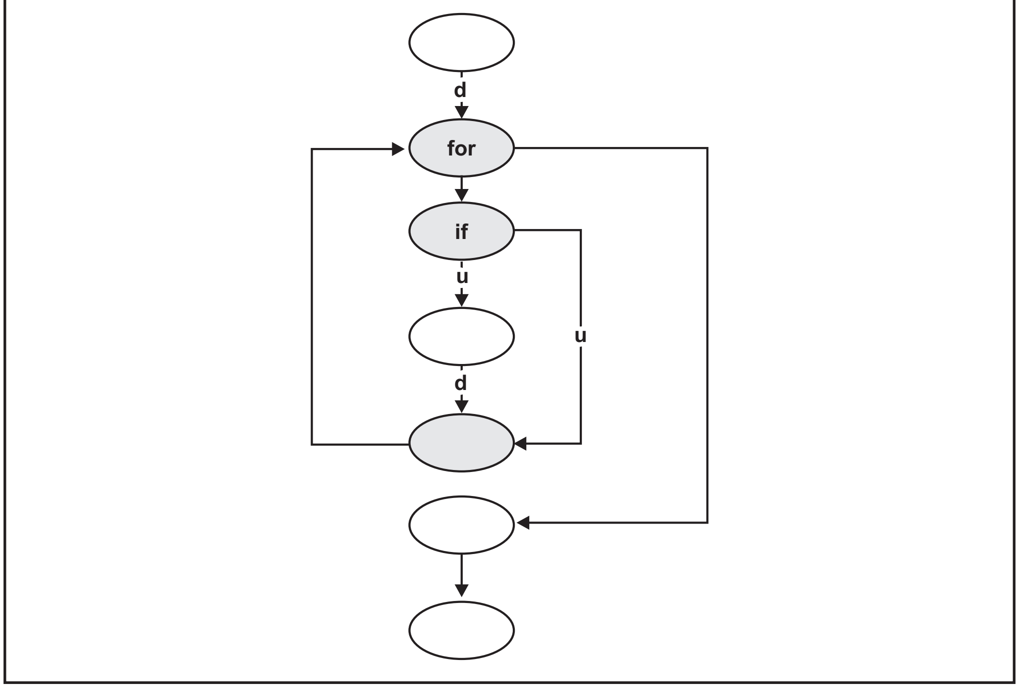
Harrold and Rothermel presented a technique for testing object-oriented software in class level. It is base on traditional data flood testing technique which is originally designed for unit testing of procedural program. Harrold and Rothermel extended the technique for object-oriented software by presenting a new idea for construction of control flow graph (CFG), while the part of the technique for creating test remained close to the traditional technique. Data flow testing is a well known testing technique for procedural programs. It is accepted as an effective technique for test case generation and also evaluation of test adequacy. The important concept of the technique is to test the program under test based on its data definitions and uses. For each piece of data in a program (usually a variable), it is defined somewhere in the program and gets used possibly in another place. For each variable, all definitions and uses are identified. As a variable is usually defined and then used later, a definition and a subsequent use without intervening definition forms a DU pair. Test execution is required to cover DU pairs for each variable in the program under test. How DU pairs are covered varies, as there are several criteria for this technique. For example, the all-use crition requires that for every use, there must be at least one DU pair of this use gets covered by the test execution. Another criterion, all-DU-path, requires that for every DU pair, every path between the def and the use must be covered by the test execution.

Data flow testing technique is also based on control flow of the program under test, in addition to data def/use. Testing with this technique requires a control flow graph of the program under test, which is constructed from the code of the program. For each variable, labels are put onto edges in the control flow graph to show where in the program, the variable is defined or used. A control flow graph is drawn to present control flow of this program as shown in figure 13.1. Labels are put on edges where there is a definition or a use of variable maxVal.

```
/**
  * Find the index of max value in an int array.
  */
Public int max (int [ ] data) {
      int maxVal = 0;
      int index = 0;
      for (int i = 0; i < data.length; i++) {
            if (data [ i ] > maxVal) {
            index = i;
            }
      }
      return index;
}
```

Notes

Figure 13.1: CFG with Def/use of Variable maxVal in Program



After the control flow graph is ready, DU pairs are identified. In this example, only variable maxVal is considered for test, so only DU pairs of the variable is considered. After DU pairs are identified, paths to be covered under tested are selected based on test adequacy criteria.

13.6.1 Test Data

Test data is the data that is used in tests of a software system. In order to test a software application we require entering some data for testing most of the features. Any such specifically identified data which is used in tests is known as test data. We can have test data in excel sheet which can be entered manually while executing test cases or it can be read automatically from files (XML, Flat Files, Database etc.) by automation tools. Some test data is used to confirm the expected result, when test data is entered the expected result should come and some test data is used to verify the software behaviour to invalid input data. Test data is generated by testers or by automation tools which support testing. Most of the times in regression testing the test data are reused; it is always a good practice to verify the test data before re using it in any kind of test.

Test data Categories

Design test data considering following categories:

No data: Run test cases on blank or default data. See if proper error messages are generated.

Valid data set: Create it to check if application is functioning as per requirements and valid input data is properly saved in database or files.

Invalid data set: Prepare invalid data set to check application behaviour for negative values, alphanumeric string inputs.

Illegal data format: Make one data set of illegal data format. System should not accept data in invalid or illegal format. Also check proper error messages are generated.

Notes

Boundary Condition data set: Data set containing out of range data. Identify application boundary cases and prepare data set that will cover lower as well as upper boundary conditions.

Data Set for Performance, Load and Stress Testing: This data set should be large in volume.

This way creating separate data sets for each test condition will ensure complete test coverage.

Preparing proper test data is a core part of "project test environment setup". Tester cannot pass the bug responsibility saying that complete data was not available for testing. Tester should create his/her own test data additional to the existing standard production data. We test data set should be ideal in terms of cost and time.

Caution Tests of data flow across a module interface are required before any other initiated. If data do not all other tests are moot.

Case Study

ATM Software Testing Case Study - NeST

NeST engineers are a global Leader in providing integrated technology solutions that enables customers to maximize their self service and security capabilities. They are a leading global supplier of ATMs, and hold the leading market position in many countries around the world offering an integrated line of self service banking solutions and ATMs. NeST engineers worked with two product versions. Modules worked are defined Security Camera, TCM, Touch screen and Seismic alarm, Fascia, Speaker, MssPci and Biometric Device.

After completing the development for each module it was tested by NeST engineers in the ATM system. Defects found were fixed and re tested to closure. Then deliverables of the same module are uploaded to Client's safe transfer site. When all the modules are released it was merged to single source code and released as single project for the particular model of ATM.

Situation

Several object files instead of actual source code (Not able to view or debug the code), Older version of Lab Windows which is technically limited. Clarification delay due to time zone difference Ensuring that new modifications are not affecting existing functionalities. NeST QA Engineers deployed for the assignment were having good testing, developing and debugging skills over Microsoft Visual Studio Platform which helped a lot to balance the demands. One module which found technically difficult in CVI was developed using Visual C++ and controlled from Test Specification Interface.

Questions

1. What is the situation of ATM software testing?

2. Explain the NeST engineers?

Self Assessment Questions

6. Top down approach is used for ....

   (*a*) testing and validation (*b*) identification of faults

   (*c*) development (*d*) reverse engineering

7. The main purpose of integration testing is to find
   (*a*) design errors (*b*) analysis errors
   (*c*) procedure errors (*d*) interface errors
8. Unit testing will be done by
   (*a*) Testers (*b*) End Users
   (*c*) Customer (*d*) Developers
9. What is correct Software Process Cycle?
   (*a*) Plan(P) >Check(C) >Act(A) >Do(D)
   (*b*) Plan(P) >Do(D) >Check(C) >Act(A)
   (*c*) Plan(P) >Do(D) >Act(A) >Check(C)\
   (*d*) None of these.
10. Which is Black Box Testing method?
    (*a*) Case validation (*b*) Code coverage
    (*c*) Fault injection (*d*) Equivalence partitioning
11. The testing which is done by going the code is known as
    (*a*) unit testing (*b*) black box testing
    (*c*) white box testing (*d*) regression testing
12. A fault simulation testing technique is
    (*a*) mutation testing (*b*) stress testing
    (*c*) black box testing (*d*) white box testing
13. fault simulation testing technique is
    (*a*) stress testing (*b*) mutation testing
    (*c*) black box testing (*d*) white box testing
14. The testing that focuses on the variables is called
    (*a*) data flow testing (*b*) white box testing
    (*c*) data variable testing (*d*) black box testing
15. What are the three generic phases of software engineering?
    (*a*) Definition, development, support (*b*) What, how, where
    (*c*) Programming, debugging, maintenance
    (*d*) Analysis, design, testing

13.7 Summary

- Software Quality Control is a set of activities for ensuring quality in software products.
- The oracle problem is addressed for random testing and testing of randomized software. The presented Statistical Oracle is a Heuristic Oracle using statistical methods, especially statistical tests.

- The Entry Exit Criteria matrix, we get the clarity of the deliverables expected from each phase.
- The black box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure.
- Boundary value analysis is a technique for test data selection. A test engineer chooses values that lie along data extremes.
- Test data is the data that is used in tests of a software system. In order to test a software application we need to enter some data for testing most of the features.

13.8 Keywords

Boundary Condition data set: Data set containing out of range data. Identify application boundary cases and prepare data set that will cover lower as well as upper boundary conditions.

Criteria: As a software organization grows and becomes more sophisticated by technology upgrades, it recognizes the need for Business Analysts and Quality Assurance to support their Information System functions.

Error: Refers to difference between Actual Output and Expected output.

Functional Testing: This black box testing type is related to functional requirements of a system; it is done by software testers.

Fundamentals: Software testing is a field with no set "best practices" Because so much of testing is based on the particular context of the test, It is often difficult to clarify, categorize and dispense advice on aspects of software testing.

Loop Testing: These strategies relate to testing single loops, concatenated loops, and nested loops. Independent and dependent code loops and values are tested by this approach.

Requirement: This is the initial stage of and in this stage requirement is gathered. Software testers also take part in this stage.

Test: Learn the various definitions of the term 'test'. Merriam Webster defines Test as "a critical examination, observation, or evaluation".

Lab Exercise

1. Write a program using verification and validation.
2. Draw the structure of testing.

13.9 Review Questions

1. What is the verification and validation?
2. Define software testing error and fault.
3. Explain the software development life cycle.
4. What is the black box testing?
5. Define software testing criteria.
6. What are the advantages and disadvantages of black box testing?
7. Discuss equivalence class partitioning.
8. What is the white box testing?
9. Define types of test case.
10. What is the test planning and analysis?

Answers for Self Assessment Questions

1.	(*a*)	2.	(*c*)	3.	(*b*)	4.	(*a*)	5.	(*b*)
6.	(*c*)	7.	(*d*)	8.	(*d*)	9.	(*b*)	10.	(*d*)
11.	(*c*)	12.	(*a*)	13.	(*b*)	14.	(*d*)	15.	(*a*)

13.10 Further Reading

Books *Software Testing,* by S. Koirala, S. Sheikh

Online link http://books.google.co.in/books?id=0y2U56ch9bUC&printsec=frontcover&dq=software+testing&hl=en&sa=X&ei=2AcJUK6PHszqrQeO3pTJCA&ved=0CFwQ6AEwBw#v=onepage&q=software%20testing&f=false

Unit 14: Flow Based Testing Process

Notes

CONTENTS

Objectives

After studying this unit, you will be able to:

- Discuss the levels of testing
- Define the test plan
- Discuss the test case specifications
- Explain execution and analysis
- Discuss the logging and tracking

Introduction

Software Testing is the process of executing a program or system with the aim of finding errors. Or, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its necessary results. Software is not unlike other physical processes where inputs are received and outputs are produced. Where software differs is in the manner in which it fails. Most physical systems fail in a permanent (and reasonably small) set of ways.

By contrast, software can fail in many bizarre ways. Detecting all of the different failure modes for software is generally infeasible.

Unlike most physical systems, most of the defects in software are design errors, not manufacturing defects. Software does not suffer from corrosion, wear-and generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects will be buried in and remain latent until activation.

Software bugs will almost always exist in any software module with moderate size: not because programmers are careless or irresponsible, but because the complexity of software is generally intractable and humans have only limited ability to manage complexity. It is also true that for any complex systems, design defects can never be completely ruled out are, is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible. Exhaustively testing a simple program to add only two integer inputs of 32-bits (yielding 2^{64} distinct test cases) would take hundreds of years, even if tests were performed at a rate of thousands per second. Obviously, for a realistic software module, the complexity can be far beyond the example mentioned here. If inputs from the real world are involved, the problem will get worse, because timing and random environmental effects and human interactions are all possible input parameters under consideration.

A further complication has to do with the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, the software may now work for a test case that it did not work for previously. But its behaviour on pre-error test cases that it passed before can no longer be guaranteed. To account for this possibility, testing should be restarted. The expense of doing this is often prohibitive. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bug.

14.1 Levels of Testing

Following are the three level of testing.

- Unit testing
- Integration testing
- System testing

14.1.1 Unit Testing

The tests that occur as part of element tests are illustrated schematically. The module boundary is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored provisionally maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

In ad local data structures should be exercised and the local impact on global data be ascertained (if possible) during unit testing. Discriminatory testing of execution paths is an essential task during the unit test cases should be designed to uncover errors due to erroneous computations, comparisons, or improper control flow.

Among the more common errors in computation are:

1. Misunderstood or incorrect arithmetic precedence,
2. Mixed mode operations,
3. Incorrect initialization,
4. Precision inaccuracy, and
5. Incorrect symbolic representation of an expression.

Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison.) Test cases should uncover errors such as:

1. Comparison of different data types,
2. Incorrect logical operators or precedence,
3. Expectation of equality when precision error makes equality unlikely,
4. Incorrect comparison of variables,
5. Improper or nonexistent loop termination,
6. Failure to exit when divergent iteration is encountered, and
7. Improperly modified loop variables.

Good design dictates that error conditions be anticipated and error handling set up to reroute or cleanly terminate processing when an error does occur. Your done calls this approach anti bugging. Unfortunately, there is a tendency to error handling into software and then never test it. A true story may serve to illustrate:

Unit Test Procedures

Unit testing is normally considered as an adjunct to the coding step. After basis level code has been developed, reviewed, and verified for correspondence to component level design, unit test case design begins. A review of design information provides because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent overhead. That is, both are software that must be written (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step.

14.1.2 Integration Testing

A neophyte in the software world might ask a seemingly legitimate question modules have been unit tested: "If they all work individually, why do you doubt they will work when we put them a together?" The problem, of course, is "putting together"-interfacing. Data can be lost across an interface; one module can have an inadvertent, adverse affect on another; sub functions, when combined, May not duce the desired major function; individually acceptable imprecision may be fined to unacceptable levels; global data structures can present problems.

Integration testing is a systematic technique for constructing the program true while at the same time conducting tests to uncover errors associated with in front of. The objective is to take unit tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt no incremental integration; that is, to structure the program using a "big bang" approaches. All components are advance. The entire program is tested as a whole. And chaos usually results of errors is encountered. Correction is difficult because isolation of causes is placated by the vast expanse of the entire program. Once these errors are new ones appear and the process continues in a seemingly endless loop.

Top-down Integration

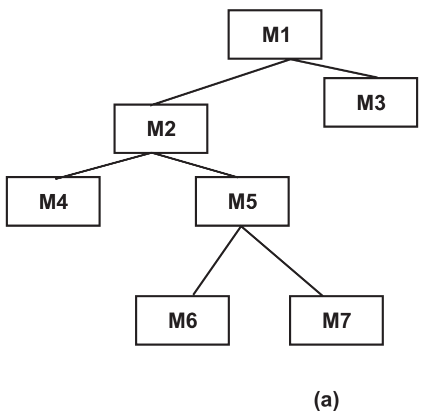
Top-down integration testing is an incremental approach to construction of structure. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated' structure in either a depth-first or breadth-first manner.

Depth-first integration would integrate all components on major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the hand path, components M_1, M_2, M_5 would be integrated first. Next, M_8 or (if necessary for proper functioning of M_2) M_6 would be integrated. Then, the central and right hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M_2, M_3, and M_4 (a replacement for stub S_4) would be integrated first. The next control level, M_5, M_6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

Figure 14.1: Top-down Integration



The process continues from step 2 until the entire program structure is built. The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential. If depth-first integration is selected, a complete function of the software may be implemented and demonstrated.

Bottom-up Integration

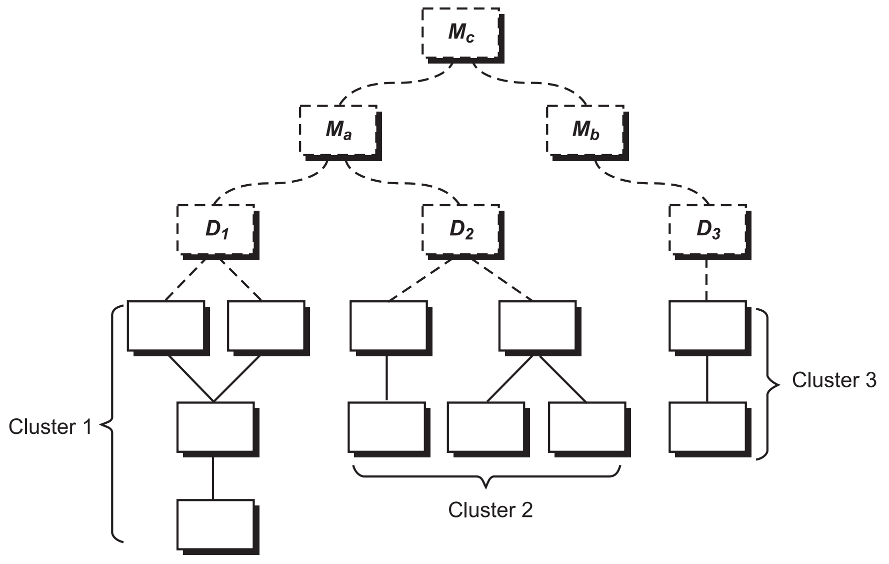
Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules i.e., components at the lowest levels in the Because components are integrated from the bottom up, processing points subordinate to a given level is always available and the eliminated.

A bottom-up integration strategy may be implemented with the following:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub function.
2. A driver (a control program for testing) is written to coordinate input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in gram structure.

Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a as a dashed block). Components in clusters 1 and 2 are subordinate to D_1 and D_2 are removed and the clusters are interfaced directly to lf. Driver D_3 for cluster 3 is removed prior to integration with module Mb. Mb will ultimately be integrated with component M, and so forth. (See Figure 14.2)

Figure 14.2: Bottom-up Integration



As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Regression Testing

Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new 1/0 may occur, and new control logic is invoked. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

In a broader context, successful tests (of any kind) result in the discovery of errors, errors must be corrected. Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) software is changed. Regression testing is the activity that helps to ensure that changes do not introduce unintended behaviour or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software.
- Additional tests that focus on software functions that are likely to by the change.
- Tests that focus on the software components that have been c.

As integration testing proceeds, the number of regression tests can grow. Therefore, the regression test suite should be designed to include only that address one or more classes of errors in each of the major program is impractical and inefficient to re-execute every test for every program a change has occurred.

Smoke Testing

Smoke testing is an integration testing approach that is commonly used when "shrink wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its frequent basis. In essence, the smoke testing approach encompasses the activities:

1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build properly performing its function. The intent should be to uncover per errors that have the highest likelihood of throwing the software behind schedule.
3. The build is integrated with other builds and the entire product is smoke tested daily. The integration approach may be top down bottom up.

Did u know? Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing.

14.1.3 System Testing

We stressed the information that software is one element of a superior computer based system. Software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.

A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem. Rather than indulging in such nonsense, the software engineer should anticipate potential interfacing problems and

1. Design error-handling paths that test all information coming from other elements of the system,

2. Conduct a series of tests that simulate bad data or other potential errors at the software interface,
3. Record the results of tests to use as "evidence" if finger-pointing does occur, and
4. Participate in planning and design of system tests to ensure that software is adequately tested.

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions. In the sections that follow, we discuss the types of system tests that are worthwhile for software-based systems.

Recovery Testing

Many computer based systems must get better from faults and resume processing within a pre-specified time. In some cases, a system must be fault tolerant; that is, processing faults must not cause overall system function to cease. In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), re-initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness. If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

Any computer-based system that manages sensitive information or causes proceedings that can improperly harm (or benefit) individuals is a target for improper or illegal diffusion. Penetration spans a broad range of activities: hackers who attempt to penetrate systems for sport; disgruntled employees who attempt to penetrate for revenge; dishonest individuals who attempt to penetrate for illicit personal increase.

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from indecent penetration. The system's security must, of course, be tested for invulnerability from frontal attack-but must also be tested for invulnerability from flank or rear attack."

During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Anything goes? The tester may attempt to acquire passwords through external clerical means; may attack the system with custom software designed to breakdown any defenses that have been constructed; may overwhelm the thereby denying service to others; may purposely cause system errors, hoping to penetrate during recovery; may browse through insecure data, hoping to find the system entry.

Stress Testing

During earlier software testing steps, white-box and black-box techniques resulted in thorough evaluation of normal program functions and performance. Stresses are designed to confront programs with abnormal situations. In essence, the taster who performs stress testing asks: How high can we crank this up before it fails.

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example:

1. special tests may be designed that generate ten interrupts per second, when one or two is the average rate,
2. Function data rates may be increased by an order of magnitude to determine how function will respond,

3. Test cases that require maximum memory or other resource are executed,
4. Test cases that may cause thrashing in a virtual operating are designed,
5. Test cases that may cause excessive hunting for disk-resident are created. Essentially, the tester attempts to break the program.

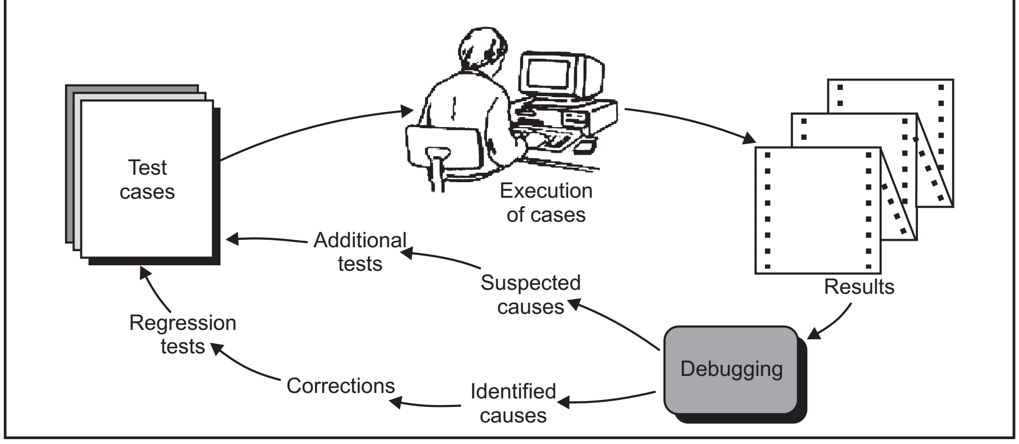
A variation of stress testing is a technique called sensitivity testing. In some situations (the most common occur in mathematical algorithms), a very small range data contained within the bounds of valid data for a program may cause extreme even erroneous processing or profound performance degradation. Sensitivity attempts to uncover data combinations within valid input classes that may work instability or improper processing.

Performance Testing

For real-time and embedded systems, software that provides required function does not conform to performance requirements is unacceptable. Performance is designed to test the run-time performance of software within the context of integrated system. Performance testing occurs throughout all steps in the process. Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted. However, it is not until all system ele¬ments are fully integrated that the true performance of a system can be ascertained.

Performance tests are often coupled with stress testing and typically require both hardware and software instrumentation. That is, it is often necessary to calculate resource utilization (e.g., processor cycles) in an exacting fashion.

Figure 14.3: The Debugging Process



External instrumentation can monitor education intervals, log events (e.g. interrupts) as they occur, in sample machine states on a regular basis, by incrementing a system, the tester can uncover situations that lead to degradation and possible system failure.

Task Write a test case for system testing.

Did u know? Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth. Mills developed structured programming concepts for practical use and tested them in a 1969 project to automate the New York Times morgue index.

14.2 Test Plan

Test plan is probably one of the most significant documents for software testing projects. Test plan may hold information associated to scope, environment, schedule, risk, resources, execution, reporting, automation, completion criteria etc.

Test plan is usually created by Test manager, Test lead or senior testers in the team. Before start preparing Test plan, information should be captured from various stakeholders of the project. Information captured from stakeholder is usually reflected in the Test plan.

Typically, every Test plan contains information about following activities. Information about these activities can be gathered from various stakeholders by asking questions that are required for your Test plan.

14.2.1 Definition of a Test Plan

A test plan can be defined as a document describing the scope, approach, resources, and schedule of planned testing activities. It identifies test items, the features to be tested, the testing tasks, who will do every task, and any risks requiring contingency planning.

In software testing, a test plan gives detailed testing information regarding an upcoming testing effort, including:

Scope Management: Before starting Test Planning activity, scope of the test activities should be known. You should have information on what features will be tested and what will not be tested. You should also have information on what areas your team owns? Are you taking care of all the types of testing that is required for the product including Performance, Security, globalization etc.? Defining scope for your testing project is very important for the management as well. If scope is properly defined, everyone will have clear understanding about what is tested and what is not.

Reference: You should clearly define documents you are referring to prepare test plan. Any changes in the documents you are referring should be reflected in your plan.

Risk Management: Test strategy is derived from the risk analysis. Risk will be different from one project to another and so your Test strategy. Risk associated with desktop tax calculation software will be different from payment gateway or life support system. In your testing strategy, you need to make sure that all the potential risks are captured and managed by your testing activities. You should, along with the other stake holders define what are the potential risk in the project? What will be the impact if these risks are materialized? What is the mitigation plan for these risks and how your testing activities are making sure that these risks are managed properly?

Test Environment: You should have information on what will be the environment for the testing. This information is captured from stake holders by asking them, what type of environment will be supported by product? What is the priority for these environments? For example, if product is supported on the entire platform and data for user distribution says that 80% are on Windows, 15 percent are on Linux and 5% are on Mac. From this data you can make out which platform will be tested more. Information captured here will be useful for planning hardware and software requirement for your Test Lab.

Criteria Definition: Criteria for Entry and Exit should be clearly defined for every activity of your testing project. You should have well defined entry/exit criteria for starting, stopping, suspending and resuming test activities. You should also have criteria defined for specifying when testing is complete.

Estimation, Scheduling and Resource Management: Mostly, testing project follows development activities. So for estimation and scheduling you should have information on the development plan and milestone. Once you have information on the development plan, you can schedule your testing activities accordingly. Resources in testing projects include hardware, software and people management.

Testing Tools and Automation: You should have information about what tools you are using to manage your testing activities. You should have information on the configuration management for test artifacts, test case management tool, defect tracking system, tools for automation etc. Ideally, test automation should be treated as separate project and you should have brief information here along with the link to automation plan.

It is also be described as a detail of how the testing will proceed, who will do the testing, what will be tested, in how much time the test will take place, and to what quality level the test will be performed.

Test plan is a document which includes introduction, assumptions, list of test cases, and list of features to be tested, approach, deliverables, resources, risks and scheduling.

A test plan is a systematic approach to testing a system such as a machine or software. The plan typically contains a detailed understanding of what the eventual workflow will be.

Caution Detecting all of the different failure modes for software is generally infeasible.

Self Assessment Questions

1. Among the more common errors in computation are:

    (*a*) Comparison of different data types

    (*b*) Precision inaccuracy

    (*c*) Incorrect logical operators or precedence

    (*d*) Incorrect comparison of variables

2. Integration testing is a systematic technique for constructing the program true while at the same time conducting tests to uncover errors associated with facing.

    (*a*) System testing (*b*) Unit testing

    (*c*) Integration testing (*d*) None of these

3. The .... strategy verifies major control or decision points early in the test process.

    (*a*) regression (*b*) bottom-up integration

    (*c*) smoke (*d*) top-down integration

4. A classic .... problem is finger-pointing.

    (*a*) recovery testing (*b*) system testing

    (*c*) security testing (*d*) stress testing

5. Test plan is a document which includes, introduction, assumptions, list of test cases, list of features to be tested, approach, deliverables, resources, risks and scheduling.

    (*a*) True (*b*) False

14.3 Test Case Specifications

Test cases are designed by the stepwise elaboration and refinement of the identified test conditions using test techniques identified in the test strategy. They should be repeatable, verifiable and noticeable back to requirements.

Test case design includes the identification of:

- The preconditions such as either project or localized test environment requirements and the plans for their delivery
- The test data requirements
- The expected results and post conditions.

A particular challenge is frequently the definition of the predictable result of a test; i.e., the identification of one or more test oracles that can be used for the test. In identifying the expected result, testers are concerned not only with outputs on the screen, but also with data and environmental post-conditions.

If the test basis is clearly defined, this may theoretically be simple. However, test bases are often vague, contradictory, lacking coverage of key areas, or plain missing. In such cases, a tester must have, or have access to, subject matter expertise. Also, even where the test basis is well specified, complex interactions of complex stimuli and responses can make the definition of expected results difficult, therefore a test oracle is essential. Test execution without any way to determine correctness of results has a very low added value or benefit, generating spurious incident reports and false confidence in the system.

The activities described above may be applied to all test levels, though the test basis will vary. For example, user acceptance tests may be based primarily on the requirements specification, use cases and defined business processes, while component tests may be based primarily on low-level design specification.

During the development of test conditions and test cases, some amount of documentation is typically performed resulting in test work products. A standard for such documentation is found in IEEE 829. This standard discusses the main document types applicable to test analysis and design, Test Design Specification and Test Case Specification, as well as test implementation. In practice the extent to which test work products are documented varies considerably. This can be impacted by, for example:

1. Project risks (what must/must not be documented)
2. The "value added" which the documentation brings to the project
3. Standards to be followed
4. Lifecycle model used (e.g. An agile approach tries to minimize documentation by ensuring close and frequent team communication)
5. The requirement for traceability from test basis, through test analysis and design.

14.4 Execution and Analysis

Execution testing determines whether the system achieves the preferred level of proficiency in a production position. Execution testing can confirm response times, turnaround times, as well as design performance. The execution of a system can be tested in whole or in part, using the actual system or a simulated model of a system.

14.4.1 Objectives

Execution testing is used to determine whether the system can meet the specific performance criteria. The objectives of execution testing include:

1. Determine the performance of the system structure.
2. Verify the optimum use of hardware and software.
3. Determine the response time to online user requests.
4. Determine transaction processing turnaround time.

14.4.2 How to Use?

Execution testing can be conducted in any phase of the system development life cycle. The testing can evaluate a single aspect of the system, for example, a critical routine in the system, or the ability of the proposed structure to satisfy performance criteria. Execution testing can be performed in any of the following manners:

1. Using hardware and software monitors
2. Simulating the functioning of all or part of the system using a simulation model
3. Creating a quick and dirty program(s) to evaluate the approximate performance of a completed system.

Execution testing may be executed on-site or off-site for the performance of the test. For example, execution testing can be performed on hardware and software before being acquired, or may be done after the application system has been completed. The earlier the technique is used, the higher the assurance that the completed application will meet the performance criteria.

Examples of the use of execution testing:

1. Calculating turnaround time on transactions processed through the application.
2. Determining that the hardware and software selected provide the optimum processing capability.
3. Using software monitors to determine that the program code is effectively used.

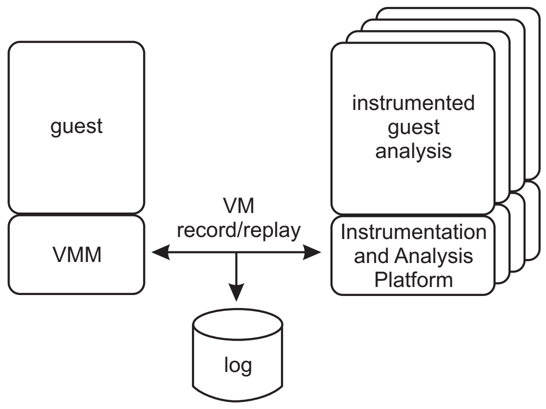
Dynamic program instrumentation and analysis enables many applications including intrusion detection and prevention, unfortunately, because these analyses are executed in line with program execution, they can substantially impact system performance, greatly reducing their utility. For example, analyses commonly used for detecting buffer overflows or use of undefined memory routinely incur overheads on the order of slowed by the analysis. Often analyses whose performance impact would be prohibitive if done inline can run with surprisingly minimal lag if run in parallel. Third, after sight can run analyses offline for situations where analyses are not known beforehand or are not time critical, such as when debugging. After sight is a general-purpose analysis framework. Any analysis that can run in the critical path of execution can run in after sight, as long as that analysis does not change the execution (this would break the determinism that after sight's replay mechanism relies upon). Also, after sight makes the entire system state at each instruction boundary available for analyses, providing greater generality than approaches based on sampling. Further, logs originating from the VMM can be replayed and analyzed in different execution environments (e.g., a simulator or VMM). This flexibility greatly eases program instrumentation and enables a variety of optimizations. We have implemented an after sight prototype on the x86 architecture, building on the record and replay capability of VMware Workstation. Our framework enables replay on the QEMU whole-system emulator, which supports easy instrumentation during replay and analysis. With this framework, we have implemented an online security analysis that can be used to detect buffer overflow attacks on running systems. We also implemented an analysis that can perform checks for memory safety and heap overflows, and we used this analysis to discover several new and serious bugs in VMware ESX Server, Linux, and Windows applications.

14.4.3 The Case for Decoupled Analysis

After sight improve dynamic analysis by decoupling the analysis from the main workload, while still providing the analysis with the identical, complete sequence of states from the main workload. This combination of decoupling and reproducibility improves dynamic analysis in the following ways. First, after sight allows analyses to be added to a running system without fear of breaking the main workload, because after sight runs analyses on a separate virtual machine from the main workload, new analyses can beaded without changing the running application, operating system, or virtual machine monitor of the main workload. Second, after sight offers users several choices along the safety/performance spectrum. Users who can tolerate some lag between the analysis and the workload can improve the performance of the workload and still get best-effort safety or offline analysis, while users who require synchronous safety can

Notes

Figure 14.4: Overview of Current System


guest
VMM
VM record/replay
log
instrumented guest analysis
Instrumentation and Analysis Platform

synchronize the output of the workload with the analysis. Third, with best-effort safety or offline analysis, after sight can improve latency for the main workload by moving.

14.5 Logging and Tracking

In the late 1960s, a bright-eyed young engineer was chosen to "write" a computer program for an automated manufacturing application. The reason for his selection was simple. He was the only person in his technical group who had attended a computer programming seminar. He knew the ins and outs of assembly language and FORTRAN but nothing about software engineering and even less about project scheduling and tracking. His boss gave him the appropriate manuals and a verbal description of what had to be done.

Earned value tracking: Do you report monthly earned value metrics? If so, are these metrics computed from an activity network of tasks for the entire effort to the next delivery?

Defect tracking against quality targets: Do you track and periodically report the number of defects found by each inspection (formal technical review) and execution test from program inception and the number of defects currently closed and open?

14.5.1 Tracking the Schedule

The project schedule provides a road map for a software project manager. If it has been properly developed, the project schedule defines the tasks and milestones that must be tracked and controlled as the project proceeds. Tracking can be accomplished in a number of different ways:

- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones have been accomplished by the scheduled date.
- Comparing actual start-date to planned start-date for each project task listed in the resource table. Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Using earned value analysis to assess progress quantitatively.

Notes

In reality, all of these tracking techniques are used by experienced project managers. Control is employed by a software project manager to administer project resources, cope with problems, and direct project staff. If things are going well (i.e., the project is on schedule and within budget, reviews indicate that real progress is being made and milestones are being reached), control is light. But when problems occur, the project manager must exercise control to reconcile them as quickly as possible. After a problem has been diagnosed, 10 additional resources may be focused on the problem area: staff may be redeployed or the project schedule can be redefined.

When faced with severe deadline pressure, experienced project managers sometimes use a project scheduling and control technique called time-boxing. The time-boxing strategy recognizes that the complete product may not be deliverable by the predefined deadline. Therefore, an incremental software paradigm is chosen and a schedule is derived for each incremental delivery.

The tasks associated with each increment are then time-boxed. This means that the schedule for each task is adjusted by working backward from the delivery date for the increment. A “box” is put around each task. When a task hits the boundary of its time box (plus or minus 10%), work stops and the next task begins. The initial reaction to the time-boxing approach is often negative: “If the work is not finished, how can we proceed?” The answer lies in the way work is accomplished.

14.5.2 Error Tracking

Throughout the software process, a project team creates work foodstuffs (e.g., requirements specifications or prototype, design documents, source code). But the team also creates (and hopefully corrects) errors associated with each work product. If error related measures and resultant metrics are collected over many software projects, a project manager can use these data as a baseline for comparison against error data collected in real time. Error tracking can be used as one means for assessing the status of a current project.

The concept of defect removal efficiency was discussed. To review briefly, the software team performs formal technical reviews (and, later, testing) to find and correct errors, E, in work products produced during software engineering tasks. Any errors that are not uncovered (but found in later tasks) are considered to be defects, D. Defect removal efficiency has been defined as

$$\text{DRE} = E/(E + D)$$

The DRE is a process metric that provides a strong indication of the effectiveness of quality assurance activities, but DRE and the error and defect counts associated with it can also be used to assist a project manager in determining the progress that is being made as a software project moves through its scheduled work tasks. Let us assume that a software organization has collected error and defect data over the past 24 months and has developed averages for the following metrics:

- Errors per requirements specification page, Ereq
- Errors per component—design level, Edesign
- Errors per component—code level, Ecode
- DRE—requirements analysis
- DRE—architectural design
- DRE—component level design
- DRE—coding.

As the project progresses through each software engineering step, the software team records and reports the number of errors found during requirements, design, and code reviews. The project manager calculates current values for Ereq, Edesign, and Ecode. These are then compared to averages for past projects. If current results vary by more than 20% from the average, there may be cause for concern and there is certainly use for investigation.

Task Define how to track a schedule.

Case Study

Software Testing

Software testing is the process of checking software, to verify that it satisfies its requirements and to detect errors. Software testing is an investigation conducted to provide the stakeholders with information about the quality of the product or service under test. Life has become dependent on software and software-based systems. Most of today's appliances, machines, and devices are completely or at least partly controlled by software. Administrative proceedings in state agencies and industry, too, rely to a large extent on highly complex IT systems. Our strong dependency on software requires ever higher investments in quality assurance activities to enable IT systems to perform reliably. Software testing is developing toward a specialized, independent field of study and professional discipline within the computer sciences. Within the discipline of software testing, "test management" is of particular importance. Test management comprises classical methods of project and risk management as well as knowledge of the appropriate use of well-defined test methods. With this stock-in-trade, the test manager can select and purposefully implement appropriate measures to ensure that a defined basic product quality will be achieved. In doing so, the test manager adopts an engineering approach. Without a structured approach that accounts for these factors, it is easy for testing to quickly disorganize. Testing can never completely establish the correctness of computer software. Instead, it furnishes a criticism or comparison that compares the state and behaviour of the product against a specification. Computer software has continued to grow in complexity and size. Every software product has a target audience. When an organization develops or invests in a software product, it presumably must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. Software testing is the process of attempting to make this assessment. A common practice of software testing is performed by an independent group of testers after the functionality is developed before it is shipped to the customer. This practice often results in the testing phase being used as project buffer to compensate for project delays, thereby compromising the time devoted to testing. Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Questions

1. Explain the importance of the software testing.
2. How to use testing process in a software project?

Self Assessment Questions

6. A further complication has to do with the dynamic nature of ………..

    (*a*) Application (*b*) programs

    (*c*) programming (*d*) testing

Notes

7. Software testing is the process of checking ………, to verify that it satisfies its requirements and to detect errors.

   (*a*) software (*b*) hardware

   (*c*) programs (*d*) testing

8. Using software monitors to determine that the program code is effectively ………..

   (*a*) used (*b*) unused

   (*c*) detection (*d*) system

9. …………..executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

   (*a*) Security testing (*b*) Function testing

   (*c*) Stress testing (*d*) System testing

10. …………. components are combined into clusters (sometimes called builds) that perform a specific software sub function.

    (*a*) Low-level (*b*) Middle-level

    (*c*) High-level (*d*) Any-level

11. Tests of data flow across a module interface are required before any other initiated. If data do not enter and ………… all other tests are moot.

    (*a*) check properly (*b*) testing properly

    (*c*) design properly (*d*) exit properly

12. Unit testing is normally considered as an adjunct to the………….step.

    (*a*) editing (*b*) coding

    (*c*) reading (*d*) deleting

13. Conducting periodic…………. status meetings in which each team member reports progress and problems.

    (*a*) testing (*b*) design

    (*c*) project (*d*) process

14. In the late 1960s, a bright-eyed young engineer was chosen to ………….. a computer program for an automated manufacturing application.

    (*a*) read (*b*) write

    (*c*) return (*d*) test

15. Integration testing is a systematic technique for constructing the program ………. while at the same time conducting tests to uncover errors associated with facing.

    (*a*) True (*b*) False

14.6 Summary

- Software Testing is the process of executing a program or system with the intent of finding errors.
- The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- Good design dictates that error conditions be anticipated and error handling set up to reroute or cleanly terminate processing when an error does occur.
- Stubs serve to replace modules that are subordinate (called by) the component to be tested.
- Integration testing is a systematic technique for constructing the program true while at the same time conducting tests to uncover errors associated with facing.
- A test plan can be defined as a document describing the scope, approach, resources, and schedule of intended testing activities.
- Test cases are designed by the stepwise elaboration and refinement of the identified test conditions using test techniques identified in the test strategy.

14.7 Keywords

Execution Testing: Execution testing can be conducted in any phase of the system development life cycle.

Quick Emulator (QEMU): The QEMU stands for "Quick Emulator" and is a processor emulator that relies on dynamic binary translation to achieve a reasonable speed while being easy to port to new host CPU architectures.

Recovery Testing: It is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

Security Testing: It attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.

Stress Testing: It executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

Lab Exercise

1. Prepare a flow chart for the states of flow based testing.
2. Write a test case for testing software.

14.8 Review Questions

1. Explain the flow based testing process.
2. How many levels of testing?
3. What is the different between the top-down and bottom-up integration testing approaches?
4. What is the system testing? How to different the integration testing?
5. Explain the recovery testing, security testing, stress testing, performance testing.
6. How do we plan a testing process?
7. Discuss the test case specifications.
8. What is the execution and analysis? How to use the execution testing with example?
9. Discuss the logging and tracking.
10. Define error tracking.

Notes

Answers for Self Assessment Questions

1.	(*a*)	2.	(*c*)	3.	(*d*)	4.	(*b*)	5.	(*a*)
6.	(*b*)	7.	(*b*)	8.	(*a*)	9.	(*c*)	10.	(*a*)
11.	(*d*)	12.	(*b*)	13.	(*c*)	14.	(*b*)	15.	(*a*)

14.9 Further Reading

Books *The Art of Software Testing*, by Glen ford J. Myers, Corey Sandler, Tom Badgett

Online link http://books.google.co.in/books?id=9dskoe_4n_UC&printsec=frontcover&dq=Testing&hl=en&sa=X&ei=k_0HUPefCMnSrQfCoPXkAg&redir_esc=y#v=onepage&q=Testing&f=false

LOVELY PROFESSIONAL UNIVERSITY
Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

978-93-90164-80-6
9 789390 164806