Software Engineering

Why Software Engineering ?

- Change in nature & complexity of software
- Concept of one "guru" is over
- We all want improvement

Ready for change

The Evolving Role of Software

- Software industry is in Crisis!

Source: The Standish Group International, Inc. (CHAOS research)

The Evolving Role of Software

This is the SORRY state of Software Engineering Today!

- **Data on 28,000 projects completed in 2000**

The Evolving Role of Software

As per the IBM report, "31%of the project get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% and for every 100 projects, there are 94 restarts".

The Evolving Role of Software

Relative Cost of Hardware and Software

The Evolving Role of Software

- Unlike Hardware
  - Moore's law: processor speed/memory capacity doubles every two years

The Evolving Role of Software

***Managers* and *Technical Persons* are asked:**

- ✓ Why does it take so long to get the program finished?
- ✓ Why are costs so high?
- ✓ Why can not we find all errors before release?
- ✓ Why do we have difficulty in measuring progress of software development?

Factors Contributing to the Software Crisis

- Larger problems,
- Lack of adequate training in software engineering,
- Increasing skill shortage,
- Low productivity improvements.

Some Software failures

Ariane 5

It took the European Space Agency **10 years and $7 billion** to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business.

The rocket was destroyed after 39 seconds of its launch, at an altitude of two and a half miles along with its payload of four expensive and uninsured scientific satellites.

Some Software failures

When the guidance system's own computer tried to convert one piece of data the sideways velocity of the rocket from a 64 bit format to a 16 bit format; the number was too big, and an overflow error resulted after 36.7 seconds. When the guidance system shutdown, it passed control to an identical, redundant unit, which was there to provide backup in case of just such a failure. Unfortunately, the second unit, which had failed in the identical manner a few milliseconds before.

Some Software failures

Y2K problem:

It was simply the ignorance about the adequacy or otherwise of using only last two digits of the year.

The 4-digit date format, like 1964, was shortened to 2-digit format, like 64.

Some Software failures

The Patriot Missile

- First time used in Gulf war
- Used as a defense from Iraqi Scud missiles
- Failed several times including one that killed 28 US soldiers in Dhahran, Saudi Arabia

Reasons:

A small timing error in the system's clock accumulated to the point that after 14 hours, the tracking system was no longer accurate. In the Dhahran attack, the system had been operating for more than 100 hours.

Some Software failures

The Space Shuttle

Part of an abort scenario for the Shuttle requires fuel dumps to lighten the spacecraft. It was during the second of these dumps that a (software) crash occurred.

...the fuel management module, which had performed one dump and successfully exited, restarted when recalled for the second fuel dump...

Some Software failures

A simple fix took care of the problem…but the programmers decided to see if they could come up with a systematic way to eliminate these generic sorts of bugs in the future. A random group of programmers applied this system to the fuel dump module and other modules.

Seventeen additional, previously unknown problems surfaced!

Some Software failures

Financial Software

Many companies have experienced failures in their accounting system due to faults in the software itself. The failures range from producing the wrong information to the whole system crashing.

Some Software failures

Windows XP

- Microsoft released Windows XP on October 25, 2001.
- On the same day company posted 18 MB of compatibility patches on the website for bug fixes, compatibility updates, and enhancements.
- Two patches fixed important security holes.

This is **Software Engineering.**

"No Silver Bullet"

The hardware cost continues to decline drastically.

However, there are desperate cries for a silver bullet something to make software costs drop as rapidly as computer hardware costs do.

But as we look to the horizon of a decade, we see no silver bullet. There is no single development, either in technology or in management technique, that by itself promises even one order of magnitude improvement in productivity, in reliability and in simplicity.

"No Silver Bullet"

The hard part of building software is the specification, design and testing of this conceptual construct, not the labour of representing it and testing the correctness of representation.

We still make syntax errors, to be sure, but they are trivial as compared to the conceptual errors (logic errors) in most systems. That is why, building software is always hard and there is inherently no silver bullet.

While there is no royal road, there is a path forward.

Is reusability (and open source) the new silver bullet?

"No Silver Bullet"

The blame for software bugs belongs to:

- Software companies
- Software developers
- Legal system
- Universities

What is software?

- **Computer programs** and **associated documentation**

What is software?

Software=Program+Documentation+Operating Procedures

Components of software

Documentation consists of different types of manuals are

List of documentation manuals

Documentation consists of different types of manuals are

List of operating procedure manuals.

Software Product

- **Software products** may be developed for a particular customer or may be developed for a general market

- **Software products** may be
  - **Generic** - developed to be sold to a range of different customers
  - **Bespoke** (custom) - developed for a single customer according to their specification

Software Product

Software product is a product designated for delivery to the user

What is software engineering?

Software engineering is an engineering discipline which is concerned with all aspects of software production

Software engineers should

- adopt a systematic and organised approach to their work
- use appropriate tools and techniques depending on
  - the problem to be solved,
  - the development constraints and
- use the resources available

What is software engineering?

At the first conference on software engineering in 1968, Fritz Bauer defined software engineering as "The establishment and use of sound engineering principles in order to obtain economically developed software that is reliable and works efficiently on real machines".

Stephen Schach defined the same as "A discipline whose aim is the production of quality software, software that is delivered on time, within budget, and that satisfies its requirements".

Both the definitions are popular and acceptable to majority. However, due to increase in cost of maintaining software, objective is now shifting to produce quality software that is maintainable, delivered on time, within budget, and also satisfies its requirements.

Software Process

The software process is the way in which we produce software.

Why is it difficult to improve software process ?

- Not enough time
- Lack of knowledge

Software Process

- Wrong motivations
- Insufficient commitment

Software Characteristics:

✓Software does not wear out.

Software Characteristics:

- ✓ Software is not manufactured
- ✓ Reusability of components
- ✓ Software is flexible

Software Characteristics:

Comparison of constructing a bridge vis-à-vis 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 requirement 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.

The Changing Nature of Software

The Changing Nature of Software

Trend has emerged to provide source code to the customer and organizations.

Software where source codes are available are known as open source software.

Examples

Open source software: LINUX, MySQL, PHP, Open office, Apache webserver etc.

Software Myths (Management Perspectives)

Management may be confident about good standards and clear procedures of the company.

But the taste of any food item is in the eating; not in the Recipe !

Software Myths (Management Perspectives)

Company has latest computers and state-of-the-art software tools, so we shouldn't worry about the quality of the product.

The infrastructure is only one of the several factors that determine the quality of the product!

Software Myths (Management Perspectives)

Addition of more software specialists, those with higher skills and longer experience may bring the schedule back on the track!

Unfortunately,
that may further delay the schedule!

Software Myths (Management Perspectives)

Software is easy to change

The reality is totally different.

Software Myths (Management Perspectives)

Computers provide greater reliability than the devices they replace

This is not always true.

Software Myths (Customer Perspectives)

A general statement of objectives is sufficient to get started with the development of software. Missing/vague requirements can easily be incorporated/detailed out as they get concretized.

If we do so, we are heading towards a disaster.

Software Myths (Customer Perspectives)

Software with more features is better software

Software can work right the first time

Both are only myths!

Software Myths (Developer Perspectives)

Once the software is demonstrated, the job is done.

Usually, the problems just begin!

Software Myths (Developer Perspectives)

Software quality can not be assessed before testing.

However, quality assessment techniques should be used through out the software development life cycle.

Software Myths (Developer Perspectives)

The only deliverable for a software development project is the tested code.

Tested code is only one of the deliverable!

Software Myths (Developer Perspectives)

Aim is to develop working programs

Those days are over. Now objective is to develop good quality maintainable programs!

Some Terminologies

- Deliverables and Milestones

Different deliverables are generated during software development. The examples are source code, user manuals, operating procedure manuals etc.

The milestones are the events that are used to ascertain the status of the project. Finalization of specification is a milestone. Completion of design documentation is another milestone. The milestones are essential for project planning and management.

Some Terminologies

- Product and Process

Product: 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.

Process: Process is the way in which we produce software. It is the collection of activities that leads to (a part of) a product. An efficient process is required to produce good quality products.

If the process is weak, the end product will undoubtedly suffer, but an obsessive over reliance on process is also dangerous.

Some Terminologies

- Measures, Metrics and Measurement

A measure provides a quantitative indication of the extent, dimension, size, capacity, efficiency, productivity or reliability of some attributes of a product or process.

Measurement is the act of evaluating a measure.

A metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute.

Some Terminologies

- Software Process and Product Metrics

Process metrics quantify the attributes of software development process and environment;

whereas product metrics are measures for the software product.

Examples

Process metrics: Productivity, Quality, Efficiency etc.

Product metrics: Size, Reliability, Complexity etc.

Some Terminologies

- Productivity and Effort

Productivity is defined as the rate of output, or production per unit of effort, i.e. the output achieved with regard to the time taken but irrespective of the cost incurred.

Hence most appropriate unit of effort is Person Months (PMs), meaning thereby number of persons involved for specified months. So, productivity may be measured as LOC/PM (lines of code produced/person month)

Some Terminologies

- Module and Software Components

There are many definitions of the term module. They range from "a module is a FORTRAN subroutine" to "a module is an Ada Package", to "Procedures and functions of PASCAL and C", to "C++ Java classes" to "Java packages" to "a module is a work assignment for an individual developer". All these definition are correct. The term subprogram is also used sometimes in place of module.

Some Terminologies

"An independently deliverable piece of functionality providing access to its services through interfaces".

"A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces".

Some Terminologies

- Generic and Customized Software Products

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 system, compilers, analyzers, 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 development projects (say about 80%)come under this category.

Role of Management in Software Development

Role of Management in Software Development

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

1.1 Software is

(a) Superset of programs (b) subset of programs
(c) Set of programs (d) none of the above

1.2 Which is NOT the part of operating procedure manuals?

(a) User manuals (b) Operational manuals
(c) Documentation manuals (d) Installation manuals

1.3 Which is NOT a software characteristic?

(a) Software does not wear out (b) Software is flexible
(c) Software is not manufactured (d) Software is always correct

1.4 Product is

(a) Deliverables (b) User expectations
(c) Organization's effort in development (d) none of the above

1.5 To produce a good quality product, process should be

(a) Complex (b) Efficient
(c) Rigorous (d) none of the above

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

1.6 Which is not a product metric?

(a) Size (b) Reliability
(c) Productivity (d) Functionality

1.7 Which is NOT a process metric?

(a) Productivity (b) Functionality
(c) Quality (d) Efficiency

1.8 Effort is measured in terms of:

(a) Person-months (b) Rupees
(c) Persons (d) Months

1.9 UML stands for

(a) Uniform modeling language (b) Unified modeling language
(c) Unit modeling language (d) Universal modeling language

1.1 An independently deliverable piece of functionality providing access to its services through interface is called

(a) Software measurement (b) Software composition
(c) Software measure (d) Software component

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

1.11 Infrastructure software are covered under

(a) Generic products
(b) Customized products
(c) Generic and Customized products
(d) none of the above

1.12 Management of software development is dependent on

(a) people
(b) product
(c) process
(d) all of the above

1.13 During software development, which factor is most crucial?

(a) People
(b) Product
(c) Process
(d) Project

1.14 Program is

(a) subset of software
(b) super set of software
(c) software
(d) none of the above

1.15 Milestones are used to

(a) know the cost of the project
(b) know the status of the project
(c) know user expectations
(d) none of the above

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

1.16 The term module used during design phase refers to

(a) Function (b) Procedure
(c) Sub program (d) All of the above

1.17 Software consists of

(a) Set of instructions + operating system
(b) Programs + documentation + operating procedures
(c) Programs + hardware manuals (d) Set of programs

1.18 Software engineering approach is used to achieve:

(a) Better performance of hardware (b) Error free software
(c) Reusable software (d) Quality software product

1.19 Concept of software engineering are applicable to

(a) Fortran language only (b) Pascal language only
(c) 'C' language only (d) All of the above

1.20 CASE Tool is

(a) Computer Aided Software Engineering (b) Component Aided Software Engineering
(c) Constructive Aided Software Engineering (d)Computer Analysis Software Engineering

Exercises

1.1 Why is primary goal of software development now shifting from producing good quality software to good quality maintainable software?

1.2 List the reasons for the "software crisis"?Why are CASE tools not normally able to control it?

1.3 "The software crisis is aggravated by the progress in hardware technology?" Explain with examples.

1.4 What is software crisis? Was Y2K a software crisis?

1.5 What is the significance of software crisis in reference to software engineering discipline.

1.6 How are software myths affecting software process? Explain with the help of examples.

1.7 State the difference between program and software. Why have documents and documentation become very important.

1.8 What is software engineering? Is it an art, craft or a science? Discuss.

Exercises

1.9 What is aim of software engineering? What does the discipline of software engineering discuss?

1.10 Define the term "Software engineering". Explain the major differences between software engineering and other traditional engineering disciplines.

1.11 What is software process? Why is it difficult to improve it?

1.12 Describe the characteristics of software contrasting it with the characteristics of hardware.

1.13 Write down the major characteristics of a software. Illustrate with a diagram that the software does not wear out.

1.14 What are the components of a software? Discuss how a software differs from a program.

1.15 Discuss major areas of the applications of the software.

1.16 Is software a product or process? Justify your answer with example

Exercises

1.17 Differentiate between the following

(i) Deliverables and milestones (ii) Product and process

(iii) Measures, metrics and measurement

1.18 What is software metric? How is it different from software measurement

1.19 Discuss software process and product metrics with the help of examples.

1.20 What is productivity? How is it related to effort. What is the unit of effort.

1.21 Differentiate between module and software component.

1.22 Distinguish between generic and customized software products. Which one has larger share of market and why?

1.23 Is software a product or process? Justify your answer with example

Exercises

1.23 Describe the role of management in software development with the help of examples.

1.24 What are various factors of management dependency in software development. Discuss each factor in detail.

1.25 What is more important: Product or process? Justify your answer.

Software Certification

Software Certification

- What is certification?
- Why should we really need it?
- Who should carry out this activity?
- Where should we do such type of certification?

Software Certification

To whom should we target

- People
- Process
- Product

We have seen many certified developers (Microsoft certified, Cisco certified, JAVA certified), certified processes (like ISO or CMM) and certified products.

There is no clarity about the procedure of software certification.

Requirement of Certification

Adam Kalawa of Parasoft has given his views on certification like:

"I strongly oppose certification of software developers. I fear that it will bring more harm than good to the software industry. It may further hurt software quality by shifting the blame for bad software. The campaign for certification assumes that unqualified developers cause software problem and that we can improve software quality by ensuring that all developers have the golden stamp of approval. However, improving quality requires improving the production process and integrating in to it practices that reduce the opportunity for introducing defects into the product"

Requirement of Certification

- How often will developers require certification to keep pace with new technologies?
- How will any certification address the issues like fundamentals of computer science, analytical & logical reasoning, programming aptitude & positive attitude?
- Process certification alone cannot guarantee high quality product.
- Whether we go for certified developers or certified processes?

Can independent certification agency provide a fair playing field for each software industry??

Types of Certification

- People
  - Industry specific
- Process
  - Industry specific
- Product
  - For the customer directly and helps to select a particular product

Certification of Persons

The individual obtaining certification receives the following values:

- Recognition by peers
- Increased confidence in personal capabilities
- Recognition by software industry for professional achievement
- Improvement in processes
- Competences maintained through recertification

Certification is employees initiated improvement process which improves competence in quality assurances methods & techniques.

Certification of Persons

Professional level of competence in the principles & practices of software quality assurance in the software industry can be achieved by acquiring the designation of:

- Certified Software Quality Analyst (CSQA)
- Certified Software Tester (CSTE)
- Certified Software Project Manager (CSPM)

Some company specific certifications are also very popular like Microsoft Office Specialist (MOS) certifications in Word, Excel and PowerPoint.

MOS is far best known computer skills certification for administrator.

Certification of Processes

The most popular process certification approaches are:

- ISO 9000
- SEI-CMM

One should always be suspicious about the quality of end product, however, certification reduces the possibility of poor quality products.

Any type of process certification helps to produce good quality and stable software product.

Certification of Products

- This is what is required for the customer.
- There is no universally accepted product certification scheme.
- Aviation industry has a popular certification "RTCA DO-178B".
- The targeted certification level is either A, B, C, D, or E.
- These levels describe the consequences of a potential failure of the software : catastrophic, hazardous severe, major, minor or no effect.

Certification of Products

DO-178B Records

Software Development Plan

Software Verification Plan

Software Configuration Management Plan

Software Quality Assurance Plan

Software Requirements Standards

Software Design Document

Software Verification Test Cases & Products

Certification of Products

DO-178B Documents

Software Verification Results

Problem Report

Software Configuration Management Records

Software Quality Assurance Records

DO-178B certification process is most demanding at higher levels.

Certification of Products

DO-178B level A will:

1. Have largest potential market
2. Require thorough labour intensive preparation of most of the items on the DO-178B support list.

DO-178B Level E would:

1. Require fewer support item and
2. Less taxing on company resources.

Certification of Products

We don't have product certification in most of the areas. RTOS (real time operating system) is the real-time operating system certification & marked as "LinuxOS-178".

The establishment of independent agencies is a viable option.

Third Party Certification for Component base Software Engineering

Weyukar has rightly said "For Component based Software Development (CBO) to revolutionalize software development, developers must be able to produce software significantly cheaper and faster than they otherwise could, even as the resulting software meets the same sort of high reliability standards while being easy to maintain".

Bill council has also given his views as "Currently, there is a little evidences that component based software engineering (CBSE) is revolutionizing software development, and lots of reasons to believe otherwise. I believe the primary reason is that the community is not showing how to develop trusted components".

Third Party Certification for Component base Software Engineering

Contractor:

- Gives the standard
- Directs any variations in specification
- Define patterns
- Allowable tolerances
- Fix the date of delivery

Third party certification is a method to ensure software components conform to well defined standards, based on this certification, trusted assemblies of components can be constructed

Third party certification is based on UL 1998, 2nd ed., UL standard for safety for software in programmable component.

Exercises

10.1 What is software certification? Discuss its importance in the changing scenario of software industry.

10.2 What are different types of certifications? Explain the significance of each type & which one is most important for the end user.

10.3 What is the role of third party certification in component based software engineering? Why are we not able to stabilize the component based software engineering practices.

10.4 Name few person specific certification schemes. Which one is most popular & why?

10.5 Why customer is only interested in product certification? Discuss any product certification techniques with their generic applicability.

Software Life Cycle Models

Software Life Cycle Models

The goal of Software Engineering is to provide models and processes that lead to the production of well-documented maintainable software in a manner that is predictable.

Software Life Cycle Models

"The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirement phase, design phase, implementation phase, test phase, installation and check out phase, operation and maintenance phase, and sometimes retirement phase".

Build & Fix Model

- Product is constructed without specifications or any attempt at design
- Adhoc approach and not well defined
- Simple two phase model

Build & Fix Model

- Suitable for small programming exercises of 100 or 200 lines
- Unsatisfactory for software for any reasonable size
- Code soon becomes unfixable & unenhanceable
- No room for structured design
- Maintenance is practically not possible

Waterfall Model

This model is named "waterfall model" because its diagrammatic representation resembles a cascade of waterfalls.

Waterfall Model

This model is easy to understand and reinforces the notion of "define before design" and "design before code".

The model expects complete & accurate requirements early in the process, which is unrealistic

Waterfall Model

Problems of waterfall model

i. It is difficult to define all requirements at the beginning of a project

ii. This model is not suitable for accommodating any change

iii. A working version of the system is not seen until late in the project's life

iv. It does not scale up well to large projects.

v. Real projects are rarely sequential.

Incremental Process Models

They are effective in the situations where requirements are defined precisely and there is no confusion about the functionality of the final product.

After every cycle a useable product is given to the customer.

Popular particularly when we have to quickly deliver a limited functionality system.

Iterative Enhancement Model

This model has the same phases as the waterfall model, but with fewer restrictions. Generally the phases occur in the same order as in the waterfall model, but they may be conducted in several cycles. Useable product is released at the end of the each cycle, with each release providing additional functionality.

- ✓ Customers and developers specify as many requirements as possible and prepare a SRS document.
- ✓ Developers and customers then prioritize these requirements
- ✓ Developers implement the specified requirements in one or more cycles of design, implementation and test based on the defined priorities.

Iterative Enhancement Model

The Rapid Application Development (RAD) Model

- Developed by IBM in 1980
- User participation is essential

The requirements specification was defined like this

The developers understood it in that way

This is how the problem was solved before.

This is how the problem is solved now

That is the program after debugging

This is how the program is described by marketing department

This, in fact, is what the customer wanted …

The Rapid Application Development (RAD) Model

- Build a rapid prototype
- Give it to user for evaluation & obtain feedback
- Prototype is refined

With active participation of users

The Rapid Application Development (RAD) Model

Not an appropriate model in the absence of user participation.

Reusable components are required to reduce development time.

Highly specialized & skilled developers are required and such developers are not easily available.

Evolutionary Process Models

Evolutionary process model resembles iterative enhancement model. The same phases as defined for the waterfall model occur here in a cyclical fashion. This model differs from iterative enhancement model in the sense that this does not require a useable product at the end of each cycle. In evolutionary development, requirements are implemented by category rather than by priority.

This model is useful for projects using new technology that is not well understood. This is also used for complex projects where all functionality must be delivered at one time, but the requirements are unstable or not well understood at the beginning.

Evolutionary Process Model

Prototyping Model

- The prototype may be a usable program but is not suitable as the final software product.
- The code for the prototype is thrown away. However experience gathered helps in developing the actual system.
- The development of a prototype might involve extra cost, but overall cost might turnout to be lower than that of an equivalent system developed using the waterfall model.

Prototyping Model

- Linear mode
- "Rapid"

Spiral Model

Models do not deal with uncertainly which is inherent to software projects.

Important software projects have failed because project risks were neglected & nobody was prepared when something unforeseen happened.

Barry Boehm recognized this and tired to incorporate the "project risk" factor into a life cycle model.

The result is the spiral model, which was presented in 1986.

Spiral Model

Spiral Model

The radial dimension of the model represents the cumulative costs. Each path around the spiral is indicative of increased costs. The angular dimension represents the progress made in completing each cycle. Each loop of the spiral from X-axis clockwise through 360° represents one phase. One phase is split roughly into four sectors of major activities.

- **Planning:** Determination of objectives, alternatives & constraints.
- **Risk Analysis:** Analyze alternatives and attempts to identify and resolve the risks involved.
- **Development:** Product development and testing product.
- **Assessment:** Customer evaluation

Spiral Model

- An important feature of the spiral model is that each phase is completed with a review by the people concerned with the project (designers and programmers)
- The advantage of this model is the wide range of options to accommodate the good features of other life cycle models.
- It becomes equivalent to another life cycle model in appropriate situations.

The spiral model has some difficulties that need to be resolved before it can be a universally applied life cycle model. These difficulties include lack of explicit process guidance in determining objectives, constraints, alternatives; relying on risk assessment expertise; and provides more flexibility than required for many applications.

The Unified Process

- Developed by I.Jacobson, G.Booch and J.Rumbaugh.
- Software engineering process with the goal of producing good quality maintainable software within specified time and budget.
- Developed through a series of fixed length mini projects called iterations.
- Maintained and enhanced by Rational Software Corporation and thus referred to as Rational Unified Process (RUP).

Phases of the Unified Process

Definition of objectives of the project

Planning & architecture for the project

Initial operational capability

Release of the Software product

Phases of the Unified Process

- Inception: defines scope of the project.
- Elaboration

  - How do we plan & design the project?

  - What resources are required?

  - What type of architecture may be suitable?
- Construction: the objectives are translated in design & architecture documents.
- Transition : involves many activities like delivering, training, supporting, and maintaining the product.

Initial development & Evolution Cycles

V1=version1, V2 =version2, V3=version3

Iterations & Workflow of Unified Process

Inception Phase

The inception phase has the following objectives:

- Gathering and analyzing the requirements.
- Planning and preparing a business case and evaluating alternatives for risk management, scheduling resources etc.
- Estimating the overall cost and schedule for the project.
- Studying the feasibility and calculating profitability of the project.

Outcomes of Inception Phase

Elaboration Phase

The elaboration phase has the following objectives:

- Establishing architectural foundations.
- Design of use case model.
- Elaborating the process, infrastructure & development environment.
- Selecting component.
- Demonstrating that architecture support the vision at reasonable cost & within specified time.

Outcomes of Elaboration Phase

Construction Phase

The construction phase has the following objectives:

- Implementing the project.
- Minimizing development cost.
- Management and optimizing resources.
- Testing the product
- Assessing the product releases against acceptance criteria

Outcomes of Construction Phase

Transition Phase

The transition phase has the following objectives:

- Starting of beta testing
- Analysis of user's views.
- Training of users.
- Tuning activities including bug fixing and enhancements for performance & usability
- Assessing the customer satisfaction.

Outcomes of Transition Phase

Selection of a Life Cycle Model

Selection of a model is based on:

a) Requirements

b) Development team

c) Users

d) Project type and associated risk

Based On Characteristics Of Requirements

Requirements	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Are requirements easily understandable and defined?	Yes	No	No	No	No	Yes
Do we change requirements quite often?	No	Yes	No	No	Yes	No
Can we define requirements early in the cycle?	Yes	No	Yes	Yes	No	Yes
Requirements are indicating a complex system to be built	No	Yes	Yes	Yes	Yes	No

Based On Status Of Development Team

Development team	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Less experience on similar projects?	No	Yes	No	No	Yes	No
Less domain knowledge (new to the technology)	Yes	No	Yes	Yes	Yes	No
Less experience on tools to be used	Yes	No	No	No	Yes	No
Availability of training if required	No	No	Yes	Yes	No	Yes

Based On User's Participation

Involvement of Users	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
User involvement in all phases	No	Yes	No	No	No	Yes
Limited user participation	Yes	No	Yes	Yes	Yes	No
User have no previous experience of participation in similar projects	No	Yes	Yes	Yes	Yes	No
Users are experts of problem domain	No	Yes	Yes	Yes	No	Yes

Based On Type Of Project With Associated Risk

Project type and risk	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Project is the enhancement of the existing system	No	No	Yes	Yes	No	Yes
Funding is stable for the project	Yes	Yes	No	No	No	Yes
High reliability requirements	No	No	Yes	Yes	Yes	No
Tight project schedule	No	Yes	Yes	Yes	Yes	Yes
Use of reusable components	No	Yes	No	No	Yes	Yes
Are resources (time, money, people etc.) scare?	No	Yes	No	No	Yes	No

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

2.1 Spiral Model was developed by

(a) Bev Littlewood (b) Berry Boehm
(c) Roger Pressman (d) Victor Basili

2.2 Which model is most popular for student's small projects?

(a) Waterfall model (b) Spiral model
(c) Quick and fix model (d) Prototyping model

2.3 Which is not a software life cycle model?

(a) Waterfall model (b) Spiral model
(c) Prototyping model (d) Capability maturity model

2.4 Project risk factor is considered in

(a) Waterfall model (b) Prototyping model
(c) Spiral model (d) Iterative enhancement model

2.5 SDLC stands for

(a) Software design life cycle (b) Software development life cycle
(c) System development life cycle (d) System design life cycle

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

2.6 Build and fix model has

(a) 3 phases (b) 1 phase
(c) 2 phases (d) 4 phases

2.7 SRS stands for

(a) Software requirements specification (b) Software requirements solution
(c) System requirements specification (d) none of the above

2.8 Waterfall model is not suitable for

(a) small projects (b) accommodating change
(c) complex projects (d) none of the above

2.9 RAD stands for

(a) Rapid application development (b) Relative application development
(c) Ready application development (d) Repeated application development

2.10 RAD model was proposed by

(a) Lucent Technologies (b) Motorola
(c) IBM (d) Microsoft

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

2.11 If requirements are easily understandable and defined,which model is best suited?

(a) Waterfall model (b) Prototyping model
(c) Spiral model (d) None of the above

2.12 If requirements are frequently changing, which model is to be selected?

(a) Waterfall model (b) Prototyping model
(c) RAD model (d) Iterative enhancement model

2.13 If user participation is available, which model is to be chosen?

(a) Waterfall model (b) Iterative enhancement model
(c) Spiral model (d) RAD model

2.14 If limited user participation is available, which model is to be selected?

(a) Waterfall model (b) Spiral model
(c) Iterative enhancement model (d) any of the above

2.15 If project is the enhancement of existing system, which model is best suited?

(a) Waterfall model (b) Prototyping model
(c) Iterative enhancement model (d) Spiral model

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

2.16 Which one is the most important feature of spiral model?

(a) Quality management (b) Risk management

(c) Performance management (d) Efficiency management

2.17 Most suitable model for new technology that is not well understood is:

(a) Waterfall model (b) RAD model

(c) Iterative enhancement model (d) Evolutionary development model

2.18 Statistically, the maximum percentage of errors belong to the following phase of SDLC

(a) Coding (b) Design

(c) Specifications (d) Installation and maintenance

2.19 Which phase is not available in software life cycle?

(a) Coding (b) Testing

(c) Maintenance (d) Abstraction

2.20 The development is supposed to proceed linearly through the phase in

(a) Spiral model (b) Waterfall model

(c) Prototyping model (d) None of the above

Multiple Choice Questions

Note: Select most appropriate answer of the following questions:

2.21 Unified process is maintained by

(a) Infosys
(b) Rational software corporation
(c) SUN Microsystems
(d) None of the above

2.22 Unified process is

(a) Iterative
(b) Incremental
(c) Evolutionary
(d) All of the above

2.23 Who is not in the team of Unified process development?

(a) I.Jacobson
(b) G.Booch
(c) B.Boehm
(d) J.Rumbaugh

2.24 How many phases are in the unified process?

(a) 4
(b) 5
(c) 2
(d) None of the above

2.25 The outcome of construction phased can be treated as:

(a) Product release
(b) Beta release
(c) Alpha release
(d) All of the above

Exercises

2.1 What do you understand by the term Software Development Life Cycle (SDLC)? Why is it important to adhere to a life cycle model while developing a large software product?

2.2 What is software life cycle? Discuss the generic waterfall model.

2.3 List the advantages of using waterfall model instead of adhoc build and fix model.

2.4 Discuss the prototyping model. What is the effect of designing a prototype on the overall cost of the project?

2.5 What are the advantages of developing the prototype of a system?

2.6 Describe the type of situations where iterative enhancement model might lead to difficulties.

2.7 Compare iterative enhancement model and evolutionary process model.

Exercises

2.8 Sketch a neat diagram of spiral model of software life cycle.

2.9 Compare the waterfall model and the spiral model of software development.

2.10 As we move outward along with process flow path of the spiral model, what can we say about software that is being developed or maintained.

2.11 How does "project risk" factor effect the spiral model of software development.

2.12 List the advantages and disadvantages of involving a software engineer throughout the software development planning process.

2.13 Explain the spiral model of software development. What are the limitations of such a model?

2.14 Describe the rapid application development (RAD) model.Discuss each phase in detail.

2.15 What are the characteristics to be considered for the selection of the life cycle model?

Exercises

2.16 What is the role of user participation in the selection of a life cycle model?.

2.17 Why do we feel that characteristics of requirements play a very significant role in the selection of a life cycle model?

2.18 Write short note on "status of development team" for the selection of a life cycle model?.

2.19 Discuss the selection process parameters for a life cycle model.

2.20 What is unified process? Explain various phases along with the outcome of each phase.

2.21 Describe the unified process work products after each phase of unified process.

2.22 What are the advantages of iterative approach over sequential approach? Why is unified process called as iterative or incremental?

Software Requirements Analysis and specification

Requirement Engineering

Requirements describe

What **not** **How**

Produces one large document written in natural language contains a description of what the system will do without describing how it will do it.

Crucial process steps

Quality of product → Process that creates it

Without well written document

-- Developers do not know what to build

-- Customers do not know what to expect

-- What to validate

Crucial Process Steps of requirement engineering

Requirement Engineering

Requirement Engineering is the disciplined application of proven principles, methods, tools, and notations to describe a proposed system's intended behavior and its associated constraints.

SRS may act as a contract between developer and customer.

State of practice

Requirements are difficult to uncover

- Requirements change
- Over reliance on CASE Tools
- Tight project Schedule
- Communication barriers
- Market driven software development
- Lack of resources

Example

A University wish to develop a software system for the student result management of its M.Tech. Programme. A problem statement is to be prepared for the software development company. The problem statement may give an overview of the existing system and broad expectations from the new software system.

Types of Requirements

Stakeholder: Anyone who should have some direct or indirect influence on the system requirements.

--- User

--- Affected persons

Types of Requirements

Functional requirements describe what the software has to do. They are often called product features.

Non Functional requirements are mostly quality requirements. That stipulate how well the software does, what it has to do.

Availability
Reliability
Usability
Flexibility

} For Users

Maintainability
Portability
Testability

} For Developers

Types of Requirements

User and system requirements

- User requirement are written for the users and include functional and non functional requirement.
- System requirement are derived from user requirement.
- The user system requirements are the parts of software requirement and specification (SRS) document.

Types of Requirements

Interface Specification

- Important for the customers.

TYPES OF INTERFACES

- Procedural interfaces (also called Application Programming Interfaces (APIs)).
- Data structures
- Representation of data.

Feasibility Study

Is cancellation of a project a bad news?

As per IBM report, "31% projects get cancelled before they are completed, 53% over-run their cost estimates by an average of 189% & for every 100 projects, there are 94 restarts.

How do we cancel a project with the least work?

➡ CONDUCT A FEASIBILTY STUDY

Feasibility Study

Technical feasibility

- Is it technically feasible to provide direct communication connectivity through space from one location of globe to another location?

- Is it technically feasible to design a programming language using “Sanskrit”?

Feasibility Study

Feasibility depends upon non technical Issues like:

- Are the project's cost and schedule assumption realistic?
- Does the business model realistic?
- Is there any market for the product?

Feasibility Study

Purpose of feasibility study

"evaluation or analysis of the potential impact of a proposed project or program."

Focus of feasibility studies

- Is the product concept viable?
- Will it be possible to develop a product that matches the project's vision statement?
- What are the current estimated cost and schedule for the project?

Feasibility Study

Focus of feasibility studies

- How big is the gap between the original cost & schedule targets & current estimates?
- Is the business model for software justified when the current cost & schedule estimate are considered?
- Have the major risks to the project been identified & can they be surmounted?
- Is the specifications complete & stable enough to support remaining development work?

Feasibility Study

Focus of feasibility studies

- Have users & developers been able to agree on a detailed user interface prototype? If not, are the requirements really stable?
- Is the software development plan complete & adequate to support further development work?

Requirements Elicitation

Perhaps

- Most difficult
- Most critical
- Most error prone
- Most communication intensive

Succeed

→ effective customer developer partnership

Selection of any method

1. It is the only method that we know
2. It is our favorite method for all situations
3. We understand intuitively that the method is effective in the present circumstances.

Normally we rely on first two reasons.

Requirements Elicitation

1. Interviews

Both parties have a common goal

--- open ended
--- structured

} Interview

Success of the project

Selection of stakeholder

1. Entry level personnel
2. Middle level stakeholder
3. Managers
4. Users of the software (Most important)

Requirements Elicitation

Types of questions.

- Any problems with existing system
- Any Calculation errors
- Possible reasons for malfunctioning
- No. of Student Enrolled

Requirements Elicitation

5. Possible benefits
6. Satisfied with current policies
7. How are you maintaining the records of previous students?
8. Any requirement of data from other system
9. Any specific problems
10. Any additional functionality
11. Most important goal of the proposed development

At the end, we may have wide variety of expectations from the proposed software.

Requirements Elicitation

2. Brainstorming Sessions

It is a group technique

↓

group discussions

New ideas Quickly

Creative Thinking

Prepare long list of requirements

Categorized
Prioritized
Pruned

*Idea is to generate views ,not to vet them.

Groups

1. Users 2. Middle Level managers 3. Total Stakeholders

Requirements Elicitation

A Facilitator may handle group bias, conflicts carefully.

-- Facilitator may follow a published agenda

-- Every idea will be documented in a way that everyone can see it.

--A detailed report is prepared.

3. Facilitated Application specification Techniques (FAST)

-- Similar to brainstorming sessions.

-- Team oriented approach

-- Creation of joint team of customers and developers.

Requirements Elicitation

Guidelines

1. Arrange a meeting at a neutral site.
2. Establish rules for participation.
3. Informal agenda to encourage free flow of ideas.
4. Appoint a facilitator.
5. Prepare definition mechanism board, worksheets, wall stickier.
6. Participants should not criticize or debate.

FAST session Preparations

Each attendee is asked to make a list of objects that are:

Requirements Elicitation

1. Part of environment that surrounds the system.
2. Produced by the system.
3. Used by the system.

A. List of constraints
B. Functions
C. Performance criteria

Activities of FAST session

1. Every participant presents his/her list
2. Combine list for each topic
3. Discussion
4. Consensus list
5. Sub teams for mini specifications
6. Presentations of mini-specifications
7. Validation criteria
8. A sub team to draft specifications

Requirements Elicitation

4. Quality Function Deployment

-- Incorporate voice of the customer

Technical requirements.

Documented

Prime concern is customer satisfaction

What is important for customer?

-- Normal requirements

-- Expected requirements

-- Exciting requirements

Requirements Elicitation

Steps

1. Identify stakeholders
2. List out requirements
3. Degree of importance to each requirement.

Requirements Elicitation

5 Points : V. Important

4 Points : Important

3 Points : Not Important but nice to have

2 Points : Not important

1 Points : Unrealistic, required further exploration

Requirement Engineer may categorize like:

(i) It is possible to achieve

(ii) It should be deferred & Why

(iii) It is impossible and should be dropped from consideration

First Category requirements will be implemented as per priority assigned with every requirement.

5. The Use Case Approach

Ivar Jacobson & others introduced Use Case approach for elicitation & modeling.

Use Case – give functional view

The terms

Use Case
Use Case Scenario
Use Case Diagram

} Often Interchanged

But they are different

Use Cases are structured outline or template for the description of user requirements modeled in a structured language like English.

Requirements Elicitation

Use case Scenarios are unstructured descriptions of user requirements.

Use case diagrams are graphical representations that

may be decomposed into further levels of abstraction.

Components of Use Case approach

Actor:

An actor or external agent, lies outside the system model, but interacts with it in some way.

Actor → Person, machine, information System

Requirements Elicitation

- Cockburn distinguishes between Primary and secondary actors.
- A Primary actor is one having a goal requiring the assistance of the system.
- A Secondary actor is one from which System needs assistance.

Use Cases

A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied.

Requirements Elicitation

* It describes the sequence of interactions between actors and the system necessary to deliver the services that satisfies the goal.

* Alternate sequence

* System is treated as black box.

Thus

Use Case captures who (actor) does what (interaction) with the system, for what purpose (goal), without dealing with system internals.

Requirements Elicitation

*defines all behavior required of the system, bounding the scope of the system.

Jacobson & others proposed a template for writing Use cases as shown below:

1. Introduction

Describe a quick background of the use case.

2.Actors

List the actors that interact and participate in the use cases.

3.Pre Conditions

Pre conditions that need to be satisfied for the use case to perform.

4. Post Conditions

Define the different states in which we expect the system to be in, after the use case executes.

5. Flow of events

5.1 Basic Flow

List the primary events that will occur when this use case is executed.

5.2 Alternative Flows

Any Subsidiary events that can occur in the use case should be separately listed. List each such event as an alternative flow. A use case can have many alternative flows as required.

6.Special Requirements

Business rules should be listed for basic & information flows as special requirements in the use case narration .These rules will also be used for writing test cases. Both success and failures scenarios should be described.

7.Use Case relationships

For Complex systems it is recommended to document the relationships between use cases. Listing the relationships between use cases also provides a mechanism for traceability

Use Case Template.

Use Case Guidelines

1. Identify all users
2. Create a user profile for each category of users including all roles of the users play that are relevant to the system.
3. Create a use case for each goal, following the use case template maintain the same level of abstraction throughout the use case. Steps in higher level use cases may be treated as goals for lower level (i.e. more detailed), sub-use cases.
4. Structure the use case
5. Review and validate with users.

Use case Diagrams

-- represents what happens when actor interacts with a system.

-- captures functional aspect of the system.

Actor

Use Case

Relationship between actors and use case and/or between the use cases.

-- Actors appear outside the rectangle.

--Use cases within rectangle providing functionality.

--Relationship association is a solid line between actor & use cases.

Requirements Elicitation

*Use cases should not be used to capture all the details of the system.

*Only significant aspects of the required functionality

*No design issues

*Use Cases are for “what” the system is , not “how” the system will be designed

* Free of design characteristics

Use case diagram for Result Management System

Requirements Elicitation

1. Maintain student Details

   Add/Modify/update students details like name, address.

2. Maintain subject Details

   Add/Modify/Update Subject information semester wise

3. Maintain Result Details

   Include entry of marks and assignment of credit points for each paper.

4. Login

   Use to Provide way to enter through user id & password.

5. Generate Result Report

   Use to print various reports

6. View Result
   (i) According to course code
   (ii) According to Enrollment number/roll number

Requirements Elicitation (Use Case)

Login

1.1 Introduction : This use case describes how a user logs into the Result Management System.

1.2 Actors : (i) Data Entry Operator
(ii) Administrator/Deputy Registrar

1.3 Pre Conditions : None

1.4 Post Conditions : If the use case is successful, the actor is logged into the system. If not, the system state is unchanged.

Requirements Elicitation (Use Case)

1.5 Basic Flow : This use case starts when the actor wishes to login to the Result Management system.

(i) System requests that the actor enter his/her name and password.

(ii) The actor enters his/her name & password.

(iii) System validates name & password, and if finds correct allow the actor to logs into the system.

1.6 Alternate Flows

1.6.1 Invalid name & password

If in the basic flow, the actor enters an invalid name and/or password, the system displays an error message. The actor can choose to either return to the beginning of the basic flow or cancel the login, at that point, the use case ends.

1.7 Special Requirements:

None

1.8 Use case Relationships:

None

2.Maintain student details

2.1 Introduction : Allow DEO to maintain student details. This includes adding, changing and deleting student information

2.2 Actors : DEO

2.3 Pre-Conditions: DEO must be logged onto the system before this use case begins.

Use Cases

2.4 Post-conditions : If use case is successful, student information is added/updated/deleted from the system. Otherwise, the system state is unchanged.

2.5 Basic Flow : Starts when DEO wishes to add/modify/update/delete Student information.

(i) The system requests the DEO to specify the function, he/she would like to perform (Add/update/delete)

(ii) One of the sub flow will execute after getting the information.

Use Cases

- If DEO selects "Add a student", "Add a student" sub flow will be executed.
- If DEO selects "update a student", "update a student" sub flow will be executed.
- If DEO selects "delete a student", "delete a student" sub flow will be executed.

2.5.1 Add a student

(i) The system requests the DEO to enter:

Name
Address
Roll No
Phone No
Date of admission

(ii) System generates unique id

2.5.2 Update a student

(i) System requires the DEO to enter student-id.

(ii) DEO enters the student_id. The system retrieves and display the student information.

(iii) DEO makes the desired changes to the student information.

(iv) After changes, the system updates the student record with changed information.

2.5.3 Delete a student

(i) The system requests the DEO to specify the student-id.

(ii) DEO enters the student-id. The system retrieves and displays the student information.

(iii) The system prompts the DEO to confirm the deletion of the student.

(iv) The DEO confirms the deletion.

(v) The system marks the student record for deletion.

2.6 Alternative flows

2.6.1 Student not found

If in the update a student or delete a student sub flows, a student with specified_id does not exist, the system displays an error message .The DEO may enter a different id or cancel the operation. At this point ,Use case ends.

2.6.2 Update Cancelled

If in the update a student sub-flow, the data entry operator decides not to update the student information, the update is cancelled and the basic flow is restarted at the begin.

2.6.3 Delete cancelled

If in the delete a student sub flows, DEO decides not to delete student record ,the delete is cancelled and the basic flow is re-started at the beginning.

2.7 Special requirements

None

2.8 Use case relationships

None

3. Maintain Subject Details

3.1 Introduction

The DEO to maintain subject information. This includes adding, changing, deleting subject information from the system

3.2 Actors : DEO

3.3 Preconditions : DEO must be logged onto the system before the use case begins.

Use Cases

3.4 Post conditions :

If the use case is successful, the subject information is added, updated, or deleted from the system, otherwise the system state is unchanged.

3.5 Basic flows :

The use case starts when DEO wishes to add, change, and/or delete subject information from the system.

(i) The system requests DEO to specify the function he/she would like to perform i.e.

- Add a subject
- Update a subject
- Delete a subject.

Use Cases

(ii) Once the DEO provides the required information, one of the sub flows is executed.

- If DEO selected "Add a subject" the "Add-a subject sub flow is executed.
- If DEO selected "Update-a subject" the "update-a- subject" sub flow is executed
- If DEO selected "Delete- a- subject", the "Delete-a-subject" sub flow is executed.

3.5.1 Add a Subject

(i) The System requests the DEO to enter the subject information. This includes :

* Name of the subject

Use Cases

* Subject Code
* Semester
* Credit points

(ii) Once DEO provides the requested information, the system generates and assigns a unique subject-id to the subject. The subject is added to the system.

(iii) The system provides the DEO with new subject-id.

3.5.2 Update a Subject

(i) The system requests the DEO to enter subject_id.

(ii) DEO enters the subject_id. The system retrieves and displays the subject information.

(iii) DEO makes the changes.

(iv) Record is updated.

3.5.3 Delete a Subject

(i) Entry of subject_id.

(ii) After this, system retrieves & displays subject information.

* System prompts the DEO to confirm the deletion.

* DEO verifies the deletion.

* The system marks the subject record for deletion.

3.6 Alternative Flow

3.6.1 Subject not found

If in any sub flows, subject-id not found, error message is displayed. The DEO may enter a different id or cancel the case ends here.

3.6.2 Update Cancelled

If in the update a subject sub-flow, the data entry operator decides not to update the subject information, the update is cancelled and the basic flow is restarted at the begin.

3.6.3 Delete Cancellation

If in delete-a-subject sub flow, the DEO decides not to delete subject, the delete is cancelled, and the basic flow is restarted from the beginning.

3.7 Special Requirements:

None

3.8 Use Case-relationships

None

4. Maintain Result Details

4.1 Introduction

This use case allows the DEO to maintain subject & marks information of each student. This includes adding and/or deleting subject and marks information from the system.

4.2 Actor

DEO

4.3 Pre Conditions

DEO must be logged onto the system.

4.4 Post Conditions

If use case is successful ,marks information is added or deleted from the system. Otherwise, the system state is unchanged.

4.5 Basic Flow

This use case starts, when the DEO wishes to add, update and/or delete marks from the system.

(i) DEO to specify the function

(ii) Once DEO provides the information one of the subflow is executed.

* If DEO selected "Add Marks ", the Add marks subflow is executed.
* If DEO selected "Update Marks", the update marks subflow is executed.
* If DEO selected "Delete Marks", the delete marks subflow is executed.

4.5.1 Add Marks Records

Add marks information .This includes:

a. Selecting a subject code.

b. Selecting the student enrollment number.

c. Entering internal/external marks for that subject code & enrollment number.

Use Cases

(ii) If DEO tries to enter marks for the same combination of subject and enrollment number,the system gives a message that the marks have already been entered.

(iii) Each record is assigned a unique result_id.

4.5.2 Delete Marks records

1. DEO makes the following entries:
   a. Selecting subject for which marks have to be deleted.
   b. Selecting student enrollment number.
   c. Displays the record with id number.
   d. Verify the deletion.
   e. Delete the record.

4.5.2 Update Marks records

1. The System requests DEO to enter the record_id.

2. DEO enters record_id. The system retrieves & displays the information.

3. DEO makes changes.

4. Record is updated.

4.5.3 Compute Result

(i) Once the marks are entered, result is computed for each student.

(ii) If a student has scored more than 50% in a subject, the associated credit points are allotted to that student.

(iii) The result is displayed with subject-code, marks & credit points.

4.6 Alternative Flow

4.6.1 Record not found

If in update or delete marks sub flows, marks with specified id number do not exist, the system displays an error message. DEO can enter another id or cancel the operation.

4.6.2 Delete Cancelled

If in Delete Marks, DEO decides not to delete marks, the delete is cancelled and basic flow is re-started at the beginning.

4.7 Special Requirements

None

4.8 Use case relationships

None

5 View/Display result

5.1 Introduction

This use case allows the student/Teacher or anyone to view the result. The result can be viewed on the basis of course code and/or enrollment number.

5.2 Actors

Administrator/DR, Teacher/Student

5.3 Pre Conditions

None

5.4 Post Conditions

If use case is successful, the marks information is displayed by the system. Otherwise, state is unchanged.

5.5 Basic Flow

Use case begins when student, teacher or any other person wish to view the result.

Two ways

-- Enrollment no.

-- Course code

Use Cases

(ii) After selection, one of the sub flow is executed.

Course code → Sub flow is executed

Enrollment no. → Sub flow is executed

5.5.1 View result enrollment number wise

(i) User to enter enrollment number

(ii) System retrieves the marks of all subjects with credit points

(iii) Result is displayed.

Use Cases

5.6 Alternative Flow

5.6.1 Record not found

Error message should be displayed.

5.7 Special Requirements

None

5.8 Use Case relationships

None

6. Generate Report

6.1 Introduction

This use case allows the DR to generate result reports. Options are

a. Course code wise
b. Semester wise
c. Enrollment Number wise

6.2 Actors

DR

6.3 Pre-Conditions

DR must logged on to the system

Use Cases

6.4 Post conditions

If use case is successful, desired report is generated. Otherwise, the system state is unchanged.

6.5 Basic Flow

The use case starts, when DR wish to generate reports.

(i) DR selects option.

(ii) System retrieves the information displays.

(iii) DR takes printed reports.

6.6 Alternative Flows

6.6.1 Record not found

If not found, system generates appropriate message. The DR can select another option or cancel the operation. At this point, the use case ends.

6.7 Special Requirements

None

6.8 Use case relationships

None

7. Maintain User Accounts

7.1 Introduction

This use case allows the administrator to maintain user account. This includes adding, changing and deleting user account information from the system.

7.2 Actors

Administrator

7.3 Pre-Conditions

The administrator must be logged on to the system before the use case begins.

Use Cases

7.4 Post-Conditions

If the use case was successful, the user account information is added, updated, or deleted from the system. Otherwise, the system state is unchanged.

7.5 Basic Flow

This use case starts when the Administrator wishes to add, change, and/or delete use account information from the system.

(i) The system requests that the Administrator specify the function he/she would like to perform (either Add a User Account, Update a User Account, or Delete a User Account).

(ii) Once the Administrator provides the requested information, one of the sub-flows is executed

Use Cases

* If the Administrator selected "Add a User Account", the **Add a User Account** sub flow is executed.

* If the Administrator selected "Update a User Account", the **Update a User Account** sub-flow is executed.

* If the Administrator selected "Delete a User Account", the **Delete a User Account** sub-flow is executed.22

7.5.1 Add a User Account

1. The system requests that the Administrator enters the user information. This includes:

(*a*) User Name

(*b*) User ID-should be unique for each user account

(*c*) Password

(*d*) Role

Use Cases

2. Once the Administrator provides the requested information, the user account information is added.

7.5.2 Update a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The Administrator makes the desired changes to the user account information. This includes any of the information specified in the **Add a User Account** sub-flow.
4. Once the Administrator updates the necessary information, the system updates the user account record with the updated information.

7.5.3 Delete a User Account

1. The system requests that the Administrator enters the User ID.
2. The Administrator enters the User ID. The system retrieves and displays the user account information.
3. The system prompts the Administrator to confirm the deletion of the user account.
4. The Administrator confirms the deletion.
5. The system deletes the user account record.

7.6 Alternative Flows

7.6.1 User Not Found

If in the **Update a User Account** or **Delete a User Account** sub-flows, a user account with the specified User ID does not exist, the system displays an error message. The Administrator can then enter a different User ID or cancel the operation, at which point the use case ends.

7.6.2 Update Cancelled

If in the **Update a User Account** sub-flow, the Administrator decides not to update the user account information, the update is cancelled and the **Basic Flow** is re-started at the beginning.

7.6.3 Delete Cancelled

If in the **Delete a User Account** sub-flow, the Administrator decides not to delete the user account information, the delete is cancelled and the **Basic Flow** is re-started at the beginning.

7.7 Special Requirements

None

7.8 Use case relationships

None

8. Reset System

8.1 Introduction

This use case allows the allows the administrator to reset the system by deleting all existing information from the system .

8.2 Actors

Administrator

8.3 Pre-Conditions

The administrator must be logged on to the system before the use case begins.

8.4 Post-Conditions

If the use case was successful, all the existing information is deleted from the backend database of the system. Otherwise, the system state is unchanged.

8.5 Basic Flow

This use case starts when the Administrator wishes to reset the system.

i. The system requests the Administrator to confirm if he/she wants to delete all the existing information from the system.

ii. Once the Administrator provides confirmation, the system deletes all the existing information from the backend database and displays an appropriate message.

Use Cases

8.6 Alternative Flows

8.6.1 Reset Cancelled

If in the Basic Flow, the Administrator decides not to delete the entire existing information, the reset is cancelled and the use case ends.

8.7 Special Requirements

None

8.8 Use case relationships

None

Requirements Analysis

We analyze, refine and scrutinize requirements to make consistent & unambiguous requirements.

Steps

Requirements Analysis Steps

Requirements Analysis

Data Flow Diagrams

DFD show the flow of data through the system.

--All names should be unique

-- It is not a flow chart

-- Suppress logical decisions

-- Defer error conditions & handling until the end of the analysis

Symbol	Name	Function
	Data Flow	Connect process
	Process	Perform some transformation of its input data to yield output data.

Requirements Analysis

Symbol	Name	Function
	Source or sink	A source of system inputs or sink of system outputs
	Data Store	A repository of data, the arrowhead indicate net input and net outputs to store

Leveling

DFD represent a system or software at any level of abstraction.

A level 0 DFD is called fundamental system model or context model represents entire software element as a single bubble with input and output data indicating by incoming & outgoing arrows.

Requirements Analysis

Data Dictionaries

DFD → DD

Data Dictionaries are simply repositories to store information about all data items defined in DFD.

Includes :

- Name of data item
- Aliases (other names for items)
- Description/Purpose
- Related data items
- Range of values
- Data flows
- Data structure definition

Data Dictionaries

Notation	Meaning
x= a+b	x consists of data element a & b
x={a/b}	x consists of either a or b
x=(a)	x consists of an optional data element a
x= y{a}	x consists of y or more occurrences
x={a}z	x consists of z or fewer occurrences of a
x=y{a}z	x consists of between y & z occurrences of a{

Entity-Relationship Diagrams

Entity-Relationship Diagrams

It is a detailed logical representation of data for an organization and uses three main constructs.

Entities

Fundamental thing about which data may be maintained. Each entity has its own identity.

Entity Type is the description of all entities to which a common definition and common relationships and attributes apply.

Entity-Relationship Diagrams

Consider an insurance company that offers both home and automobile insurance policies .These policies are offered to individuals and businesses.

Entity-Relationship Diagrams

Relationships

A relationship is a reason for associating two entity types.

Binary relationships ⟶ involve two entity types

A CUSTOMER is insured by a POLICY. A POLICY CLAIM is made against a POLICY.

Relationships are represented by diamond notation in a ER diagram.

Relationships added to ERD

Entity-Relationship Diagrams

A training department is interested in tracking which training courses each of its employee has completed.

Each employee may complete more than one course,and each course may be completed by more than one employee.

Entity-Relationship Diagrams

Degree of relationship

It is the number of entity types that participates in that relationship.

Unary relationship

One to One

One to many

Entity-Relationship Diagrams

Binary Relationship

One to One

One to many

Many to many

Entity-Relationship Diagrams

Ternary relationship

Cardinalities and optionality

Two entity types A,B, connected by a relationship.
The cardinality of a relationship is the number of instances of entity B that can be associated with each instance of entity A

Entity-Relationship Diagrams

Minimum cardinality is the minimum number of instances of entity B that may be associated with each instance of entity A.

Minimum no. of tapes available for a movie is zero.We say VIDEO TAPE is an optional participant in the is-stocked-as relationship.

Entity-Relationship Diagrams

Attributes

Each entity type has a set of attributes associated with it.

An attribute is a property or characteristic of an entity that is of interest to organization.

Attribute

Entity-Relationship Diagrams

A candidate key is an attribute or combination of attributes that uniquely identifies each instance of an entity type.

Student_ID ⟶ Candidate Key

If there are more candidate keys, one of the key may be chosen as the Identifier.
It is used as unique characteristic for an entity type.

Identifier

Entity-Relationship Diagrams

Vendors quote prices for several parts along with quantity of parts. Draw an E-R diagram.

Approaches to problem analysis

1. List all inputs, outputs and functions.
2. List all functions and then list all inputs and outputs associated with each function.

Structured requirements definition (SRD)

Step1

Define a user level DFD. Record the inputs and outputs for each individual in a DFD.

Step2

Define a combined user level DFD.

Step3

Define application level DFD.

Step4

Define application level functions.

Requirements Documentation

This is the way of representing requirements in a consistent format

SRS serves many purpose depending upon who is writing it.

-- written by customer
-- written by developer

Serves as contract between customer & developer.

Nature of SRS

Basic Issues

- Functionality
- External Interfaces
- Performance
- Attributes
- Design constraints imposed on an Implementation

Requirements Documentation

SRS Should

-- Correctly define all requirements

-- not describe any design details

-- not impose any additional constraints

Characteristics of a good SRS

An SRS Should be

✓ Correct

✓ Unambiguous

✓ Complete

✓ Consistent

Requirements Documentation

- ✓ Ranked for important and/or stability
- ✓ Verifiable
- ✓ Modifiable
- ✓ Traceable

Requirements Documentation

Correct

An SRS is correct if and only if every requirement stated therein is one that the software shall meet.

Unambiguous

An SRS is unambiguous if and only if, every requirement stated therein has only one interpretation.

Complete

An SRS is complete if and only if, it includes the following elements

(i) All significant requirements, whether related to functionality, performance, design constraints, attributes or external interfaces.

Requirements Documentation

(ii) Responses to both valid & invalid inputs.

(iii) Full Label and references to all figures, tables and diagrams in the SRS and definition of all terms and units of measure.

Consistent

An SRS is consistent if and only if, no subset of individual requirements described in it conflict.

Ranked for importance and/or Stability

If an identifier is attached to every requirement to indicate either the importance or stability of that particular requirement.

Requirements Documentation

Verifiable

An SRS is verifiable, if and only if, every requirement stated therein is verifiable.

Modifiable

An SRS is modifiable, if and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining structure and style.

Traceable

An SRS is traceable, if the origin of each of the requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.

Organization of the SRS

IEEE has published guidelines and standards to organize an SRS.
First two sections are same. The specific tailoring occurs in section-3.

1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Definition, Acronyms and abbreviations
    - 1.4 References
    - 1.5 Overview

Requirements Documentation

2. The Overall Description

2.1 Product Perspective

- 2.1.1 System Interfaces
- 2.1.2 Interfaces
- 2.1.3 Hardware Interfaces
- 2.1.4 Software Interfaces
- 2.1.5 Communication Interfaces
- 2.1.6 Memory Constraints
- 2.1.7 Operations
- 2.1.8 Site Adaptation Requirements

Requirements Documentation

2.2 Product Functions
2.3 User Characteristics
2.4 Constraints
2.5 Assumptions for dependencies
2.6 Apportioning of requirements

3. Specific Requirements

3.1 External Interfaces
3.2 Functions
3.3 Performance requirements
3.4 Logical database requirements
3.5 Design Constraints
3.6 Software System attributes
3.7 Organization of specific requirements
3.8 Additional Comments.

Requirements Validation

Check the document for:

- ✓ Completeness & consistency
- ✓ Conformance to standards
- ✓ Requirements conflicts
- ✓ Technical errors
- ✓ Ambiguous requirements

Requirements Validation

Requirements Review Process

Requirements Validation

Problem actions

- Requirements clarification
- Missing information
  - find this information from stakeholders
- Requirements conflicts
  - Stakeholders must negotiate to resolve this conflict
- Unrealistic requirements
  - Stakeholders must be consulted
- Security issues
  - Review the system in accordance to security standards

Review Checklists

- ✓ Redundancy
- ✓ Completeness
- ✓ Ambiguity
- ✓ Consistency
- ✓ Organization
- ✓ Conformance
- ✓ Traceability

Prototyping

Validation prototype should be reasonably complete & efficient & should be used as the required system.

Requirements Management

- Process of understanding and controlling changes to system requirements.

ENDURING & VOLATILE REQUIREMENTS

o Enduring requirements: They are core requirements & are related to main activity of the organization.

  Example: issue/return of a book, cataloging etc.

o Volatile requirements: likely to change during software development lifer cycle or after delivery of the product

Requirements Management Planning

- Very critical.
- Important for the success of any project.

Requirements Change Management

- Allocating adequate resources
- Analysis of requirements
- Documenting requirements
- Requirements traceability
- Establishing team communication
- Establishment of baseline

Download from

Q:\IRM\PRIVATE\INITIATIATI\QA\QAPLAN\SRSPLAN.DOC

Multiple Choice Questions

Note. Choose the most appropriate answer of the following questions.

3.1 Which one is not a step of requirement engineering?

(a) Requirements elicitation
(b) Requirements analysis
(c) Requirements design
(d) Requirements documentation

3.2 Requirements elicitation means

(a) Gathering of requirements
(b) Capturing of requirements
(c) Understanding of requirements
(d) All of the above

Multiple Choice Questions

3.3 SRS stands for

(a) Software requirements specification
(b) System requirements specification
(c) Systematic requirements specifications
(d) None of the above

3.4 SRS document is for

(a) "What" of a system?
(b) How to design the system?
(c) Costing and scheduling of a system
(d) System's requirement.

Multiple Choice Questions

3.5 Requirements review process is carried out to

(a) Spend time in requirements gathering
(b) Improve the quality of SRS
(c) Document the requirements
(d) None of the above

3.6 Which one of the statements is not correct during requirements engineering?

(a) Requirements are difficult to uncover
(b) Requirements are subject to change
(c) Requirements should be consistent
(d) Requirements are always precisely known.

Multiple Choice Questions

3.7 Which one is not a type of requirements?

(a) Known requirements
(b) Unknown requirements
(c) Undreamt requirements
(d) Complex requirements

3.8 Which one is not a requirements elicitation technique?

(a) Interviews
(b) The use case approach
(c) FAST
(d) Data flow diagram.

Multiple Choice Questions

3.9 FAST stands for

(a) Functional Application Specification Technique
(b) Fast Application Specification Technique
(c) Facilitated Application Specification Technique
(d) None of the above

3.10 QFD in requirement engineering stands for

(a) Quality function design
(b) Quality factor design
(c) Quality function development
(d) Quality function deployment

Multiple Choice Questions

3.11 Which is not a type of requirements under quality function deployment

(a) Normal requirements
(b) Abnormal requirements
(c) Expected requirements
(d) Exciting requirements

3.12 Use case approach was developed by

(a) I. Jacobson and others
(b) J.D. Musa and others
(c) B. Littlewood
(d) None of the above

Multiple Choice Questions

3.13 Context diagram explains

(a) The overview of the system
(b) The internal view of the system
(c) The entities of the system
(d) None of the above

3.14 DFD stands for

(a) Data Flow design
(b) Descriptive functional design
(c) Data flow diagram
(d) None of the above

Multiple Choice Questions

3.15 Level-O DFD is similar to

(a) Use case diagram
(b) Context diagram
(c) System diagram
(d) None of the above

3.16 ERD stands for

(a) Entity relationship diagram
(b) Exit related diagram
(c) Entity relationship design
(d) Exit related design

Multiple Choice Questions

3.17 Which is not a characteristic of a good SRS?

(a) Correct
(b) Complete
(c) Consistent
(d) Brief

3.18 Outcome of requirements specification phase is

(a) Design Document
(b) Software requirements specification
(c) Test Document
(d) None of the above

Multiple Choice Questions

3.19 The basic concepts of ER model are:

(a) Entity and relationship
(b) Relationships and keys
(c) Entity, effects and relationship
(d) Entity, relationship and attribute

3.20 The DFD depicts

(a) Flow of data
(b) Flow of control
(c) Both (a) and (b)
(d) None of the above

Multiple Choice Questions

3.21 Product features are related to:

(a) Functional requirements
(b) Non functional requirements
(c) Interface requirement
(d) None of the above

3.22 Which one is a quality attribute?

(a) Reliability
(b) Availability
(c) Security
(d) All of the above

Multiple Choice Questions

3.23 IEEE standard for SRS is:

(a) IEEE Standard 837-1998
(b) IEEE Standard 830-1998
(c) IEEE Standard 832-1998
(d) IEEE Standard 839-1998

3.24 Which one is not a functional requirement?

(a) Efficiency
(b) Reliability
(c) Product features
(d) Stability

Multiple Choice Questions

3.23 APIs stand for:

(a) Application performance interfaces
(b) Application programming interfaces
(c) Application programming integration
(d) Application performance integration

Exercises

3.1 Discuss the significance and use of requirement engineering. What are the problems in the formulation of requirements?

3.2 Requirements analysis is unquestionably the most communication intensive step in the software engineering process. Why does the communication path frequently break down ?

3.3 What are crucial process steps of requirement engineering ? Discuss with the help of a diagram.

3.4 Discuss the present state of practices in requirement engineering. Suggest few steps to improve the present state of practice.

3.5 Explain the importance of requirements. How many types of requirements are possible and why ?

3.6 Describe the various steps of requirements engineering. Is it essential to follow these steps ?

3.7 What do you understand with the term "requirements elicitation" ? Discuss any two techniques in detail.

3.8 List out requirements elicitation techniques. Which one is most popular and why ?

Exercises

3.9 Describe facilitated application specification technique (FAST) and compare this with brainstorming sessions.

3.10 Discuss quality function deployment technique of requirements elicitation. Why an importance or value factor is associated with every requirement ?

3.11. Explain the use case approach of requirements elicitation. What are use-case guidelines ?

3.12. What are components of a use case diagram. Explain their usage with the help of an example.

3.13. Consider the problem of library management system and design the following:

(*i*) Problem statement

(*ii*) Use case diagram

(*iii*) Use cases.

Exercises

3.14. Consider the problem of railway reservation system and design the following:

(*i*) Problem statement

(*ii*) Use case diagram

(*iii*) Use cases.

3.15. Explain why a many to many relationship is to be modeled as an associative entity ?

3.16. What are the linkages between data flow and E–R diagrams ?

3.17. What is the degree of a relationship ? Give an example of each of the relationship degree.

3.18. Explain the relationship between minimum cardinality and optional and mandatory participation.

3.19. An airline reservation is an association between a passenger, a flight, and a seat. Select a few

pertinent attributes for each of these entity types and represent a reservation in an E–R diagram.

Exercises

3.20. A department of computer science has usual resources and usual users for these resources. A software is to be developed so that resources are assigned without conflict. Draw a DFD specifying the above system.

3.21. Draw a DFD for result preparation automation system of B. Tech. courses (or MCA program) of any university. Clearly describe the working of the system. Also mention all assumptions made by you.

3.22. Write short notes on

(*i*) Data flow diagram

(*ii*) Data dictionary.

3.23. Draw a DFD for borrowing a book in a library which is explained below: "A borrower can borrow a book if it is available else he/she can reserve for the book if he/she so wishes. He/she can borrow a maximum of three books".

3.24. Draw the E–R diagram for a hotel reception desk management.

Explain why, for large software systems development, is it recommended that prototypes should be "throw-away" prototype ?

Exercises

3.26. Discuss the significance of using prototyping for reusable components and explain the problems,which may arise in this situation.

3.27. Suppose a user is satisfied with the performance of a prototype. If he/she is interested to buy this

for actual work, what should be the response of a developer ?

3.28. Comment on the statement: "The term throw-away prototype is inappropriate in that these prototypes expand and enhance the knowledge base that is retained and incorporated in the final prototype; therefore they are not disposed of or thrown away at all."

3.29. Which of the following statements are ambiguous ? Explain why.

(*a*) The system shall exhibit good response time.

(*b*) The system shall be menu driven.

(*c*) There shall exist twenty-five buttons on the control panel, numbered PF1 to PF25.

(*d*) The software size shall not exceed 128K of RAM.

Exercises

3.30. Are there other characteristics of an SRS (besides listed in section 3.4.2) that are desirable ? List a few and describe why ?

3.31. What is software requirements specification (SRS) ? List out the advantages of SRS standards.

Why is SRS known as the black box specification of a system ?

3.32. State the model of a data dictionary and its contents. What are its advantages ?

3.33. List five desirable characteristics of a good SRS document. Discuss the relative advantages of formal requirement specifications. List the important issues, which an SRS must address.

3.34. Construct an example of an inconsistent (incomplete) SRS.

3.35. Discuss the organization of a SRS. List out some important issues of this organization.

Exercises

3.36. Discuss the difference between the following:

(a) Functional & nonfunctional requirements

(b) User & system requirements

Software Project Planning

Software Project Planning

After the finalization of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS.

Software Project Planning

In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software Project Planning

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.

Fig. 1: Activities during Software Project Planning

Size Estimation

Lines of Code (LOC)

If LOC is simply a count of the number of lines then figure shown below contains 18 LOC .

When comments and blank lines are ignored, the program in figure 2 shown below contains 17 LOC.

Fig. 2: Function for sorting an array

```
1.  int. sort (int x[ ], int n)
2.  {
3.  int i, j, save, im1;
4.  /*This function sorts array x in ascending order */
5.  If (n<2) return 1;
6.  for (i=2; i<=n; i++)
7.  {
8.  im1=i-1;
9.  for (j=1; j<=im; j++)
10. if (x[i] < x[j])
11. {
12. Save = x[i];
13. x[i] = x[j];
14. x[j] = save;
15. }
16. }
17. return 0;
18. }
```

Growth of Lines of Code (LOC)

Furthermore, if the main interest is the size of the program for specific functionality, it may be reasonable to include executable statements. The only executable statements in figure shown above are in lines 5-17 leading to a count of 13. The differences in the counts are 18 to 17 to 13. One can easily see the potential for major discrepancies for large programs with many comments or programs written in language that allow a large number of descriptive but non-executable statement. Conte has defined lines of code as:

Software Project Planning

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable statements".

This is the predominant definition for lines of code used by researchers. By this definition, figure shown above has 17 LOC.

Function Count

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

Software Project Planning

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

- Inputs : information entering the system
- Outputs : information leaving the system
- Enquiries : requests for instant access to information
- Internal logical files : information held within the system
- External interface files : information held by other system that is used by the system being analyzed.

The FPA functional units are shown in figure given below:

Fig. 3: FPAs functional units System

Software Project Planning

The five functional units are divided in two categories:

(i) Data function types

- Internal Logical Files (ILF): A user identifiable group of logical related data or control information maintained within the system.
- External Interface files (EIF): A user identifiable group of logically related data or control information referenced by the system, but maintained within another system. This means that EIF counted for one system, may be an ILF in another system.

(ii) Transactional function types

- External Input (EI): An EI processes data or control information that comes from outside the system. The EI is an elementary process, which is the smallest unit of activity that is meaningful to the end user in the business.
- External Output (EO): An EO is an elementary process that generate data or control information to be sent outside the system.
- External Inquiry (EQ): An EQ is an elementary process that is made up to an input-output combination that results in data retrieval.

Software Project Planning

Special features

- Function point approach is independent of the language, tools, or methodologies used for implementation; i.e. they do not take into consideration programming languages, data base management systems, processing hardware or any other data base technology.
- Function points can be estimated from requirement specification or design specification, thus making it possible to estimate development efforts in early phases of development.

Software Project Planning

- Function points are directly linked to the statement of requirements; any change of requirements can easily be followed by a re-estimate.
- Function points are based on the system user's external view of the system, non-technical users of the software system have a better understanding of what function points are measuring.

Counting function points

Functional Units	Weighting factors		
	Low	Average	High
External Inputs (EI)	3	4	6
External Output (EO)	4	5	7
External Inquiries (EQ)	3	4	6
External logical files (ILF)	7	10	15
External Interface files (EIF)	5	7	10

Table 1 : Functional units with weighting factors

Software Project Planning

Table 2: UFP calculation table

Functional Units	Count Complexity			Complexity Totals	Functional Unit Totals
External Inputs (EIs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External Outputs (EOs)		Low x 4	=		
		Average x 5	=		
		High x 7	=		
External Inquiries (EQs)		Low x 3	=		
		Average x 4	=		
		High x 6	=		
External logical Files (ILFs)		Low x 7	=		
		Average x 10	=		
		High x 15	=		
External Interface Files (EIFs)		Low x 5	=		
		Average x 7	=		
		High x 10	=		
Total Unadjusted Function Point Count					

The weighting factors are identified for all functional units and multiplied with the functional units accordingly. The procedure for the calculation of Unadjusted Function Point (UFP) is given in table shown above.

Software Project Planning

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

W_{ij} : It is the entry of the ith row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

Software Project Planning

Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to $[0.65 + 0.01 \times \Sigma F_i]$. The F_i (i=1 to 14) are the degree of influence and are based on responses to questions noted in table 3.

Software Project Planning

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.

Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Software Project Planning

Functions points may compute the following important metrics:

Productivity	=	FP / persons-months
Quality	=	Defects / FP
Cost	=	Rupees / FP
Documentation	=	Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Functions Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method). An ISO standard for function point method is also being developed.

Example: 4.1

Consider a project with the following functional units:

Number of user inputs	= 50
Number of user outputs	= 40
Number of user enquiries	= 35
Number of user files	= 06
Number of external interfaces	= 04

Assume all complexity adjustment factors and weighting factors are average. Compute the function points for the project.

Solution

We know

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

UFP = 50 x 4 + 40 x 5 + 35 x 4 + 6 x 10 + 4 x 7

= 200 + 200 + 140 + 60 + 28 = 628

CAF = $(0.65 + 0.01\ \Sigma F_i)$

= (0.65 + 0.01 (14 x 3)) = 0.65 + 0.42 = 1.07

FP = UFP x CAF

= 628 x 1.07 = 672

Example:4.2

An application has the following:

10 low external inputs, 12 high external outputs, 20 low internal logical files, 15 high external interface files, 12 average external inquiries, and a value of complexity adjustment factor of 1.10.

What are the unadjusted and adjusted function point counts ?

Solution

Unadjusted function point counts may be calculated using as:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

= 10 x 3 + 12 x 7 + 20 x 7 + 15 + 10 + 12 x 4

= 30 + 84 +140 + 150 + 48

= 452

FP = UFP x CAF

= 452 x 1.10 = 497.2.

Software Project Planning

Example: 4.3

Consider a project with the following parameters.

(*i*) External Inputs:

(a) 10 with low complexity

(b) 15 with average complexity

(c) 17 with high complexity

(*ii*) External Outputs:

(a) 6 with low complexity

(b) 13 with high complexity

(*iii*) External Inquiries:

(a) 3 with low complexity

(b) 4 with average complexity

(c) 2 high complexity

Software Project Planning

(*iv*) Internal logical files:

(a) 2 with average complexity

(b) 1 with high complexity

(v) External Interface files:

(a) 9 with low complexity

In addition to above, system requires

i. Significant data communication

ii. Performance is very critical

iii. Designed code may be moderately reusable

iv. System is not designed for multiple installation in different organizations.

Other complexity adjustment factors are treated as average. Compute the function points for the project.

Software Project Planning

Solution: Unadjusted function points may be counted using table 2

Functional Units	Count	Complexity		Complexity Totals	Functional Unit Totals
External Inputs (EIs)	10	Low x 3	=	30	
	15	Average x 4	=	60	
	17	High x 6	=	102	192
External Outputs (EOs)	6	Low x 4	=	24	
	0	Average x 5	=	0	
	13	High x 7	=	91	115
External Inquiries (EQs)	3	Low x 3	=	9	
	4	Average x 4	=	16	
	2	High x 6	=	12	37
External logical Files (ILFs)	0	Low x 7	=	0	
	2	Average x 10	=	20	
	1	High x 15	=	15	35
External Interface Files (EIFs)	9	Low x 5	=	45	
	0	Average x 7	=	0	
	0	High x 10	=	0	45
Total Unadjusted Function Point Count					424

Software Project Planning

$$\sum_{i=1}^{14} F_i = 3+4+3+5+3+3+3+3+3+3+2+3+0+3=41$$

$$\begin{aligned} CAF &= (0.65 + 0.01 \text{ x } \Sigma F_i) \\ &= (0.65 + 0.01 \text{ x } 41) \\ &= 1.06 \\ FP &= UFP \text{ x } CAF \\ &= 424 \text{ x } 1.06 \\ &= 449.44 \end{aligned}$$

Hence $\boxed{FP = 449}$

Software Project Planning

Relative Cost of Software Phases

Software Project Planning

Cost to Detect and Fix Faults

Cost Estimation

A number of estimation techniques have been developed and are having following attributes in common :

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

Software Project Planning

Static, Single Variable Models

Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc. They all depend on the same variable used as predictor (say, size). An example of the most common equations is :

$$\mathbf{C = a\,L^{b}} \qquad \text{(i)}$$

C is the cost, L is the size and a,b are constants

$$E = 1.4\,L^{0.93}$$

$$DOC = 30.4\,L^{0.90}$$

$$D = 4.6\,L^{0.26}$$

Effort (E in Person-months), documentation (DOC, in number of pages) and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

Static, Multivariable Models

These models are often based on equation (i), they actually depend on several variables representing various aspects of the software development environment, for example method used, user participation, customer oriented changes, memory constraints, etc.

$$E = 5.2\ L^{0.91}$$

$$D = 4.1\ L^{0.36}$$

The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i$$

Software Project Planning

Example: 4.4

Compare the Walston-Felix model with the SEL model on a software development expected to involve 8 person-years of effort.

(a)Calculate the number of lines of source code that can be produced.

(b)Calculate the duration of the development.

(c)Calculate the productivity in LOC/PY

(d)Calculate the average manning

Solution

The amount of manpower involved = 8 PY = 96 person-months

(a) Number of lines of source code can be obtained by reversing equation to give:

$$L = (E/a)^{1/b}$$

Then

$$L(SEL) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$

$$L(SEL) = (96/5.2)^{1/0.91} = 24632 \text{ LOC.}$$

Software Project Planning

(b) Duration in months can be calculated by means of equation

$$D(SEL) = 4.6\ (L)^{0.26}$$

$$= 4.6\ (94.264)^{0.26} = 15 \text{ months}$$

$$D(W\text{-}F) = 4.1\ L^{0.36}$$

$$= 4.1(24.632)^{0.36} = 13 \text{ months}$$

(c) Productivity is the lines of code produced per person/month (year)

$$P(SEL) = \frac{94264}{8} = 11783\ LOC / Person - Years$$

$$P(W - F) = \frac{24632}{8} = 3079\ LOC / Person - Years$$

Software Project Planning

(d) Average manning is the average number of persons required per month in the project.

$$M(SEL) = \frac{96P - M}{15M} = 6.4 Persons$$

$$M(W - F) = \frac{96P - M}{13M} = 7.4 Persons$$

The Constructive Cost Model (COCOMO)

Constructive Cost model
(COCOMO)

Basic Intermediate Detailed

Model proposed by
B. W. Boehm's
through his book
Software Engineering Economics in 1981

Software Project Planning

Software Project Planning

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

Table 4: The comparison of three COCOMO modes

Basic Model

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software Project Planning

Software Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Table 4(a): Basic COCOMO coefficients

Software Project Planning

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} KLOC / PM$$

Example: 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

Solution

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (KLOC)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

Software Project Planning

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$

Software Project Planning

Example: 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12$ PM

$D = 2.5(1133.12)^{0.35} = 29.3$ PM

Average staff size $(SS) = \frac{E}{D} Persons$

$$= \frac{1133.12}{29.3} = 38.67 Persons$$

Software Project Planning

$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\, KLOC / PM$$

$$P = 176\, LOC / PM$$

Intermediate Model

Cost drivers

(*i*) Product Attributes

- Required s/w reliability
- Size of application database
- Complexity of the product

(*ii*) Hardware Attributes

- Run time performance constraints
- Memory constraints
- Virtual machine volatility
- Turnaround time

Software Project Planning

(*iii*) Personal Attributes

- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(*iv*) Project Attributes

- Modern programming practices
- Use of software tools
- Required development Schedule

Software Project Planning

Multipliers of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Software Project Planning

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Table 5: Multiplier values for effort calculations

Software Project Planning

Intermediate COCOMO equations

$$E = a_i (KLOC)^{b_i} * EAF$$

$$D = c_i (E)^{d_i}$$

Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

Detailed COCOMO Model

Detailed COCOMO

Phase-Sensitive effort multipliers

Cost drivers

design & test

Manpower allocation for each phase

Three level product hierarchy

Modules subsystem

System level

Software Project Planning

Development Phase

Plan / Requirements

EFFORT	:	6% to 8%
DEVELOPMENT TIME	:	10% to 40%

% depend on mode & size

Software Project Planning

Design

Effort	:	16% to 18%
Time	:	19% to 38%

Programming

Effort	:	48% to 68%
Time	:	24% to 64%

Integration & Test

Effort	:	16% to 34%
Time	:	18% to 34%

Principle of the effort estimate

Size equivalent

As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I$$

The size equivalent is obtained by

$$S\ (\text{equivalent}) = (S \times A) / 100$$

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Lifecycle Phase Values of μ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded extra large S≈320	0.08	0.18	0.24	0.24	0.34

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Lifecycle Phase Values of τ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded extra large S≈320	0.40	0.38	0.16	0.16	0.30

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Software Project Planning

Distribution of software life cycle:

1. Requirement and product design
   (a) Plans and requirements
   (b) System design
2. Detailed Design
   (a) Detailed design
3. Code & Unit test
   (a) Module code & test
4. Integrate and Test
   (a) Integrate & Test

Software Project Planning

Example: 4.7

A new project with estimated 400 KLOC embedded system has to be developed. Project manager has a choice of hiring from two pools of developers: Very highly capable with very little experience in the programming language being used

Or

Developers of low quality but a lot of experience with the programming language. What is the impact of hiring all developers from one or the other pool ?

Solution

This is the case of embedded mode and model is intermediate COCOMO.

Hence $$E = a_i (KLOC)^{d_i}$$

$$= 2.8\ (400)^{1.20} = 3712 \text{ PM}$$

Case I: Developers are very highly capable with very little experience in the programming being used.

EAF = 0.82 x 1.14 = 0.9348

E = 3712 x .9348 = 3470 PM

D = $2.5\ (3470)^{0.32}$ = 33.9 M

Software Project Planning

Case II: Developers are of low quality but lot of experience with the programming language being used.

EAF = 1.29 x 0.95 = 1.22

E = 3712 x 1.22 = 4528 PM

D = $2.5\ (4528)^{0.32}$ = 36.9 M

Case II requires more effort and time. Hence, low quality developers with lot of programming language experience could not match with the performance of very highly capable developers with very litter experience.

Software Project Planning

Example: 4.8

Consider a project to develop a full screen editor. The major components identified are:

I. Screen edit

II. Command Language Interpreter

III. File Input & Output

IV. Cursor Movement

V. Screen Movement

The size of these are estimated to be 4k, 2k, 1k, 2k and 3k delivered source code lines. Use COCOMO to determine

1. Overall cost and schedule estimates (assume values for different cost drivers, with at least three of them being different from 1.0)
2. Cost & Schedule estimates for different phases.

Solution

Size of five modules are:

Screen edit	= 4 KLOC
Command language interpreter	= 2 KLOC
File input and output	= 1 KLOC
Cursor movement	= 2 KLOC
Screen movement	= 3 KLOC
Total	**= 12 KLOC**

Software Project Planning

Let us assume that significant cost drivers are

i. Required software reliability is high, i.e.,1.15

ii. Product complexity is high, i.e.,1.15

iii. Analyst capability is high, i.e.,0.86

iv. Programming language experience is low,i.e.,1.07

v. All other drivers are nominal

EAF = 1.15x1.15x0.86x1.07 = 1.2169

Software Project Planning

(a) The initial effort estimate for the project is obtained from the following equation

$$E = a_i\,(KLOC)^{bi} \times EAF$$

$$= 3.2(12)^{1.05} \times 1.2169 = 52.91 \text{ PM}$$

Development time $D = C_i(E)^{di}$

$$= 2.5(52.91)^{0.38} = 11.29 \text{ M}$$

(b) Using the following equations and referring Table 7, phase wise cost and schedule estimates can be calculated.

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Software Project Planning

Since size is only 12 KLOC, it is an organic small model. Phase wise effort distribution is given below:

System Design	= 0.16 x 52.91 = 8.465 PM
Detailed Design	= 0.26 x 52.91 = 13.756 PM
Module Code & Test	= 0.42 x 52.91 = 22.222 PM
Integration & Test	= 0.16 x 52.91 = 8.465 Pm

Now Phase wise development time duration is

System Design	= 0.19 x 11.29 = 2.145 M
Detailed Design	= 0.24 x 11.29 = 2.709 M
Module Code & Test	= 0.39 x 11.29 = 4.403 M
Integration & Test	= 0.18 x 11.29 = 2.032 M

COCOMO-II

The following categories of applications / projects are identified by COCOMO-II and are shown in fig. 4 shown below:

Fig. 4 : Categories of applications / projects

Software Project Planning

Stage No	Model Name	Application for the types of projects	Applications
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project.

Table 8: Stages of COCOMO-II

Application Composition Estimation Model

Fig.5: Steps for the estimation of effort in person months

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

i. Assess object counts: Estimate the number of screens, reports and 3 GL components that will comprise this application.

ii. Classification of complexity levels: We have to classify each object instance into simple, medium and difficult complexity levels depending on values of its characteristics.

Number of views contained	*# and sources of data tables*		
	Total < 4 (< 2 server < 3 client)	*Total < 8 (2 – 3 server 3 – 5 client)*	*Total 8 + (> 3 server, > 5 client)*
< 3	Simple	Simple	Medium
3 – 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

Table 9 (a): For screens

Software Project Planning

Number of sections contained	# and sources of data tables		
	Total < 4 (< 2 server < 3 client)	Total < 8 (2 – 3 server 3 – 5 client)	Total 8 + (> 3 server, > 5 client)
0 or 1	Simple	Simple	Medium
2 or 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

Table 9 (b): For reports

Software Project Planning

iii. Assign complexity weight to each object : The weights are used for three object types i.e., screen, report and 3GL components using the Table 10.

Object Type	*Complexity Weight*		
	Simple	*Medium*	*Difficult*
Screen	1	2	3
Report	2	5	8
3GL Component	—	—	10

Table 10: Complexity weights for each level

Software Project Planning

iv. Determine object points: Add all the weighted object instances to get one number and this known as object-point count.

v. Compute new object points: We have to estimate the percentage of reuse to be achieved in a project. Depending on the percentage reuse, the new object points (NOP) are computed.

$$\textbf{NOP} = \frac{(\text{object points}) * (100 - \%\text{reuse})}{100}$$

NOP are the object points that will need to be developed and differ from the object point count because there may be reuse.

Software Project Planning

vi. Calculation of productivity rate: The productivity rate can be calculated as:

Productivity rate (PROD) = NOP/Person month

Developer's experience & capability; ICASE maturity & capability	*PROD (NOP/PM)*
Very low	4
Low	7
Nominal	13
High	25
Very high	50

Table 11: Productivity values

Software Project Planning

vii. Compute the effort in Persons-Months: When PROD is known, we may estimate effort in Person-Months as:

$$\textbf{Effort in PM} = \frac{NOP}{PROD}$$

Software Project Planning

Example: 4.9

Consider a database application project with the following characteristics:

I. The application has 4 screens with 4 views each and 7 data tables for 3 servers and 4 clients.

II. The application may generate two report of 6 sections each from 07 data tables for two server and 3 clients. There is 10% reuse of object points.

The developer's experience and capability in the similar environment is low. The maturity of organization in terms of capability is also low. Calculate the object point count, New object points and effort to develop such a project.

Solution

This project comes under the category of application composition estimation model.

Number of screens = 4 with 4 views each

Number of reports = 2 with 6 sections each

From Table 9 we know that each screen will be of medium complexity and each report will be difficult complexity.

Using Table 10 of complexity weights, we may calculate object point count.

$$= 4 \times 2 + 2 \times 8 = 24$$

$$\textbf{NOP} = \frac{24 * (100 - 10)}{100} = 21.6$$

Software Project Planning

Table 11 gives the low value of productivity (PROD) i.e. 7.

$$\text{Efforts in PM} = \frac{\text{NOP}}{\text{PROD}}$$

$$\text{Efforts} = \frac{21.6}{7} = 3.086 \text{ PM}$$

The Early Design Model

The COCOMO-II models use the base equation of the form

$$PM_{nominal} = A * (size)^B$$

where

$PM_{nominal}$ = Effort of the project in person months

A = Constant representing the nominal productivity, provisionally set to 2.5

B = Scale factor

Size = Software size

Software Project Planning

Scale factor	Explanation	Remarks
Precedentness	Reflects the previous experience on similar projects. This is applicable to individuals & organization both in terms of expertise & experience	Very low means no previous experiences, Extra high means that organization is completely familiar with this application domain.
Development flexibility	Reflect the degree of flexibility in the development process.	Very low means a well defined process is used. Extra high means that the client gives only general goals.
Architecture/ Risk resolution	Reflect the degree of risk analysis carried out.	Very low means very little analysis and Extra high means complete and through risk analysis.

Cont…

Table 12: Scaling factors required for the calculation of the value of B

Scale factor	Explanation	Remarks
Team cohesion	Reflects the team management skills.	Very low means no previous experiences, Extra high means that organization is completely familiar with this application domain.
Process maturity	Reflects the process maturity of the organization. Thus it is dependent on SEI-CMM level of the organization.	Very low means organization has no level at all and extra high means organization is related as highest level of SEI-CMM.

Table 12: Scaling factors required for the calculation of the value of B

Software Project Planning

Scaling factors	Very low	Low	Nominal	High	Very high	Extra high
Precedent ness	6.20	4.96	3.72	2.48	1.24	0.00
Development flexibility	5.07	4.05	3.04	2.03	1.01	0.00
Architecture/ Risk resolution	7.07	5.65	4.24	2.83	1.41	0.00
Team cohesion	5.48	4.38	3.29	2.19	1.10	0.00
Process maturity	7.80	6.24	4.68	3.12	1.56	0.00

Table 13: Data for the Computation of B

The value of B can be calculated as:

B=0.91 + 0.01 * (Sum of rating on scaling factors for the project)

Early design cost drivers

There are seven early design cost drivers and are given below:

i. Product Reliability and Complexity (RCPX)

ii. Required Reuse (RUSE)

iii. Platform Difficulty (PDIF)

iv. Personnel Capability (PERS)

v. Personnel Experience (PREX)

vi. Facilities (FCIL)

vii. Schedule (SCED)

Post architecture cost drivers

There are 17 cost drivers in the Post Architecture model. These are rated on a scale of 1 to 6 as given below :

Very Low	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
1	2	3	4	5	6

The list of seventeen cost drivers is given below :

i. Reliability Required (RELY)

ii. Database Size (DATA)

iii. Product Complexity (CPLX)

iv. Required Reusability (RUSE)

Software Project Planning

v. Documentation (DOCU)

vi. Execution Time Constraint (TIME)

vii. Main Storage Constraint (STOR)

viii. Platform Volatility (PVOL)

ix. Analyst Capability (ACAP)

x. Programmers Capability (PCAP)

xi. Personnel Continuity (PCON)

xii. Analyst Experience (AEXP)

Software Project Planning

xiii. Programmer Experience (PEXP)

xiv. Language & Tool Experience (LTEX)

xv. Use of Software Tools (TOOL)

xvi. Site Locations & Communication Technology between Sites (SITE)

xvii. Schedule (SCED)

Mapping of early design cost drivers and post architecture cost drivers

The 17 Post Architecture Cost Drivers are mapped to 7 Early Design Cost Drivers and are given in Table 14

Early Design Cost Drivers	Counter part Combined Post Architecture Cost drivers
RCPX	RELY, DATA, CPLX, DOCU
RUSE	RUSE
PDIF	TIME, STOR, PVOL
PERS	ACAP, PCAP, PCON
PREX	AEXP, PEXP, LTEX
FCIL	TOOL, SITE
SCED	SCED

Table 14: Mapping table

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

Product of cost drivers for early design model

i. Product Reliability and Complexity (RCPX): The cost driver combines four Post Architecture cost drivers which are RELY, DATA, CPLX and DOCU.

RCPX	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of RELY, DATA, CPLX, DOCU ratings	5, 6	7, 8	9-11	12	13-15	16-18	19-21
Emphasis on reliability, documentation	Very Little	Little	Some	Basic	Strong	Very Strong	Extreme
Product complexity	Very Simple	Simple	Some	Moderate	Complex	Very Complex	Extremely Complex
Database size	Small	Small	Small	Moderate	Large	Very Large	Very Large

Software Project Planning

ii. Required Reuse (RUSE) : This early design model cost driver is same as its Post architecture Counterpart. The RUSE rating levels are (as per Table 16):

	Vary Low	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RUSE	1	2	3	4	5	6
		None	Across project	Across program	Across product line	Across multiple product line

Software Project Planning

iii. Platform Difficulty (PDIF) : This cost driver combines TIME, STOR and PVOL of Post Architecture Cost Drivers.

PDIF	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of Time, STOR & PVOL ratings	8	9	10-12	13-15	16-17
Time & storage constraint	≤ 50%	≤ 50%	65%	80%	90%
Platform Volatility	Very stable	Stable	Somewhat stable	Volatile	Highly Volatile

Software Project Planning

iv. Personnel Capability (PERS) : This cost driver combines three Post Architecture Cost Drivers. These drivers are ACAP, PCAP and PCON.

PERS	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of ACAP, PCAP, PCON ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Combined ACAP & PCAP Percentile	20%	39%	45%	55%	65%	75%	85%
Annual Personnel Turnover	45%	30%	20%	12%	9%	5%	4%

Software Project Planning

v. Personnel Experience (PREX) : This early design driver combines three Post Architecture Cost Drivers, which are AEXP, PEXP and LTEX.

PREX	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of AEXP, PEXP and LTEX ratings	3, 4	5, 6	7, 8	9	10, 11	12, 13	14, 15
Applications, Platform, Language & Tool Experience	≤ 3 months	5 months	9 months	1 year	2 year	4 year	6 year

Software Project Planning

vi. Facilities (FCIL): This depends on two Post Architecture Cost Drivers, which are TOOL and SITE.

FCIL	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
Sum of TOOL & SITE ratings	2	3	4, 5	6	7, 8	9, 10	11
Tool support	Minimal	Some	Simple CASE tools	Basic life cycle tools	Good support of tools	Very strong use of tools	Very strong & well integrated tools
Multisite conditions development support	Weak support of complex multisite development	Some support	Moderate support	Basic support	Strong support	Very strong support	Very strong support

Software Project Planning

vii. Schedule (SCED) : This early design cost driver is the same as Post Architecture Counterpart and rating level are given below using table 16.

SCED	*Very Low*	*Low*	*Nominal*	*High*	*Very High*
Schedule	75% of Nominal	85%	100%	130%	160%

Software Project Planning

The seven early design cost drivers have been converted into numeric values with a Nominal value 1.0. These values are used for the calculation of a factor called "Effort multiplier" which is the product of all seven early design cost drivers. The numeric values are given in Table 15.

Early design Cost drivers	*Extra Low*	*Very Low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RCPX	.73	.81	.98	1.0	1.30	1.74	2.38
RUSE	—	—	0.95	1.0	1.07	1.15	1.24
PDIF	—	—	0.87	1.0	1.29	1.81	2.61
PERS	2.12	1.62	1.26	1.0	0.83	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	—	1.43	1.14	1.0	1.0	1.0	—

Table 15: Early design parameters

Software Project Planning

The early design model adjusts the nominal effort using 7 effort multipliers (EMs). Each effort multiplier (also called drivers) has 7 possible weights as given in Table 15. These factors are used for the calculation of adjusted effort as given below:

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{7} EM_i \right]$$

$PM_{adjusted}$ effort may very even up to 400% from $PM_{nominal}$

Hence $PM_{adjusted}$ is the fine tuned value of effort in the early design phase

Example: 4.10

A software project of application generator category with estimated 50 KLOC has to be developed. The scale factor (B) has low precedentness, high development flexibility and low team cohesion. Other factors are nominal. The early design cost drivers like platform difficult (PDIF) and Personnel Capability (PERS) are high and others are nominal. Calculate the effort in person months for the development of the project.

Solution

Here B = 0.91 + 0.01 * (Sum of rating on scaling factors for the project)

= 0.91 + 0.01 * (4.96 + 2.03 + 4.24 + 4.38 + 4.68)

= 0.91 + 0.01(20.29)=1.1129

$PM_{nominal} = A*(size)^B$

$= 2.5 * (50)^{1.1129} = 194.41$ Person months

The 7 cost drivers are

PDIF = high (1.29)

PERS = high (0.83)

RCPX = nominal (1.0)

RUSE = nominal (1.0)

PREX = nominal (1.0)

FCIL = nominal (1.0)

SCEO = nominal (1.0)

Software Project Planning

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{7} EM_i \right]$$

= 194.41 * [1.29 x 0.83)

= 194.41 x 1.07

= 208.155 Person months

Post Architecture Model

The post architecture model is the most detailed estimation model and is intended to be used when a software life cycle architecture has been completed. This model is used in the development and maintenance of software products in the application generators, system integration or infrastructure sectors.

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{17} EM_i \right]$$

EM : Effort multiplier which is the product of 17 cost drivers.

The 17 cost drivers of the Post Architecture model are described in the table 16.

Software Project Planning

Cost driver	*Purpose*	*Very low*	*Low*	*Nominal*	*High*	*Very High*	*Extra High*
RELY (Reliability required)	Measure of the extent to which the software must perform its intended function over a period of time	Only slight inconvenience	Low, easily recoverable losses	Moderate, easily recoverable losses	High financial loss	Risk to human life	—
DATA (Data base size)	Measure the affect of large data requirements on product development	—	$\frac{\text{Database size(D)}}{\text{Prog. size (P)}} < 10$	$10 \le \frac{D}{P} < 100$	$100 \le \frac{D}{P} < 1000$	$\frac{D}{P} \ge 1000$	—
CPLX (Product complexity)	Complexity is divided into five areas: Control operations, computational operations, device dependent operations, data management operations & User Interface management operations.	See Table 4.17					
DOCU Documentation	Suitability of the project's documentation to its life cycle needs	Many life cycle needs uncovered	Some needs uncovered	Adequate	Excessive for life cycle needs	Very Excessive	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

Software Project Planning

TIME (Execution Time constraint)	Measure of execution time constraint on software	—	—	≤ 50% use of a available execution time	70%	85%	95%
STOR (Main storage constraint)	Measure of main storage constraint on software	—	—	≤ 50% use of available storage	70%	85%	95%
PVOL (Platform Volatility)	Measure of changes to the OS, compilers, editors, DBMS etc.	—	Major changes every 12 months & minor changes every 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 week Minor: 2 days	—
ACAP (Analyst capability)	Should include analysis and design ability, efficiency & thoroughness, and communication skills.	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

Software Project Planning

PCAP (Programmers capability)	Capability of Programmers as a team. It includes ability, efficiency, thoroughness & communication skills	15th Percentile	35th Percentile	55th Percentile	75th Percentile	90th Percentile	—
PCON (Personnel Continuity)	Rating is in terms of Project's annual personnel turnover	48%/year	24%/year	12%/year	6%/year	3%/year	—
AEXP (Applications Experience)	Rating is dependent on level of applications experience.	≤ 2 months	6 months	1 year	3 year	6 year	—
PEXP (Platform experience)	Measure of Platform experience	≤ 2 months	6 months	1 year	3 year	6 year	—

Table 16: Post Architecture Cost Driver rating level summary

Cont…

LTEX (Language & Tool experience)	Rating is for Language & tool experience	≤ 2 months	6 months	1 year	3 year	6 year	—
TOOL (Use of software tools)	It is the indicator of usage of software tools	No use	Beginning to use	Some use	Good use	Routine & habitual use	—
SITE (Multisite development)	Site location & Communication technology between sites	International with some phone & mail facility	Multicity & multi company with individual phones, FAX	Multicity & multi company with Narrow band mail	Same city or Metro with wideband electronic communication	Same building or complex with wideband electronic communication & Video conferencing	Fully co-located with interactive multimedia
SCED (Required Development Schedule)	Measure of Schedule constraint. Ratings are defined in terms of percentage of schedule stretch-out or acceleration with respect to nominal schedule	75% of nominal	85%	100%	130%	160%	—

Table 16: Post Architecture Cost Driver rating level summary

Software Project Planning

Product complexity is based on control operations, computational operations, device dependent operations, data management operations and user interface management operations. Module complexity rating are given in table 17.

The numeric values of these 17 cost drivers are given in table 18 for the calculation of the product of efforts i.e., effort multiplier (EM). Hence PM adjusted is calculated which will be a better and fine tuned value of effort in person months.

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Very Low	Straight-line code with a few non-nested structured programming operators: Dos. Simple module composition via procedure calls or simple scripts.	Evaluation of simple expressions: e.g., A=B+C*(D-E)	Simple read, write statements with simple formats.	Simple arrays in main memory. Simple COTSDB queries, updates.	Simple input forms, report generators.
Low	Straight forward nesting of structured programming operators. Mostly simple predicates	Evaluation of moderate-level expressions: e.g., D=SQRT(B**2-4*A*C)	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level.	Single file sub setting with no data structure changes, no edits, no intermediate files, Moderately complex COTS-DB queries, updates.	User of simple graphics user interface (GUI) builders.

Table 17: Module complexity ratings

Cont…

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Nominal	Mostly simple nesting. Some inter module control Decision tables. Simple callbacks or message passing, including middleware supported distributed processing.	Use of standard maths and statistical routines. Basic matrix/ vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits. Complex COTS-DB queries, updates.	Simple use of widget set.
High	Highly nested structured programming operators with many compound predicates. Queue and stack control. Homogeneous, distributed processing. Single processor soft real time control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, round off concerns.	Operations at physical I/O level (physical storage address translations; seeks, read etc.) Optimized I/O overlap.	Simple triggers activated by data stream contents. Complex data restructuring.	Widget set development and extension. Simple voice I/O multimedia.

Table 17: Module complexity ratings

Cont…

Software Project Planning

	Control Operations	Computational Operations	Device-dependent Operations	Data management Operations	User Interface Management Operations
Very High	Reentrant and recursive coding. Fixed-priority interrupt handling. Task synchronization, complex callbacks, heterogeneous distributed processing. Single processor hard real time control.	Difficult but structured numerical analysis: near singular matrix equations, partial differential equations. Simple parallelization.	Routines for interrupt diagnosis, servicing, masking. Communication line handling. Performance intensive embedded systems.	Distributed database coordination. Complex triggers. Search optimization.	Moderately complex 2D/3D, dynamic graphics, multimedia.
Extra High	Multiple resource scheduling with dynamically changing priorities. Microcode-level control. Distributed hard real time control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data. Complex parallelization.	Device timing dependent coding, micro programmed operations. Performance critical embedded systems.	Highly coupled, dynamic relational and object structures. Natural language data management.	Complex multimedia, virtual reality.

Table 17: Module complexity ratings

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	1.15	1.39	
DATA		0.93	1.00	1.09	1.19	
CPLX	0.75	0.88	1.00	1.15	1.30	1.66
RUSE		0.91	1.00	1.14	1.29	1.49
DOCU	0.89	0.95	1.00	1.06	1.13	
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	

Table 18: 17 Cost Drivers

Cont…

Cost Driver	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Table 18: 17 Cost Drivers

Schedule estimation

Development time can be calculated using $PM_{adjusted}$ as a key factor and the desired equation is:

$$TDEV_{\text{nominal}} = [\phi \times (PM_{adjusted})^{(0.28+0.2(B-0.091))}] * \frac{SCED\%}{100}$$

where Φ = constant, provisionally set to 3.67

$TDEV_{nominal}$ = calendar time in months with a scheduled constraint

B = Scaling factor

$PM_{adjusted}$ = Estimated effort in Person months (after adjustment)

Size measurement

Size can be measured in any unit and the model can be calibrated accordingly. However, COCOMO II details are:

i. Application composition model uses the size in object points.

ii. The other two models use size in KLOC

Early design model uses unadjusted function points. These function points are converted into KLOC using Table 19. Post architecture model may compute KLOC after defining LOC counting rules. If function points are used, then use unadjusted function points and convert it into KLOC using Table 19.

Software Project Planning

Language	SLOC/UFP
Ada	71
AI Shell	49
APL	32
Assembly	320
Assembly (Macro)	213
ANSI/Quick/Turbo Basic	64
Basic-Compiled	91
Basic-Interpreted	128
C	128
C++	29

Table 19: Converting function points to lines of code

Cont…

Software Project Planning

Language	SLOC/UFP
ANSI Cobol 85	91
Fortan 77	105
Forth	64
Jovial	105
Lisp	64
Modula 2	80
Pascal	91
Prolog	64
Report Generator	80
Spreadsheet	6

Table 19: Converting function points to lines of code

Example: 4.11

Consider the software project given in example 4.10. Size and scale factor (B) are the same. The identified 17 Cost drivers are high reliability (RELY), very high database size (DATA), high execution time constraint (TIME), very high analyst capability (ACAP), high programmers capability (PCAP). The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

Solution

Here B = 1.1129

$PM_{nominal}$ = 194.41 Person-months

$$PM_{adjusted} = PM_{\text{nominal}} \times \left[\prod_{i=7}^{17} EM_i \right]$$

= 194.41 x (1.15 x 1.19 x 1.11 x 0.67 x 0.87)

= 194.41 x 0.885

= 172.05 Person-months

Putnam Resource Allocation Model

Norden of IBM

Rayleigh curve

Model for a range of hardware development projects.

Fig.6: The Rayleigh manpower loading curve

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Software Project Planning

Putnam observed that this curve was a close approximation at project level and software subsystem level.

No. of projects = 150

The Norden / Rayleigh Curve

The curve is modeled by differential equation

$$m(t) = \frac{dy}{dt} = 2kate^{-at^2} \qquad \text{---------- (1)}$$

$\frac{dy}{dt}$ = manpower utilization rate per unit time

a = parameter that affects the shape of the curve

K = area under curve in the interval [0, ∞]

t = elapsed time

Software Project Planning

On Integration on interval [o, t]

$$y(t) = K\,[1-e^{-at^2}] \quad \text{----------(2)}$$

Where y(t): cumulative manpower used upto time t.

$$y(0) = 0$$

$$y(\infty) = k$$

The cumulative manpower is null at the start of the project, and grows monotonically towards the total effort K (area under the curve).

Software Project Planning

$$\frac{d^2y}{dt^2} = 2kae^{-at^2}[1-2at^2] = 0$$

$$t_d^2 = \frac{1}{2a}$$

"t_d": time where maximum effort rate occurs

Replace "t_d" for *t* in equation (2)

$$E = y(t) = k\left(1 - e^{\frac{t_d^2}{2t_d^2}}\right) = K(1 - e^{-0.5})$$

$$E = y(t) = 0.3935\ k$$

$$a = \frac{1}{2t_d^2}$$

Replace "a" with $\frac{1}{2t_d^2}$ in the Norden/Rayleigh model. By making this substitution in equation we have

$$m(t) = \frac{2K}{2t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

$$= \frac{K}{t_d^2} te^{-\frac{t^2}{2t_d^2}}$$

Software Project Planning

Fig.7: Influence of parameter 'a' on the manpower distribution

Software Project Planning

At time $t=t_d$, peak manning $m(t_d)$ is obtained and denoted by m_o.

$$m_o = \frac{k}{t_d \sqrt{e}}$$

k = Total project cost/effort in person-years.

t_d = Delivery time in years

m_0 = No. of persons employed at the peak

e = 2.71828

Example: 4.12

A software development project is planned to cost 95 MY in a period of 1 year and 9 months. Calculate the peak manning and average rate of software team build up.

Solution

Software development cost k=95 MY

Peak development time t_d = 1.75 years

Peak manning $m_o = \frac{k}{t_d \sqrt{e}}$

$$\frac{95}{1.75 \times 1.648} = 32.94 = 33 \ persons$$

Average rate of software team build up

$$= \frac{m_0}{t_d} = \frac{33}{1.75} = 18.8 \, persons / \ year \ or \ 1.56 \ person / month$$

Example: 4.13

Consider a large-scale project for which the manpower requirement is K=600 PY and the development time is 3 years 6 months.

(a)Calculate the peak manning and peak time.

(b)What is the manpower cost after 1 year and 2 months?

Solution

(a) We know t_d=3 years and 6 months = 3.5 years

NOW $$m_0 = \frac{K}{t_d \sqrt{e}}$$

$$\therefore \quad m_0 = 600/(3.5\text{x}1.648) \cong 104 \text{ persons}$$

(b) We know

$$y(t) = K\left[1 - e^{-at^2}\right]$$

t = 1 year and 2 months

= 1.17 years

$$a = \frac{1}{2t_d^2} = \frac{1}{2\times(3.5)^2} = 0.041$$

$$y(1.17) = 600\left[1 - e^{-0.041(1.17)^2}\right]$$

= 32.6 PY

Difficulty Metric

Slope of manpower distribution curve at start time t=0 has some useful properties.

$$m'(t) = \frac{d^2 y}{dt^2} = 2kae^{-at^2}(1 - 2at^2)$$

Then, for t=0

$$m'(0) = 2Ka = \frac{2K}{2t_d^2} = \frac{K}{t_d^2}$$

Software Project Planning

The ratio $\frac{K}{t_d^2}$ is called difficulty and denoted by D, which is measured in person/year :

$$D = \frac{k}{t_d^2} \text{ persons/year}$$

Software Project Planning

Project is difficult to develop if

Manpower demand is high

When time schedule is short

Software Project Planning

Peak manning is defined as:

$$m_0 = \frac{k}{t_d \sqrt{e}}$$

$$D = \frac{k}{t_d^2} = \frac{m_0 \sqrt{e}}{t_d}$$

Thus difficult projects tend to have a higher peak manning for a given development time, which is in line with Norden's observations relative to the parameter "a".

Manpower buildup

D is dependent upon "K". The derivative of D relative to "K" and "t_d" are

$$D'(t_d) = \frac{-2k}{t_d^3} \; persons / year^2$$

$$D'(k) = \frac{1}{t_d^2} \; year^{-2}$$

Software Project Planning

$D^1(K)$ will always be very much smaller than the absolute value of $D^1(t_d)$. This difference in sensitivity is shown by considering two projects

Project A : Cost = 20 PY & t_d = 1 year
Project B : Cost = 120 PY & t_d = 2.5 years

The derivative values are

Project A : $D`(t_d) = -40$ & $D`(K) = 1$
Project B : $D`(t_d) = -15.36$ & $D`(K) = 0.16$

This shows that a given software development is time sensitive.

Software Project Planning

Putnam observed that

Difficulty derivative relative to time

Behavior of s/w development

If project scale is increased, the development time also increase to such an extent that $\frac{k}{t_d^3}$ remains constant around a value which could be 8,15,27.

Software Project Planning

It is represented by D_0 and can be expressed as:

$$D_0 = \frac{k}{t_d^3} \, person / year^2$$

D_0 =8, new s/w with many interfaces & interactions with other systems.

D_0 =15, New standalone system.

D_0 =27, The software is rebuild form existing software.

Example: 4.14

Consider the example 4.13 and calculate the difficulty and manpower build up.

Solution

We know

Difficulty $$D = \frac{K}{t_d^2}$$

$$= \frac{600}{(3.5)^2} = 49 \ person/year$$

Manpower build up can be calculated by following equation

$$D_0 = \frac{K}{t_d^3}$$

$$= \frac{600}{(3.5)^3} = 14 \ person/year^2$$

Productivity Versus Difficulty

Productivity = No. of LOC developed per person-month

$$P \propto D^{\beta}$$

Avg. productivity

$$P = \frac{LOC\ produced}{cumulative\ manpower\ used\ to\ produce\ code}$$

Software Project Planning

P = S/E

$$P = \phi D^{-2/3}$$

$$S = \phi D^{-2/3} E$$

$$= \phi D^{-2/3} (0.3935\ K)$$

$$S = \phi \left[\frac{k}{t_d^2} \right]^{-\frac{2}{3}} k(0.3935)$$

$$S = 0.3935 \phi K^{1/3} t_d^{\ 4/3}$$

Software Project Planning

Software Project Planning

C → 610 – 57314

K : P-Y

T : Years

$$S = CK^{1/3} t_d^{4/3}$$

$$C = S.K^{-1/3} t_d^{-4/3}$$

The trade off of time versus cost

$$K^{1/3} t_d^{4/3} = S / C$$

$$K = \frac{1}{t_d^4} \left(\frac{S}{C} \right)^3$$

Software Project Planning

C = 5000

S = 5,00,000 LOC

$$K = \frac{1}{t_d^4}(100)^3$$

t_d (years)	K (P-Y)
5.0	1600
4.0	3906
3.5	6664
3.0	12346

Table 20: (Manpower versus development time)

Development Subcycle

All that has been discussed so far is related to project life cycle as represented by project curve

Fig.8: Project life cycle

Software Engineering (3rd ed.), By K.K Aggarwal & Yogesh Singh, Copyright © New Age International Publishers, 2007

Project life cycle

Project curve is the addition of two curves

Development Curve

Test & Validation Curve

Software Project Planning

$$\therefore\ m_d(t) = 2k_d bt\, e^{-bt^2}$$

$$y_d(t) = K_d\,[1-e^{-bt^2}]$$

An examination of $m_d(t)$ function shows a non-zero value of m_d at time t_d.

This is because the manpower involved in design & coding is still completing this activity after t^d in form of rework due to the validation of the product.

Nevertheless, for the model, a level of completion has to be assumed for development.

It is assumed that 95% of the development will be completed by the time t_d.

Software Project Planning

$$\frac{y_d(t)}{K_d} = 1 - e^{-bt^2} = 0.95$$

$\therefore$ We may say that $b = \dfrac{1}{2t_{od}^2}$

T_{od}: time at which development curve exhibits a peak manning.

$$t_{od} = \frac{t_d}{\sqrt{6}}$$

Software Project Planning

Relationship between K_d & K must be established.

At the time of origin, both cycles have the same slope.

$$\left(\frac{dm}{dt}\right)_o = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2} = \left(\frac{dm_d}{dt}\right)_o$$

$K_d = K/6$

$$D = \frac{K}{t_d^2} = \frac{K_d}{t_{od}^2}$$

Software Project Planning

This does not apply to the manpower build up D_0.

$$D_o = \frac{K}{t_d^3} = \frac{K_d}{\sqrt{6}t_{od}^3}$$

Conte investigated that

Larger projects → reasonable

Medium & small projects → overestimate

Example: 4.15

A software development requires 90 PY during the total development sub-cycle. The development time is planned for a duration of 3 years and 5 months

(a) Calculate the manpower cost expended until development time

(b) Determine the development peak time

(c) Calculate the difficulty and manpower build up.

Solution

(a) Duration t_d = 3.41 years

We know from equation $$\frac{y_d(t)}{K_d} = 1 - e^{-bt_d} = 0.95$$

$$\frac{y_d(t_d)}{K_d} = 0.95$$

$$Y_d(t_d) = 0.95 \times 90$$

= 85.5 PY

(b) We know from equation $t_{od} = \frac{t_d}{\sqrt{6}}$

$$t_{od} = \frac{t_d}{\sqrt{6}} = 3.41 / 2.449 = 1.39 \ years$$

$$\cong 17 \ months$$

Software Project Planning

(c) Total Manpower development

$$K_d = y_d(t_d)/0.95$$

= 85.5 / 0.95 = 90

$$K = 6K_d = 90 \times 6 = 540PY$$

$$D = K/t_d^2 = 540/(3.41)^2 = 46 \text{ persons/years}$$

$$D_o = \frac{K}{t_d^3} = 540/(3.41)^3 = 13.6 \text{ persons/years}^2$$

Example:4.16

A software development for avionics has consumed 32 PY up to development cycle and produced a size of 48000 LOC. The development of project was completed in 25 months. Calculate the development time, total manpower requirement, development peak time, difficulty, manpower build up and technology factor.

Solution:

Development time t_d = 25 months = 2.08 years

Total manpower development $k_d = \frac{Y_d(t_d)}{0.95} = \frac{32}{0.95} = 33.7\ PY$

Development peak time $t_{od} = \frac{(t_d)}{\sqrt{6}} = 0.85\ years = 10\ months$

$$K = 6K_d = 6 \text{ x } 33.7 = 202 \text{ PY}$$

$$D = \frac{k}{t_d^2} = \frac{202}{(2.08)^2} = 46.7\ pesons / years$$

Software Project Planning

$$D_0 = \frac{k}{t_d^3} = \frac{202}{(2.08)^3} = 22.5 \; Persons / year^2$$

Technology factor

$$C = SK^{-1/3} t_d^{-4/3}$$

$$= 3077$$

Example 4.17

What amount of software can be delivered in 1 year 10 months in an organization whose technology factor is 2400 if a total of 25 PY is permitted for development effort.

Solution:

t_d = 1.8 years

K_d = 25 PY

K = 25 x 6 = 150 PY

C = 2400

We know $S = CK^{1/3} t_d^{4/3}$

= 2400 x 5.313 x 2.18 = 27920 LOC

Example 4.18

The software development organization developing real time software has been assessed at technology factor of 2200. The maximum value of manpower build up for this type of software is D_o=7.5. The estimated size to be developed is S=55000 LOC.

(a) Determine the total development time, the total development manpower cost, the difficulty and the development peak manning.

(b) The development time determined in (a) is considered too long. It is recommended that it be reduced by two months. What would happen?

Solution

We have $$S = CK^{1/3} t_d^{\ 4/3}$$

$$\left(\frac{s}{c}\right)^3 = kt_d^4$$

which is also equivalent to $$\left(\frac{S}{C}\right)^3 = D_o t_d^7$$

then $$t_d = \left[\frac{1}{D_0}\left(\frac{S}{C}\right)^3\right]^{1/7}$$

Software Project Planning

$$Since \quad \frac{S}{C} = 25$$

$t_d = 3$ years

$$K = D_0 t_d^3 = 7.5 \times 27 = 202\ PY$$

Total development manpower cost $K_d = \frac{202}{06} = 33.75 PY$

$D = D_0 t_d = 22.5$ persons / year

$$t_{od} = \frac{t_d}{\sqrt{6}} = \frac{3}{\sqrt{6}} = 1.2\ years$$

Software Project Planning

$$M_d(t) = 2k_d\, bte^{-bt^2}$$

$$Y_d(t) = k_d\,(1-e^{-bt^2})$$

Here $t = t_{od}$

Peak manning $= m_{od} = Dt_{od}e^{-1/2}$

$= 22.5 \times 1.2 \times .606 = 16$ persons

Software Project Planning

III. If development time is reduced by 2 months

Developing s/w at higher manpower build-up

Producing less software

Software Project Planning

(i) Increase Manpower Build-up

$$D_o = \frac{1}{t_d^7}\left(\frac{S}{C}\right)^3$$

Now t_d = 3 years – 2 months = 2.8 years

$$D_o = (25)^3 / (2.8)^7 = 11.6 \; persons / years$$

$$k = D_0 t_d^3 = 254 \, PY$$

$$K_d = \frac{254}{6} = 42.4 \, PY$$

Software Project Planning

$D = D_0 t_d = 32.5$ persons / year

The peak time is $t_{od} = 1.14$ years

Peak manning $m_{od} = Dt_{od}\, e^{-0.5}$

$= 32.5 \times 1.14 \times 0.6$

$= 22$ persons

Note the huge increase in peak manning & manpower cost.

(ii) Produce Less Software

$$\left(\frac{S}{C}\right)^3 = D_0 t_d^7 = 7.5 \times (2.8)^7 = 10119.696$$

$$\left(\frac{S}{C}\right)^3 = 21.62989$$

Then for C=2200

S=47586 LOC

Productivity versus difficult

Example 4.19

A stand alone project for which the size is estimated at 12500 LOC is to be developed in an environment such that the technology factor is 1200. Choosing a manpower build up D_o=15, Calculate the minimum development time, total development man power cost, the difficulty, the peak manning, the development peak time, and the development productivity.

Solution

Size (S) = 12500 LOC

Technology factor (C) = 1200

Manpower buildup (D_o) = 15

Now

$$S = CK^{1/3} t_d^{4/3}$$

$$\frac{S}{C} = K^{1/3} t_d^{4/3}$$

$$\left(\frac{S}{C}\right)^3 = K t_d^4$$

Software Project Planning

Also we know $D_o = \frac{K}{t_d^3}$

$$K = D_o t_d^3 = D_o t_d^3$$

Hence $\left(\frac{S}{C}\right)^3 = D_o t_d^7$

Substituting the values, we get $\left(\frac{12500}{1200}\right)^3 = 15 t_d^7$

$$t_d = \left[\frac{(10.416)^3}{15}\right]^{1/7}$$

$$t_d = 1.85 \text{ years}$$

Software Project Planning

(i) Hence Minimum development time (t_d)=1.85 years

(ii) Total development manpower cost $K_d = \frac{K}{6}$

Hence $K = 15 t_d^3$

$$= 15(1.85)^3 = 94.97 \text{ PY}$$

$$K_d = \frac{K}{6} = \frac{94.97}{6} = 15.83\ PY$$

(iii) Difficulty $D = \frac{K}{t_d^2} = \frac{94.97}{(1.85)^2} = 27.75\ Persons/year$

Software Project Planning

(iv) Peak Manning $$m_0 = \frac{K}{t_d \sqrt{e}}$$

$$= \frac{94.97}{1.85 \times 1.648} = 31.15 Person$$

(v) Development Peak time $$t_{od} = \frac{t_d}{\sqrt{6}}$$

$$= \frac{1.85}{2.449} = 0.755 \ years$$

(vi) Development Productivity

$$= \frac{No.of\ lines\ of\ code\ (S)}{effort\ (K_d)}$$

$$= \frac{12500}{15.83} = 789.6\ LOC/PY$$

Software Risk Management

- We Software developers are extremely optimists.
- We assume, everything will go exactly as planned.
- Other view
  - not possible to predict what is going to happen ?

  Software surprises
  - Never good news

Software Project Planning

Risk management is required to reduce this surprise factor

Dealing with concern before it becomes a crisis.

Quantify probability of failure & consequences of failure.

What is risk ?

Tomorrow's problems are today's risks.

"Risk is a problem that may cause some loss or threaten the success of the project, but which has not happened yet".

Software Project Planning

Risk management is the process of identifying addressing and eliminating these problems before they can damage the project.

Current problems &

Potential Problems

Typical Software Risk

Capers Jones has identified the top five risk factors that threaten projects in different applications.

1. Dependencies on outside agencies or factors.
   - Availability of trained, experienced persons
   - Inter group dependencies
   - Customer-Furnished items or information
   - Internal & external subcontractor relationships

Software Project Planning

2. Requirement issues

Uncertain requirements

Wrong product

or

Right product badly

Either situation results in unpleasant surprises and unhappy customers.

Software Project Planning

- Lack of clear product vision
- Lack of agreement on product requirements
- Unprioritized requirements
- New market with uncertain needs
- Rapidly changing requirements
- Inadequate Impact analysis of requirements changes

Software Project Planning

3. Management Issues

Project managers usually write the risk management plans, and most people do not wish to air their weaknesses in public.

- Inadequate planning
- Inadequate visibility into actual project status
- Unclear project ownership and decision making
- Staff personality conflicts
- Unrealistic expectation
- Poor communication

Software Project Planning

4. Lack of knowledge

- Inadequate training
- Poor understanding of methods, tools, and techniques
- Inadequate application domain experience
- New Technologies
- Ineffective, poorly documented or neglected processes

Software Project Planning

5. Other risk categories

- Unavailability of adequate testing facilities
- Turnover of essential personnel
- Unachievable performance requirements
- Technical approaches that may not work

Risk Management Activities

Fig. 9: Risk Management Activities

Risk Assessment

Identification of risks

Risk analysis involves examining how project outcomes might change with modification of risk input variables.

Risk prioritization focus for severe risks.

Risk exposure: It is the product of the probability of incurring a loss due to the risk and the potential magnitude of that loss.

Software Project Planning

Another way of handling risk is the risk avoidance. Do not do the risky things! We may avoid risks by not undertaking certain projects, or by relying on proven rather than cutting edge technologies.

Risk Control

Risk Management Planning produces a plan for dealing with each significant risks.

- Record decision in the plan.

Risk resolution is the execution of the plans of dealing with each risk.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

4.1 After the finalization of SRS, we may like to estimate

(a) Size (b) Cost

(c) Development time (d) All of the above.

4.2 Which one is not a size measure for software

(a) LOC (b) Function Count

(c) Cyclomatic Complexity (d) Halstead's program length

4.3 Function count method was developed by

(a) B.Beizer (b) B.Boehm

(c) M.halstead (d) Alan Albrecht

4.4 Function point analysis (FPA) method decomposes the system into functional units. The total number of functional units are

(a) 2 (b) 5

(c) 4 (d) 1

Multiple Choice Questions

4.5 IFPUG stand for
(a) Initial function point uniform group
(b) International function point uniform group
(c) International function point user group
(d) Initial function point user group

4.6 Function point can be calculated by
(a) UFP * CAF
(b) UFP * FAC
(c) UFP * Cost
(d) UFP * Productivity

4.7 Putnam resource allocation model is based on
(a) Function points
(b) Norden/ Rayleigh curve
(c) Putnam theory of software management
(d) Boehm's observation on manpower utilisation rate

4.8 Manpower buildup for Putnam resource allocation model is

$(a)\, K / t_d^2 \ persons / year^2$
$(b)\, K / t_d^3 \ persons / year^2$
$(c)\, K / t_d^2 \ persons / year$
$(d)\, K / t_d^3 \ persons / year$

Multiple Choice Questions

4.9 COCOMO was developed initially by

(a) B.W.Bohem (b) Gregg Rothermal

(c) B.Beizer (d) Rajiv Gupta

4.10 A COCOMO model is

(a) Common Cost estimation model

(b) Constructive cost Estimation model

(c) Complete cost estimation model

(d) Comprehensive Cost estimation model

4.11 Estimation of software development effort for organic software is COCOMO is

(a) $E=2.4(KLOC)^{1.05}PM$ (b) $E=3.4(KLOC)^{1.06}PM$

(c) $E=2.0(KLOC)^{1.05}PM$ (d) $E-2.4(KLOC)^{1.07}PM$

4.12 Estimation of size for a project is dependent on

(a) Cost (b) Schedule

(c) Time (d) None of the above

4.13 In function point analysis, number of Complexity adjustment factor are

(a) 10 (b) 20

(c) 14 (d) 12

Multiple Choice Questions

4.14 COCOMO-II estimation model is based on

(a) Complex approach
(b) Algorithm approach
(c) Bottom up approach
(d) Top down approach

4.15 Cost estimation for a project may include

(a) Software Cost
(b) Hardware Cost
(c) Personnel Costs
(d) All of the above

4.16 In COCOMO model, if project size is typically 2-50 KLOC, then which mode is to be selected?

(a) Organic
(b) Semidetached
(c) Embedded
(d) None of the above

4.17 COCOMO-II was developed at

(a) University of Maryland
(b) University of Southern California
(c) IBM
(d) AT & T Bell labs

4.18 Which one is not a Category of COCOMO-II

(a) End User Programming
(b) Infrastructure Sector
(c) Requirement Sector
(d) System Integration

Multiple Choice Questions

4.19 Which one is not an infrastructure software?

(a) Operating system
(b) Database management system
(c) Compilers
(d) Result management system

4.20 How many stages are in COCOMO-II?

(a) 2
(b) 3
(c) 4
(d) 5

4.21 Which one is not a stage of COCOMO-II?

(a) Application Composition estimation model
(b) Early design estimation model
(c) Post architecture estimation model
(d) Comprehensive cost estimation model

4.22 In Putnam resource allocation model, Rayleigh curve is modeled by the equation

$(a)\ \ m(t) = 2at\, e^{-at^2}$

$(b)\ \ m(t) = 2Kt\, e^{-at^2}$

$(c)\ \ m(t) = 2Kat\, e^{-at^2}$

$(d)\ \ m(t) = 2Kbt\, e^{-at^2}$

Multiple Choice Questions

4.23 In Putnam resource allocation model, technology factor 'C' is defined as

$(a)\ C = SK^{-1/3}t_d^{-4/3}$ $(b)\ C = SK^{1/3}t_d^{4/3}$

$(c)\ C = SK^{1/3}t_d^{-4/3}$ $(d)\ C = SK^{-1/3}t_d^{4/3}$

4.24 Risk management activities are divided in

(a) 3 Categories (b) 2 Categories

(c) 5 Categories (d) 10 Categories

4.25 Which one is not a risk management activity?

(a) Risk assessment (b) Risk control

(c) Risk generation (d) None of the above

Exercises

4.1 What are various activities during software project planning?

4.2 Describe any two software size estimation techniques.

4.3 A proposal is made to count the size of 'C' programs by number of semicolons, except those occurring with literal strings. Discuss the strengths and weaknesses to this size measure when compared with the lines of code count.

4.4 Design a LOC counter for counting LOC automatically. Is it language dependent? What are the limitations of such a counter?

4.5 Compute the function point value for a project with the following information domain characteristics.

Number of user inputs = 30

Number of user outputs = 42

Number of user enquiries = 08

Number of files = 07

Number of external interfaces = 6

Assume that all complexity adjustment values are moderate.

Exercises

4.6 Explain the concept of function points. Why FPs are becoming acceptable in industry?

4.7 What are the size metrics? How is function point metric advantageous over LOC metric? Explain.

4.8 Is it possible to estimate software size before coding? Justify your answer with suitable example.

4.9 Describe the Albrecht's function count method with a suitable example.

4.10 Compute the function point FP for a payroll program that reads a file of employee and a file of information for the current month and prints cheque for all the employees. The program is capable of handling an interactive command to print an individually requested cheque immediately.

Exercises

4.11 Assume that the previous payroll program is expected to read a file containing information about all the cheques that have been printed. The file is supposed to be printed and also used by the program next time it is run, to produce a report that compares payroll expenses of the current month with those of the previous month. Compute functions points for this program. Justify the difference between the function points of this program and previous one by considering how the complexity of the program is affected by adding the requirement of interfacing with another application (in this case, itself).

4.12 Explain the Walson & Felix model and compare with the SEL model.

4.13 The size of a software product to be developed has been estimated to be 22000 LOC. Predict the manpower cost (effort) by Walston-Felix Model and SEL model.

4.14 A database system is to be developed. The effort has been estimated to be 100 Persons-Months. Calculate the number of lines of code and productivity in LOC/Person-Month.

Exercises

4.15 Discuss various types of COCOMO mode. Explain the phase wise distribution of effort.

4.16 Explain all the levels of COCOMO model. Assume that the size of an organic software product has been estimated to be 32,000 lines of code. Determine the effort required to developed the software product and the nominal development time.

4.17 Using the basic COCOMO model, under all three operating modes, determine the performance relation for the ratio of delivered source code lines per person-month of effort. Determine the reasonableness of this relation for several types of software projects.

4.18 The effort distribution for a 240 KLOC organic mode software development project is: product design 12%, detailed design 24%, code and unit test 36%, integrate and test 28%. How would the following changes, from low to high, affect the phase distribution of effort and the total effort: analyst capability, use of modern programming languages, required reliability, requirements volatility?

Exercises

4.19 Specify, design, and develop a program that implements COCOMO. Using reference as a guide, extend the program so that it can be used as a planning tool.

4.20 Suppose a system for office automation is to be designed. It is clear from requirements that there will be five modules of size 0.5 KLOC, 1.5 KLOC, 2.0 KLOC, 1.0 KLOC and 2.0 KLOC respectively. Complexity, and reliability requirements are high. Programmer's capability and experience is low. All other factors are of nominal rating. Use COCOMO model to determine overall cost and schedule estimates. Also calculate the cost and schedule estimates for different phases.

4.21 Suppose that a project was estimated to be 600 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

4.22 Explain the COCOMO-II in detail. What types of categories of projects are identified?

Exercises

4.23 Discuss the Infrastructure Sector of COCOMO-II.

4.24 Describe various stages of COCOMO-II. Which stage is more popular and why?

4.25 A software project of application generator category with estimated size of 100 KLOC has to be developed. The scale factor (B) has high percedentness, high development flexibility. Other factors are nominal. The cost drivers are high reliability, medium database size, high Personnel capability, high analyst capability. The other cost drivers are nominal. Calculate the effort in Person-Months for the development of the project.

4.26 Explain the Putnam resource allocation model. What are the limitations of this model?

4.27 Describe the trade-off between time versus cost in Putnam resource allocation model.

4.28 Discuss the Putnam resources allocation model. Derive the time and effort equations.

Exercises

4.29 Assuming the Putnam model, with S=100,000 , C=5000, D_o=15, Compute development time t_d and manpower development K_d.

4.30 Obtain software productivity data for two or three software development programs. Use several cost estimating models discussed in this chapter. How to the results compare with actual project results?

4.31 It seems odd that cost and size estimates are developed during software project planning-before detailed software requirements analysis or design has been conducted. Why do we think this is done? Are there circumstances when it should not be done?

4.32 Discuss typical software risks. How staff turnover problem affects software projects?

4.33 What are risk management activities? Is it possible to prioritize the risk?

Exercises

4.34 What is risk exposure? What techniques can be used to control each risk?

4.35 What is risk? Is it economical to do risk management? What is the effect of this activity on the overall cost of the project?

4.36 There are significant risks even in student projects. Analyze a student project and list all the risk.

Software Design

Software Design

- More creative than analysis
- Problem solving activity

WHAT IS DESIGN

'HOW'

Software design document (SDD)

Software Design

Fig. 1 : Design framework

Software Design

Conceptual Design and Technical Design

Fig. 2 : A two part design process

Software Design

Conceptual design answers :

- ✓ Where will the data come from ?
- ✓ What will happen to data in the system?
- ✓ How will the system look to users?
- ✓ What choices will be offered to users?
- ✓ What is the timings of events?
- ✓ How will the reports & screens look like?

Software Design

Technical design describes :

- Hardware configuration
- Software needs
- Communication interfaces
- I/O of the system
- Software architecture
- Network architecture
- Any other thing that translates the requirements in to a solution to the customer's problem.

Software Design

The design needs to be

- Correct & complete
- Understandable
- At the right level
- Maintainable

Software Design

Fig. 3 : The transformation of an informal design to a detailed design.

MODULARITY

There are many definitions of the term module. Range is from :

i. Fortran subroutine

ii. Ada package

iii. Procedures & functions of PASCAL & C

iv. C++ / Java classes

v. Java packages

vi. Work assignment for an individual programmer

Software Design

All these definitions are correct. A modular system consist of well defined manageable units with well defined interfaces among the units.

Software Design

Properties :

i. Well defined subsystem

ii. Well defined purpose

iii. Can be separately compiled and stored in a library.

iv. Module can use other modules

v. Module should be easier to use than to build

vi. Simpler from outside than from the inside.

Software Design

Modularity is the single attribute of software that allows a program to be intellectually manageable.

It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of software product.

Software Design

Fig. 4 : Modularity and software cost

Module Coupling

Coupling is the measure of the degree of interdependence between modules.

(Uncoupled : no dependencies)

(a)

Loosely coupled:
some dependencies

(B)

Highly coupled:
many dependencies

(C)

Fig. 5 : Module coupling

Software Design

This can be achieved as:

- Controlling the number of parameters passed amongst modules.
- Avoid passing undesired data to calling module.
- Maintain parent / child relationship between calling & called modules.
- Pass data, not the control information.

Consider the example of editing a student record in a 'student information system'.

Fig. 6 : Example of coupling

Software Design

Data coupling	Best
Stamp coupling	
Control coupling	
External coupling	
Common coupling	
Content coupling	Worst

Fig. 7 : The types of module coupling

Given two procedures A & B, we can identify number of ways in which they can be coupled.

Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicate by only passing of data. Other than communicating through data, the two modules are independent.

Stamp coupling

Stamp coupling occurs between module A and B when complete data structure is passed from one module to another.

Control coupling

Module A and B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of changes.

Fig. 8 : Example of common coupling

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. In Fig. 9, module B branches into D, even though D is supposed to be under the control of C.

Fig. 9 : Example of content coupling

Module Cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related.

Fig. 10 : Cohesion=Strength of relations within modules

Types of cohesion

- Functional cohesion
- Sequential cohesion
- Procedural cohesion
- Temporal cohesion
- Logical cohesion
- Coincident cohesion

Software Design

Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Fig. 11 : Types of module cohesion

Functional Cohesion

- A and B are part of a single functional task. This is very good reason for them to be contained in the same procedure.

Sequential Cohesion

- Module A outputs some data which forms the input to B. This is the reason for them to be contained in the same procedure.

Procedural Cohesion

- Procedural Cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed.

Temporal Cohesion

- Module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span.

Logical Cohesion

- Logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions.

Coincidental Cohesion

- Coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another.

Relationship between Cohesion & Coupling

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

Fig. 12 : View of cohesion and coupling

STRATEGY OF DESIGN

A good system design strategy is to organize the program modules in such a way that are easy to develop and latter to, change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program. It is important for two reasons:

- First, even pre-existing code, if any, needs to be understood, organized and pieced together.
- Second, it is still common for the project team to have to write some code and produce original programs that support the application logic of the system.

Bottom-Up Design

These modules are collected together in the form of a “library”.

Fig. 13 : Bottom-up tree structure

Top-Down Design

A top down design approach starts by identifying the major modules of the system, decomposing them into their lower level modules and iterating until the desired level of detail is achieved. This is stepwise refinement; starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly.

Hybrid Design

For top-down approach to be effective, some bottom-up approach is essential for the following reasons:

- To permit common sub modules.
- Near the bottom of the hierarchy, where the intuition is simpler, and the need for bottom-up testing is greater, because there are more number of modules at low levels than high levels.
- In the use of pre-written library modules, in particular, reuse of modules.

FUNCTION ORIENTED DESIGN

Function Oriented design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function. Thus, system is designed from a functional viewpoint.

Software Design

Consider the example of scheme interpreter. Top-level function may look like:

```
While (not finished)
{
        Read an expression from the terminal;
        Evaluate the expression;
        Print the value;
}
```

We thus get a fairly natural division of our interpreter into a "read" module, an "evaluate" module and a "print" module. Now we consider the "print" module and is given below:

```
        Print (expression exp)
{
        Switch (exp → type)
        Case integer: /*print an integer*/
        Case real:    /*print a real*/
        Case list:    /*print a list*/
        :::
}
```

Software Design

We continue the refinement of each module until we reach the statement level of our programming language. At that point, we can describe the structure of our program as a tree of refinement as in design top-down structure as shown in fig. 14.

Fig. 14 : Top-down structure

Software Design

If a program is created top-down, the modules become very specialized. As one can easily see in top down design structure, each module is used by at most one other module, its parent. For a module, however, we must require that several other modules as in design reusable structure as shown in fig. 15.

Fig. 15 : Design reusable structure

Design Notations

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. For a function oriented design, the design can be represented graphically or mathematically by the following:

- Data flow diagrams
- Data Dictionaries
- Structure Charts
- Pseudocode

Structure Chart

It partition a system into block boxes. A black box means that functionality is known to the user without the knowledge of internal design.

Fig. 16 : Hierarchical format of a structure chart

Software Design

Fig. 17 : Structure chart notations

Software Design

A structure chart for "update file" is given in fig. 18.

Fig. 18 : Update file

Software Design

A transaction centered structure describes a system that processes a number of different types of transactions. It is illustrated in Fig.19.

Fig. 19 : Transaction-centered structure

Software Design

In the above figure the MAIN module controls the system operation its functions is to:

- invoke the INPUT module to read a transaction;
- determine the kind of transaction and select one of a number of transaction modules to process that transaction, and
- output the results of the processing by calling OUTPUT module.

Pseudocode

Pseudocode notation can be used in both the preliminary and detailed design phases.

Using pseudocode, the designer describes system characteristics using short, concise, English language phrases that are structured by key words such as It-Then-Else, While-Do, and End.

Functional Procedure Layers

- Function are built in layers, Additional notation is used to specify details.
- Level 0
  - Function or procedure name
  - Relationship to other system components (e.g., part of which system, called by which routines, etc.)
  - Brief description of the function purpose.
  - Author, date

Software Design

- Level 1
  - Function Parameters (problem variables, types, purpose, etc.)
  - Global variables (problem variable, type, purpose, sharing information)
  - Routines called by the function
  - Side effects
  - Input/Output Assertions

Software Design

- Level 2
  - Local data structures (variable etc.)
  - Timing constraints
  - Exception handling (conditions, responses, events)
  - Any other limitations
- Level 3
  - Body (structured chart, English pseudo code, decision tables, flow charts, etc.)

Software Design

IEEE Recommended practice for software design descriptions (IEEE STD 1016-1998)

- Scope

An SDD is a representation of a software system that is used as a medium for communicating software design information.

- References

i. IEEE std 830-1998, IEEE recommended practice for software requirements specifications.

ii. IEEE std 610.12-1990, IEEE glossary of software engineering terminology.

Software Design

- Definitions

i. **Design entity.** An element (Component) of a design that is structurally and functionally distinct from other elements and that is separately named and referenced.

ii. **Design View.** A subset of design entity attribute information that is specifically suited to the needs of a software project activity.

iii. **Entity attributes.** A named property or characteristics of a design entity. It provides a statement of fact about the entity.

iv. **Software design description (SDD).** A representation of a software system created to facilitate analysis, planning, implementation and decision making.

Software Design

- Purpose of an SDD

The SDD shows how the software system will be structured to satisfy the requirements identified in the SRS. It is basically the translation of requirements into a description of the software structure, software components, interfaces, and data necessary for the implementation phase. Hence, SDD becomes the blue print for the implementation activity.

- Design Description Information Content
  - Introduction
  - Design entities
  - Design entity attributes

Software Design

The attributes and associated information items are defined in the following subsections:

a) Identification

b) Type

c) Purpose

d) Function

e) Subordinates

f) Dependencies

g) Interface

h) Resources

i) Processing

j) Data

Software Design

- Design Description Organization

Each design description writer may have a different view of what are considered the essential aspects of a software design. The organization of SDD is given in table 1. This is one of the possible ways to organize and format the SDD.

A recommended organization of the SDD into separate design views to facilitate information access and assimilation is given in table 2.

Software Design

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions and acronyms
2. References
3. Decomposition description
   - 3.1 Module decomposition
     - 3.1.1 Module 1 description
     - 3.1.2 Module 2 description
   - 3.2 Concurrent Process decompostion
     - 3.2.1 Process 1 description
     - 3.2.2 Process 2 description
   - 3.3 Data decomposition
     - 3.3.1 Data entity 1 description
     - 3.3.2 Data entity 2 description

Cont…

Software Design

4. Dependency description
   - 4.1 Intermodule dependencies
   - 4.2 Interprocess dependencies
   - 4.3 Data dependencies
5. Interface description
   - 5.1 Module Interface
     - 5.1.1 Module 1 description
     - 5.1.2 Module 2 description
   - 5.2 Process interface
     - 5.2.1 Process 1 description
     - 5.2.2 Process 2 description
6. Detailed design
   - 6.1 Module detailed design
     - 6.1.1 Module 1 detail
     - 6.1.2 Module 2 detail
   - 6.2 Data detailed design
     - 6.2.1 Data entry 1 detail
     - 6.2.2 Data entry 2 detail

Table 1: Organization of SDD

Software Design

Design View	Scope	Entity attribute	Example representation
Decomposition description	Partition of the system into design entities	Identification, type purpose, function, subordinate	Hierarchical decomposition diagram, natural language
Dependency description	Description of relationships among entities of system resources	Identification, type, purpose, dependencies, resources	Structure chart, data flow diagrams, transaction diagrams
Interface description	List of everything a designer, developer, tester needs to know to use design entities that make up the system	Identification, function, interfaces	Interface files, parameter tables
Detail description	Description of the internal design details of an entity	Identification, processing, data	Flow charts, PDL etc.

Table 2: Design views

Object Oriented Design

Object oriented design is the result of focusing attention not on the function performed by the program, but instead on the data that are to do manipulated by the program. Thus, it is orthogonal to function oriented design.

Object Oriented Design begins with an examination of the real world “things” that are part of the problem to be solved. These things (which we will call objects) are characterized individually in terms of their attributes and behavior.

Software Design

- Basic Concepts

Object Oriented Design is not dependent on any specific implementation language. Problems are modeled using objects. Objects have:

- Behavior (they do things)
- State (which changes when they do things)

Software Design

The various terms related to object design are:

i. Objects

The word "Object" is used very frequently and conveys different meaning in different circumstances. Here, meaning is an entity able to save a state (information) and which offers a number of operations (behavior) to either examine or affect this state. An object is characterized by number of operations and a state which remembers the effect of these operations.

ii. Messages

Objects communicate by message passing. Messages consist of the identity of the target object, the name of the requested operation and any other operation needed to perform the function. Message are often implemented as procedure or function calls.

iii. Abstraction

In object oriented design, complexity is managed using abstraction. Abstraction is the elimination of the irrelevant and the amplification of the essentials.

iv. Class

In any system, there shall be number of objects. Some of the objects may have common characteristics and we can group the objects according to these characteristics. This type of grouping is known as a class. Hence, a class is a set of objects that share a common structure and a common behavior.

We may define a class "car" and each object that represent a car becomes an instance of this class. In this class "car", Indica, Santro, Maruti, Indigo are instances of this class as shown in fig. 20.

Classes are useful because they act as a blueprint for objects. If we want a new square we may use the square class and simply fill in the particular details (i.e. colour and position) fig. 21 shows how can we represent the square class.

Software Design

Fig.20: Indica, Santro, Maruti, Indigo are all instances of the class "car"

Fig. 21: The square class

v. Attributes

An attributes is a data value held by the objects in a class. The square class has two attributes: a colour and array of points. Each attributes has a value for each object instance. The attributes are shown as second part of the class as shown in fig. 21.

vi. Operations

An operation is a function or transformation that may be applied to or by objects in a class. In the square class, we have two operations: set colour() and draw(). All objects in a class share the same operations. An object "knows" its class, and hence the right implementation of the operation. Operation are shown in the third part of the class as indicated in fig. 21.

vii. Inheritance

Imagine that, as well as squares, we have triangle class. Fig. 22 shows the class for a triangle.

Fig. 22: The triangle class

Software Design

Now, comparing fig. 21 and 22, we can see that there is some difference between triangle and squares classes.

For example, at a high level of abstraction, we might want to think of a picture as made up of shapes and to draw the picture, we draw each shape in turn. We want to eliminate the irrelevant details: we do not care that one shape is a square and the other is a triangle as long as both can draw themselves.

To do this, we consider the important parts out of these classes in to a new class called Shape. Fig. 23 shows the results.

Software Design

Fig. 23: Abstracting common features in a new class

This sort of abstraction is called inheritance. The low level classes (known as subclasses or derived classes) inherit state and behavior from this high level class (known as a super class or base class).

viii. Polymorphism

When we abstract just the interface of an operation and leave the implementation to subclasses it is called a polymorphic operation and process is called polymorphism.

ix. Encapsulation (Information Hiding)

Encapsulation is also commonly referred to as “Information Hiding”. It consists of the separation of the external aspects of an object from the internal implementation details of the object.

x. Hierarchy

Hierarchy involves organizing something according to some particular order or rank. It is another mechanism for reducing the complexity of software by being able to treat and express sub-types in a generic way.

Software Design

Fig. 24: Hierarchy

- Steps to Analyze and Design Object Oriented System

There are various steps in the analysis and design of an object oriented system and are given in fig. 25

Software Design

Fig. 25: Steps for analysis & design of object oriented system

Software Design

i. Create use case model

First step is to identify the actors interacting with the system. We should then write the use case and draw the use case diagram.

ii. Draw activity diagram (If required)

Activity Diagram illustrate the dynamic nature of a system by modeling the flow of control form activity to activity. An activity represents an operation on some class in the system that results in a change in the state of the system. Fig. 26 shows the activity diagram processing an order to deliver some goods.

Fig. 26: Activity diagram

iii. Draw the interaction diagram

An interaction diagram shows an interaction, consisting of a set of objects and their relationship, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system.

Steps to draws interaction diagrams are as under:

a) Firstly, we should identify that the objects with respects to every use case.

b) We draw the sequence diagrams for every use case.

d) We draw the collaboration diagrams for every use case.

The object types used in this analysis model are entity objects, interface objects and control objects as given in fig. 27.

Fig. 27: Object types

iv. Draw the class diagram

The class diagram shows the relationship amongst classes. There are four types of relationships in class diagrams.

a) **Association** are semantic connection between classes. When an association connects two classes, each class can send messages to the other in a sequence or a collaboration diagram. Associations can be bi-directional or unidirectional.

b) **Dependencies** connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.

c) **Aggregations** are stronger form of association. An aggregation is a relationship between a whole and its parts.

d) **Generalizations** are used to show an inheritance relationship between two classes.

v. Design of state chart diagrams

A state chart diagram is used to show the state space of a given class, the event that cause a transition from one state to another, and the action that result from a state change. A state transition diagram for a "book" in the library system is given in fig. 28.

Fig. 28: Transition chart for "book" in a library system.

Software Design

vi. Draw component and development diagram

Component diagrams address the static implementation view of a system they are related to class diagrams in that a component typically maps to one or more classes, interfaces or collaboration.

Deployment Diagram Captures relationship between physical components and the hardware.

Software Design

A software has to be developed for automating the manual library of a University. The system should be stand alone in nature. It should be designed to provide functionality's as explained below:

Issue of Books:

- A student of any course should be able to get books issued.
- Books from General Section are issued to all but Book bank books are issued only for their respective courses.
- A limitation is imposed on the number of books a student can issue.
- A maximum of 4 books from Book bank and 3 books from General section is issued for 15 days only.The software takes the current system date as the date of issue and calculates date of return.

Software Design

- A bar code detector is used to save the student as well as book information.
- The due date for return of the book is stamped on the book.

Return of Books:

- Any person can return the issued books.
- The student information is displayed using the bar code detector.
- The system displays the student details on whose name the books were issued as well as the date of issue and return of the book.
- The system operator verifies the duration for the issue.
- The information is saved and the corresponding updating take place in the database.

Software Design

Query Processing:

- The system should be able to provide information like:
- Availability of a particular book.
- Availability of book of any particular author.
- Number of copies available of the desired book.

The system should also be able to generate reports regarding the details of the books available in the library at any given time. The corresponding printouts for each entry (issue/return) made in the system should be generated. Security provisions like the 'login authenticity should be provided. Each user should have a user id and a password. Record of the users of the system should be kept in the log file. Provision should be made for full backup of the system.

Software Design

Use case diagram for library management system

Software Design

Sequence diagram—Login

Software Design

Sequence diagram—issue book

Software Design

Sequence diagram—return book

Software Design

Sequence diagram—query book

Software Design

Sequence diagram—maintain catalog

Software Design

Sequence diagram—generate reports

Software Design

Sequence diagram—maintain login

Software Design

Sequence diagram—maintain student details

Software Design

Class diagram of entity classes

Class diagram of entity classes

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

5.1 The most desirable form of coupling is
(a) Control Coupling (b) Data Coupling
(c) Common Coupling (d) Content Coupling

5.2 The worst type of coupling is
(a) Content coupling (b) Common coupling
(c) External coupling (d) Data coupling

5.3 The most desirable form of cohesion is
(a) Logical cohesion (b) Procedural cohesion
(c) Functional cohesion (d) Temporal cohesion

5.4 The worst type of cohesion is
(a) Temporal cohesion (b) Coincidental cohesion
(c) Logical cohesion (d) Sequential cohesion

5.5 Which one is not a strategy for design?
(a) Bottom up design (b) Top down design
(c) Embedded design (d) Hybrid design

Multiple Choice Questions

5.6 Temporal cohesion means

(a) Cohesion between temporary variables
(b) Cohesion between local variable
(c) Cohesion with respect to time
(d) Coincidental cohesion

5.7 Functional cohesion means

(a) Operations are part of single functional task and are placed in same procedures
(b) Operations are part of single functional task and are placed in multiple procedures
(c) Operations are part of multiple tasks
(d) None of the above

5.8 When two modules refer to the same global data area, they are related as

(a) External coupled
(b) Data coupled
(c) Content coupled
(d) Common coupled

5.9 The module in which instructions are related through flow of control is

(a) Temporal cohesion
(b) Logical cohesion
(c) Procedural cohesion
(d) Functional cohesion

Multiple Choice Questions

5.10 The relationship of data elements in a module is called

(a) Coupling (b) Cohesion
(c) Modularity (d) None of the above

5.11 A system that does not interact with external environment is called

(a) Closed system (b) Logical system
(c) Open system (d) Hierarchal system

5.12 The extent to which different modules are dependent upon each other is called

(a) Coupling (b) Cohesion
(c) Modularity (d) Stability

Exercises

5.1 What is design? Describe the difference between conceptual design and technical design.

5.2 Discuss the objectives of software design. How do we transform an informal design to a detailed design?

5.3 Do we design software when we "write" a program? What makes software design different from coding?

5.4 What is modularity? List the important properties of a modular system.

5.5 Define module coupling and explain different types of coupling.

5.6 Define module cohesion and explain different types of cohesion.

5.7 Discuss the objectives of modular software design. What are the effects of module coupling and cohesion?

5.8 If a module has logical cohesion, what kind of coupling is this module likely to have with others?

5.9 What problems are likely to arise if two modules have high coupling?

Exercises

5.10 What problems are likely to arise if a module has low cohesion?

5.11 Describe the various strategies of design. Which design strategy is most popular and practical?

5.12 If some existing modules are to be re-used in building a new system, which design strategy is used and why?

5.13 What is the difference between a flow chart and a structure chart?

5.14 Explain why it is important to use different notations to describe software designs.

5.15 List a few well-established function oriented software design techniques.

5.16 Define the following terms: Objects, Message, Abstraction, Class, Inheritance and Polymorphism.

5.17 What is the relationship between abstract data types and classes?

Exercises

5.18 Can we have inheritance without polymorphism? Explain.

5.19 Discuss the reasons for improvement using object-oriented design.

5.20 Explain the design guidelines that can be used to produce "good quality" classes or reusable classes.

5.21 List the points of a simplified design process.

5.22 Discuss the differences between object oriented and function oriented design.

5.23 What documents should be produced on completion of the design phase?

5.24 Can a system ever be completely "decoupled"? That is, can the degree of coupling be reduced so much that there is no coupling between modules?

Software Metrics

Software Metrics

Software Metrics: What and Why ?

1. How to measure the size of a software?
2. How much will it cost to develop a software?
3. How many bugs can we expect?
4. When can we stop testing?
5. When can we release the software?

Software Metrics

6. What is the complexity of a module?
7. What is the module strength and coupling?
8. What is the reliability at the time of release?
9. Which test technique is more effective?
10. Are we testing hard or are we testing smart?
11. Do we have a strong program or a week test suite?

Software Metrics

- Pressman explained as "A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of the product or process".
- Measurement is the act of determine a measure
- The metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- Fenton defined measurement as " it is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules".

Software Metrics

- Definition

Software metrics can be defined as "*The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products*".

Software Metrics

- Areas of Applications

The most established area of software metrics is cost and size estimation techniques.

The prediction of quality levels for software, often in terms of reliability, is another area where software metrics have an important role to play.

The use of software metrics to provide quantitative checks on software design is also a well established area.

Software Metrics

- Problems During Implementation

  - Statement : Software development is to complex; it cannot be managed like other parts of the organization.

  Management view : Forget it, we will find developers and managers who will manage that development.

  - Statement : I am only six months late with project.

  Management view : Fine, you are only out of a job.

Software Metrics

- Statement : I am only six months late with project.

Management view : Fine, you are only out of a job.

- Statement : But you cannot put reliability constraints in the contract.

Management view : Then we may not get the contract.

Software Metrics

- Categories of Metrics

  i. **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

  ii. **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

    - effort required in the process
    - time to produce the product
    - effectiveness of defect removal during development
    - number of defects found during testing
    - maturity of the process

Software Metrics

ii. **Project metrics:** describe the project characteristics and execution. Examples are :

- number of software developers
- staffing pattern over the life cycle of the software
- cost and schedule
- productivity

Token Count

The size of the vocabulary of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2$$

where

η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

Software Metrics

The length of the program in the terms of the total number of tokens used is

$$N = N_1 + N_2$$

where

N : program length

N_1 : total occurrences of operators

N_2 : total occurrences of operands

Software Metrics

Volume

$$V = N * \log_2 \eta$$

The unit of measurement of volume is the common unit for size "bits". It is the actual size of a program if a uniform binary encoding for the vocabulary is used.

Program Level

$$L = V^* / V$$

The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).

Software Metrics

Program Difficulty

$$D = 1 / L$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

Effort

$$E = V / L = D * V$$

The unit of measurement of E is elementary mental discriminations.

Software Metrics

- Estimated Program Length

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

$$\hat{N} = 14 \log_2 14 + 10 \log_2 10$$

= 53.34 + 33.22 = 86.56

The following alternate expressions have been published to estimate program length.

$$N_J = Log_2(\eta_1!) + \log_2(\eta_2!)$$

Software Metrics

$$N_B = \eta_1 Log_2 \eta_2 + \eta_2 \log_2 \eta_1$$

$$N_c = \eta_1 \sqrt{\eta_1} + \eta_2 \sqrt{\eta_2}$$

$$N_s = (\eta \log_2 \eta) / 2$$

The definitions of unique operators, unique operands, total operators and total operands are not specifically delineated.

Software Metrics

- Counting rules for C language

1. Comments are not considered.
2. The identifier and function declarations are not considered.
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.

Software Metrics

5. Local variables with the same name in different functions are counted as unique operands.

6. Functions calls are considered as operators.

7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.

8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.

Software Metrics

9. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.

10. All the brackets, commas, and terminators are considered as operators.

11. GOTO is counted as an operator and the label is counted as an operand.

12. The unary and binary occurrence of "+" and "-" are dealt separately. Similarly "*" (multiplication operator) are dealt with separately.

Software Metrics

13. In the array variables such as “array-name [index]” “array-name” and “index” are considered as operands and [ ] is considered as operator.

14. In the structure variables such as “struct-name, member-name” or “struct-name -> member-name”, struct-name, member-name are taken as operands and ‘.’, ‘->’ are taken as operators. Some names of member elements in different structure variables are counted as unique operands.

15. All the hash directive are ignored.

Software Metrics

- Potential Volume

$$V^* = (2 + \eta_2^*)\log_2(2 + \eta_2^*)$$

- Estimated Program Level / Difficulty

Halstead offered an alternate formula that estimate the program level.

$$\hat{L} = 2\eta_2 / (\eta_1 N_2)$$

where

$$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

Software Metrics

- Effort and Time

$$E = V / \hat{L} = V * \hat{D}$$

$$= (n_1 N_2 N \log_2 \eta) / 2\eta_2$$

$$T = E / \beta$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing.

Software Metrics

- Language Level

$$\lambda = L \times V^* = L^2 V$$

Using this formula, Halstead and other researchers determined the language level for various languages as shown in Table 1.

Software Metrics

Language	*Language Level* λ	*Variance* σ
PL/1	1.53	0.92
ALGOL	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42
PASCAL	2.54	–
APL	2.42	–
C	0.857	0.445

Table 1: Language levels

Software Metrics

Example- 6.I

Consider the sorting program in Fig. 2 of chapter 4. List out the operators and operands and also calculate the values of software science measures like η, N, V, E, λ *etc*.

Software Metrics

Solution

The list of operators and operands is given in table 2.

Operators	*Occurrences*	*Operands*	*Occurrences*
int	4	SORT	1
( )	5	x	7
,	4	n	3
[ ]	7	i	8
if	2	j	7
<	2	save	3

(Contd.)...

Software Metrics

;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
< =	2	—	—
+ +	2	—	—
return	2	—	—
{ }	3	—	—
$\eta_1 = 14$	$N_1 = 53$	$\eta_2 = 10$	$N_2 = 38$

Table 2: Operators and operands of sorting program of fig. 2 of chapter 4

Software Metrics

Here N_1=53 and N_2=38. The program length $N=N_1+N_2=91$

Vocabulary of the program $\eta = \eta_1 + \eta_2 = 14 + 10 = 24$

Volume $V = N \times \log_2 \eta$

$= 91 \times \log_2 24 = 417$ bits

The estimated program length $\hat{N}$ of the program

$= 14 \log_2 14 + 10 \log_2 10$

$= 14 * 3.81 + 10 * 3.32$

$= 53.34 + 33.2 = 86.45$

Software Metrics

Conceptually unique input and output parameters are represented by η_2^*

$\eta_2^* = 3$ {x: array holding the integer to be sorted. This is used both as input and output}.

{N: the size of the array to be sorted}.

The potential volume $V^* = 5 \log_2 5 = 11.6$

Since $L = V^* / V$

Software Metrics

$$= \frac{11.6}{417} = 0.027$$

D = I / L

$$= \frac{1}{0.027} = 37.03$$

Estimated program level

$$\hat{L} = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2} = \frac{2}{14} \times \frac{10}{38} = 0.038$$

Software Metrics

We may use another formula

$$\hat{V} = V \times \hat{L} = 417 \times 0.038 = 15.67$$

$$\hat{E} = V / \hat{L} = \hat{D} \times V$$

=417 / 0.038 = 10973.68

Therefore, 10974 elementary mental discrimination are required to construct the program.

$$T = E / \beta = \frac{10974}{18} = 610 \text{ seconds} = 10 \text{ minutes}$$

This is probably a reasonable time to produce the program, which is very simple

Software Metrics

```
#include < stdio.h >
#define MAXLINE 100
int getline(char line[],int max);
int strindex(char source[],char search for[]);
char pattern[ ]="ould";
int main()
{
     char line[MAXLINE];
     int found = 0;
     while(getline(line,MAXLINE)>0)
          if(strindex(line, pattern)>=0)
          {
               printf("%s",line);
               found++;
          }
     return found;
}
```

Table 3

(Contd.)...

Software Metrics

```
int getline(char s[],int lim)
{
    int c,i=0;
    while(--lim > 0 && (c=getchar())!= EOF && c!='\n')
        s[i++]=c;
    if(c=='\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
int strindex(char s[],char t[])
{
    int i,j,k;
    for(i=0;s[i] !='\0';i++)
    {
        for(j=i,k=0;t[k] != '\0',s[j] ==t[k];j++,k++);
        if(k>0 && t[k] =='\0')
            return i;
    }
    return -1;
}
```

Table 3

Example- 6.2

Consider the program shown in Table 3. Calculate the various software science metrics.

Software Metrics

Solution

List of operators and operands are given in Table 4.

Operators	*Occurrences*	*Operands*		*Occurrences*
main ()	1		—	—
–	1	**Extern variable**	pattern	1
for	2	**main function**	line	3
= =	3		found	2
! =	4	**getline function**	s	3
getchar	1		lim	1

Table 4

(Contd.)...

Software Metrics

( )	1	*c*	5
&&	3	*i*	4
– –	1	**Strindex function** *s*	2
return	4	*t*	3
+ +	6	*i*	5
printf	1	*j*	3
> =	1	*k*	6
strindex	1	**Numerical Operands** 1	1
If	3	MAXLINE	1
>	3	0	8
getline	1	'\0'	4
while	2	'*n*'	2
{ }	5	**strings "ould"**	1
=	10	—	—
[ ]	9	—	—
,	6	—	—
;	14	—	—
EOF	1	—	—
$\eta_1 = 24$	$N_1 = 84$	$\eta_2 = 18$	$N_2 = 55$

Table 5

Software Metrics

Program vocabulary $\eta = 42$

Program length $N = N_1 + N_2$

$= 84 + 55 = 139$

Estimated length $\hat{N} = 24\log_2 24 + 18\log_2 18 = 185.115$

% error $= 24.91$

Program volume $V = 749.605$ bits

Estimated program level $= \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$

$$= \frac{2}{24} \times \frac{18}{55} = 0.02727$$

Software Metrics

Minimal volume $V^* = 20.4417$

Effort $= V / \hat{L}$

$$= \frac{748.605}{.02727}$$

= 27488.33 elementary mental discriminations.

Time T = $E / \beta = \frac{27488.33}{18}$

= 1527.1295 seconds

= 25.452 minutes

Data Structure Metrics

Program	Data Input	Internal Data	Data Output
Payroll	Name/ Social Security No./ Pay Rate/ Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spreadsheet	Item Names/ Item amounts/ Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size/ No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig.1: Some examples of input, internal, and output data

Software Metrics

- The Amount of Data

One method for determining the amount of data is to count the number of entries in the cross-reference list.

A variable is a string of alphanumeric characters that is defined by a developer and that is used to represent some value during either compilation or execution.

Software Metrics

```
1.  #include < stdio. h >
2.  struct check
3.    {
4.      float gross, tax, net;
5.    } pay;
6.    float hours, rate;
7.    void main ( )
8.      {
9.      while (! feof (stdin))
10.       {
11.         scanf("%f %f", & hours, & rate);
12.         pay. gross = hours * rate;
13.         pay. tax = 0.25 * pay. gross;
14.         pay. net = pay. gross - pay. tax;
15.         printf("%f %f %f/n", pay. gross, pay. tax, pay. net);
16.       }
17.     }
```

Fig.2: Payday program

Software Metrics

check	2				
gross	4	12	13	14	15
hours	6	11	12		
net	4	14	15		
pay	5	12	13	13	14
	14	14	15	15	15
rate	6	11	12		
tax	4	13	14	15	

Fig.3: A cross reference of program payday

feof	9
stdin	10

Fig.4: Some items not counted as VARS

η_2 = VARS + unique constants + labels.

Halstead introduced a metric that he referred to as η_2 to be a count of the operands in a program – including all variables, constants, and labels. Thus,

$$\eta_2 = VARS + \text{unique constants} + \text{labels}$$

Software Metrics

```
1   # include < stdio. h >
2   struct [check]
3   {
4     float [gross], [tax], [net];
5   } [pay];
6   float [hours], [rate);
7   void main ( )
8   {
9   while (! feof (stdin))
10    {
11      scanf("% f % f", & [hour], & [rate]);
12      [pay] . [gross] = [hours] * [rate];
13      [pay] . [tax] = 0.25 * [pay] . [gross];
14      [pay] . [net] = [pay] . [gross] - [pay] . [tax];
15      printf("% f % f % f/n", [pay] . [gross] [pay] . [tax], [pay] . [net]);
16    }
17  }
```

Fig.6: Program payday with operands in brackets

The Usage of Data within a Module

- ✓ Live Variables

Definitions :

1. A variable is live from the beginning of a procedure to the end of the procedure.
2. A variable is live at a particular statement only if it is referenced a certain number of statements before or after that statement.
3. A variable is live from its first to its last references within a procedure.

Software Metrics

```
#include < stdio. h >

void swap (int x [ ], int K)
  {
    int t;
    t = x[K];
    x[K] = x[K + 1];
    x[K + 1] = t;
  }

void main ( )
{
int i, j, last, size, continue, a[100], b[100];
scanf("% d", & size);
for (j = 1; j < = size; j + +)
scanf("%d %d", & a[j], & b[j];
last = size;
continue = 1;
```

```
while(continue)
{
  continue = 0;
  last = last-1;
  i = 1;
while (i < = last)
{
  if (a[i] > a[i + 1])
  {
    continue = 1;
    swap (a, i);
    swap (b, i);
  }
    i = i + 1;
}
}
for (j = 1; j < = size; j ++)
printf("%d %d\n", a[j], b[j]);
}
```

Fig.6: Bubble sort program

Software Metrics

It is thus possible to define the average number of live variables, $\overline{(LV)}$ which is the sum of the count of live variables divided by the count of executable statements in a procedure. This is a complexity measure for data usage in a procedure or program. The live variables in the program in fig. 6 appear in fig. 7 the average live variables for this program is

$$\frac{124}{34} = 3.647$$

Software Metrics

Line	Live Variables	Count
4	----	0
5	----	0
6	t, x, k	3
7	t, x, k	3
8	t, x, k	3
9	----	0
10	----	0
11	----	0
12	----	0
13	----	0
14	size	1
15	size, j	2
16	Size, j, a, b	4

cont…

Software Metrics

Line	Live Variables	Count
17	size, j, a, b, last	5
18	size, j, a, b, last, continue	6
19	size, j, a, b, last, continue	6
20	size, j, a, b, last, continue	6
21	size, j, a, b, last, continue	6
22	size, j, a, b, last, continue	6
23	size, j, a, b, last, continue, i	7
24	size, j, a, b, last, continue, i	7
25	size, j, a, b, continue, i	6
26	size, j, a, b, continue, i	6
27	size, j, a, b, continue, i	6
28	size, j, a, b, continue, i	6
29	size, j, a, b, i	5

cont…

Line	Live Variables	Count
30	size, j, a, b, i	5
31	size, j, a, b, i	5
32	size, j, a, b, i	5
33	size, j, a, b	4
34	size, j, a, b	4
35	size, j, a, b	4
36	j, a, b	3
37	--	0

Fig.7: Live variables for the program in fig.6

Software Metrics

- ✓ Variable spans

```
…
21          scanf ("%d %d, &a, &b)
…
32          x =a;
…
45          y = a – b;
…
53          z = a;
…
60          printf ("%d %d, a, b);
…
```

Fig.: Statements in ac program referring to variables a and b.

The size of a span indicates the number of statements that pass between successive uses of a variables

Software Metrics

- Making program-wide metrics from intra-module metrics

For example if we want to characterize the average number of live variables for a program having modules, we can use this equation.

$$\overline{LV}\,program = \frac{\sum_{i=1}^{m} \overline{LV_i}}{m}$$

where $\overline{(LV)}_i$ is the average live variable metric computed from the *i*th module

The average span size $\overline{(SP)}$ for a program of n spans could be computed by using the equation.

$$\overline{SP}\,program = \frac{\sum_{i=1}^{n} \overline{SP_i}}{n}$$

Software Metrics

- Program Weakness

A program consists of modules. Using the average number of live variables $\overline{(LV)}$ and average life variables (γ), the module weakness has been defined as

$$WM = \overline{LV} * \gamma$$

Software Metrics

A program is normally a combination of various modules, hence program weakness can be a useful measure and is defined as:

$$WP = \frac{\left(\sum_{i=1}^{m} WM_i \right)}{m}$$

where, WM_i : weakness of *i*th module

WP : weakness of the program

m : number of modules in the program

Example- 6.3

Consider a program for sorting and searching. The program sorts an array using selection sort and than search for an element in the sorted array. The program is given in fig. 8. Generate cross reference list for the program and also calculate $\overline{LV}$ and γ, WM for the program.

Solution

The given program is of 66 lines and has 11 variables. The variables are a, I, j, item, min, temp, low, high, mid, loc and option.

```
/*****************************************************************/
/****  PROGRAM TO SORT AN ARRAY USING SELECTION SORT & THEN SEARCH
*******/
/****  FOR AN ELEMENT IN THE SORTED ARRAY ********/
/*****************************************************************/

#include <stdio.h>
#define MAX 10

main ()
{
```

(Contd.)...

Software Metrics

```
int a[MAX];
int i,j,item,min,temp;
int low=0,high,mid,loc;
char option;

for (i=0;i<MAX;i++)
  {
  printf("Enter a[%d]:",i);
  scanf("%d",&a[i]);
  }
/* selection sort */
  for (i=0;i<(MAX-1);i++)
  {
  min=i;
  for (j=i+1; j<MAX;j++)
    {
    if (a[min]>a[j])
```

(Contd.)...

Software Metrics

```
28          {
29          temp=a[min];
30          a[min]=a[j];
31          a[j]=temp;
32         }
33       }
34     }
35  printf("\n The Sorted Array:\n");
36  for (i=0; i<MAX;i++)
37    printf("\n a[%d]=%d",i,a[i]);
38  printf("\n Do you want to search any element in the array
                                                    (Y/N):");
39  fflush(stdin);
40  scanf("%c",&option);
41  if (toupper(option)=='Y')
42     {
43     printf("\n Enter the item to be searched :");
```

(Contd.)...

Software Metrics

```
44      scanf("%d", &item);
45      high=MAX;
46      mid=(int)(low+high)/2;
47      while ((low<=high)&&(item!=a[mid]))
48      {
49        if (item>a[mid])
50        low=mid+1;
51        else high=mid-1;
52        mid=(int) (low+high)/2;
53      }
54      if(low>high)
55      {
56        loc=0;
57        printf("\n No such item is present in the array\n");
58      }
```

(Contd.)...

Software Metrics

```
59      if  (item==a[mid])
60      {
61        loc=mid;
62        printf("\n The item %d is present at location %d in the sorted
          array\n",item,loc);
63      }
64    }
65   printf("\n Sorting & Searching done");
66  }
```

Fig.8: Sorting & searching program

Software Metrics

Cross-Reference list of the program is given below:

a	11	18	19	27	27	29	30	30	31	37	47	49	59		
i	12	16	16	16	18	19	22	22	22	24	36	36	36	37	37
j	12	25	25	25	27	30	31								
item	12	44	47	49	59	62									
min	12	24	27	29	30										
temp	12	29	31												
low	13	46	47	50	52	54									
high	13	45	46	47	51	52	54								
mid	13	46	47	49	50	51	52	59	61						
loc	13	56	61	62											
option	14	40	41												

Live Variables per line are calculated as:

Line	Live Variables	Count
13	low	1
14	low	1
15	low	1
16	low, i	2
17	low, i	2
18	low, i, a	3
19	low, i, a	3
20	low, i, a	3
22	low, i, a	3
23	low, i, a	3
24	low, i, a, min	4
25	low, i, a, min, j	5
26	low, i, a, min, j	5

cont..

Software Metrics

Line	Live Variables	Count
27	low, i, a, min, j	5
28	low, i, a, min, j	5
29	low, i, a, min, j, temp	6
30	low, i, a, min, j, temp	6
31	low, i, a, j, temp	5
32	low, i, a	3
33	low, i, a	3
34	low, i, a	3
35	low, i, a	3
36	low, i, a	3
37	low, i, a	3
38	low, a	2
39	low, a	2

cont..

Software Metrics

Line	Live Variables	Count
40	low, a, option	3
41	low, a, option	3
42	low, a	2
43	low, a	2
44	low, a, item	3
45	low, a, item, high	4
46	low, a, item, high, mid	5
47	low, a, item, high, mid	5
48	low, a, item, high, mid	5
49	low, a, item, high, mid	5
50	low, a, item, high, mid	5
51	low, a, item, high, mid	5
52	low, a, item, high, mid	5

cont..

Software Metrics

Line	Live Variables	Count
53	low, a, item, high, mid	5
54	low, a, item, high, mid	5
55	a, item, mid	3
56	a, item, mid, loc	4
57	a, item, mid, loc	4
58	a, item, mid, loc	4
59	a, item, mid, loc	4
60	item, mid, loc	3
61	item, mid, loc	3
62	item, loc	2

cont...

Software Metrics

Line	Live Variables	Count
63		0
64		0
65		0
66		0
	Total	174

Software Metrics

Average number of live variables $(\overline{LV}) = \dfrac{\text{Sum of count of live variables}}{\text{Count of executable statements}}$

$$LV = \frac{174}{53} = 3.28$$

$$\gamma = \frac{\text{Sum of count of live variables}}{\text{Total number of variables}}$$

$$\gamma = \frac{174}{11} = 15.8$$

$$\text{Module Weakness (WM)} = \overline{\text{LV}} \times \gamma$$

$$WM = 3.28 \times 15.8 = 51.8$$

- The Sharing of Data Among Modules

A program normally contains several modules and share coupling among modules. However, it may be desirable to know the amount of data being shared among the modules.

Fig.10: Three modules from an imaginary program

Software Metrics

Fig.11: ”Pipes” of data shared among the modules

Fig.12: The data shared in program bubble

Information Flow Metrics

Component : Any element identified by decomposing a (software) system into its constituent parts.

Cohesion : The degree to which a component performs a single function.

Coupling : The term used to describe the degree of linkage between one component to others in the same system.

- The Basic Information Flow Model

Information Flow metrics are applied to the Components of a system design. Fig. 13 shows a fragment of such a design, and for component 'A' we can define three measures, but remember that these are the simplest models of IF.

1. 'FAN IN' is simply a count of the number of other Components that can call, or pass control, to Component A.
2. 'FANOUT' is the number of Components that are called by Component A.
3. This is derived from the first two by using the following formula. We will call this measure the INFORMATION FLOW index of Component A, abbreviated as IF(A).

$$IF(A) = [FAN\ IN(A) \times FAN\ OUT\ (A)]^2$$

Software Metrics

Fig.13: Aspects of complexity

Software Metrics

The following is a step-by-step guide to deriving these most simple of IF metrics.

1. Note the level of each Component in the system design.

2. For each Component, count the number of calls so that Component – this is the FAN IN of that Component. Some organizations allow more than one Component at the highest level in the design, so for Components at the highest level which should have a FAN IN of zero, assign a FAN IN of one. Also note that a simple model of FAN IN can penalize reused Components.

3. For each Component, count the number of calls from the Component. For Component that call no other, assign a FAN OUT value of one.

cont…

Software Metrics

4. Calculate the IF value for each Component using the above formula.

5. Sum the IF value for all Components within each level which is called as the LEVEL SUM.

6. Sum the IF values for the total system design which is called the SYSTEM SUM.

7. For each level, rank the Component in that level according to FAN IN, FAN OUT and IF values. Three histograms or line plots should be prepared for each level.

8. Plot the LEVEL SUM values for each level using a histogram or line plot.

Software Metrics

- A More Sophisticated Information Flow Model

a = the number of components that call A.

b = the number of parameters passed to A from components higher in the hierarchy.

c = the number of parameters passed to A from components lower in the hierarchy.

d = the number of data elements read by component A.

Then:

$$\text{FAN IN}(A) = a + b + c + d$$

Software Metrics

Also let:

e = the number of components called by A;

f = the number of parameters passed from A to components higher in the hierarchy;

g = the number of parameters passed from A to components lower in the hierarchy;

h = the number of data elements written to by A.

Then:

$$\text{FAN OUT}(A) = e + f + g + h$$

Object Oriented Metrics

Terminologies

S.No	Term	Meaning/purpose
1	Object	Object is an entity able to save a state (information) and offers a number of operations (behavior) to either examine or affect this state.
2	Message	A request that an object makes of another object to perform an operation.
3	Class	A set of objects that share a common structure and common behavior manifested by a set of methods; the set serves as a template from which object can be created.
4	Method	an operation upon an object, defined as part of the declaration of a class.
5	Attribute	Defines the structural properties of a class and unique within a class.
6	Operation	An action performed by or on an object, available to all instances of class, need not be unique.

Object Oriented Metrics

Terminologies

S.No	Term	Meaning/purpose
7	Instantiation	The process of creating an instance of the object and binding or adding the specific data.
8	Inheritance	A relationship among classes, where in an object in a class acquires characteristics from one or more other classes.
9	Cohesion	The degree to which the methods within a class are related to one another.
10	Coupling	Object A is coupled to object B, if and only if A sends a message to B.

Object Oriented Metrics

- Measuring on class level
  - coupling
  - inheritance
  - methods
  - attributes
  - cohesion
- Measuring on system level

Object Oriented Metrics

Size Metrics:

- Number of Methods per Class (NOM)
- Number of Attributes per Class (NOA)
- Weighted Number Methods in a Class (WMC)
  - Methods implemented within a class or the sum of the complexities of all methods

Object Oriented Metrics

Coupling Metrics:

- Response for a Class (RFC)
  - Number of methods (internal and external) in a class.

- Data Abstraction Coupling(DAC)
  - Number of Abstract Data Types in a class.

- Coupling between Objects (CBO)
  - Number of other classes to which it is coupled.

Object Oriented Metrics

- Message Passing Coupling (MPC)
  - Number of send statements defined in a class.
- Coupling Factor (CF)
  - Ratio of actual number of coupling in the system to the max. possible coupling.

Object Oriented Metrics

Cohesion Metrics:

- LCOM: Lack of cohesion in methods
  - Consider a class C_1 with n methods M_1, M_2…., M_n. Let (I_j) = set of all instance variables used by method M_i. There are n such sets $\{I_1\}$,…….$\{I_n\}$. Let

$$P = \{(I_i, I_j) \mid I_i \cap I_j = 0\} \text{ and } Q = \{((I_i, I_j) \mid I_i \cap I_j \neq 0\}$$

If all n $\{(I_1\}, \ldots\ldots\ldots .(I_n)\}$ sets are 0 then P=0

$$\text{LCOM} = |P| - |Q|, \text{ if } |P| > |Q|$$
$$= 0 \text{ otherwise}$$

Object Oriented Metrics

- Tight Class Cohesion (TCC)

  Percentage of pairs of public methods of the class with common attribute usage.

- Loose Class Cohesion (LCC)
  - Same as TCC except that this metric also consider indirectly connected methods.
- Information based Cohesion (ICH)
  - Number of invocations of other methods of the same class, weighted by the number of parameters of the invoked method.

Object Oriented Metrics

Inheritance Metrics:

- DIT - Depth of inheritance tree
- NOC - Number of children
  - only immediate subclasses are counted.

Object Oriented Metrics

Inheritance Metrics:

- AIF- Attribute Inheritance Factor
  - Ratio of the sum of inherited attributes in all classes of the system to the total number of attributes for all classes.

$$\text{AIF} = \frac{\sum_{i=1}^{TC} A_d(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

$$A_a(C_i) = A_i(C_i) + A_d(C_i)$$

TC= total number of classes

Ad (Ci) = number of attribute declared in a class

Ai (Ci) = number of attribute inherited in a class

Object Oriented Metrics

Inheritance Metrics:

- MIF- Method Inheritance Factor
  - Ratio of the sum of inherited methods in all classes of the system to the total number of methods for all classes.

$$\text{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

$$M_a(C_i) = M_i(C_i) + M_d(C_i)$$

TC= total number of classes

Md(Ci)= the number of methods declared in a class

Mi(Ci)= the number of methods inherited in a class

Use-Case Oriented Metrics

- Counting actors

Type	Description	Factor
Simple	Program interface	1
Average	Interactive or protocol driven interface	2
Complex	Graphical interface	3

Actor weighting factors

- Simple actor: represents another system with a defined interface.
- Average actor: another system that interacts through a text based interface through a protocol such as TCP/IP.
- Complex actor: person interacting through a GUI interface.

The actors weight can be calculated by adding these values together.

Use-Case Oriented Metrics

- Counting use cases

Type	Description	Factor
Simple	3 or fewer transactions	5
Average	4 to 7 transactions	10
Complex	More than 7 transactions	15

Transaction-based weighting factors

The number of each use case type is counted in the software and then each number is multiplied by a weighting factor as shown in table above.

Web Engineering Project Metrics

- Number of static web pages
- Number of dynamic web pages
- Number of internal page links
- Word count
- Web page similarity
- Web page search and retrieval
- Number of static content objects
- Number of dynamic content objects

Metrics Analysis

Statistical Techniques

- Summary statistics such as mean, median, max. and min.
- Graphical representations such as histograms, pie charts and box plots.
- Principal component analysis
- Regression and correlation analysis
- Reliability models for predicting future reliability.

Metrics Analysis

Problems with metric data:

- Normal Distribution
- Outliers
- Measurement Scale
- Multicollinearity

Metrics Analysis

Common pool of data:

- The selection of projects should be representative and not all come from a single application domain or development styles.
- No single very large project should be allowed to dominate the pool.
- For some projects, certain metrics may not have been collected.

Metrics Analysis

Pattern of Successful Applications:

- Any metric is better then none.
- Automation is essential.
- Empiricism is better then theory.
- Use multifactor rather then single metrics.
- Don't confuse productivity metrics with complexity metrics.
- Let them mature.
- Maintain them.
- Let them die.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

6.1 Which one is not a category of software metrics ?

(a) Product metrics (b) Process metrics

(c) Project metrics (d) People metrics

6.2 Software science measures are developed by

(a) M.Halstead (b) B.Littlewood

(c) T.J.McCabe (d) G.Rothermal

6.3 Vocabulary of a program is defined as:

$(a)\eta = \eta_1 + \eta_2$ $(b)\eta = \eta_1 - \eta_2$

$(c)\eta = \eta_1 \times \eta_2$ $(d)\eta = \eta_1 / \eta_2$

6.4 In halstead theory of software science, volume is measured in bits. The bits are

(a) Number of bits required to store the program

(b) Actual size of a program if a uniform binary encoding scheme for vocabulary is used

(c) Number of bits required to execute the program

(d) None of the above

Multiple Choice Questions

6.5 In Halstead theory, effort is measured in

(a) Person-months (b) Hours

(c) Elementary mental discriminations (d) None of the above

6.6 Language level is defined as

$(a)\ \lambda = L^3V$ $(b)\ \lambda = LV$

$(c)\ \lambda = LV\ *$ $(d)\ \lambda = L^2V$

6.7 Program weakness is

$(a) WM = \overline{LV} \times \gamma$ $(b) WM = \overline{LV} / \gamma$

$(a) WM = \overline{LV} + \gamma$ (d) None of the above

6.8 'FAN IN' of a component A is defined as

(a) Count of the number of components that can call, or pass control, to component A

(b) Number of components related to component A

(c) Number of components dependent on component A

(d) None of the above

Multiple Choice Questions

6.9 'FAN OUT' of a component A is defined as

(a) number of components related to component A

(b) number of components dependent on component A

(c) number of components that are called by component A

(d) none of the above

6.10 Which is not a size metric?

(a) LOC (b) Function count

(c) Program length (d) Cyclomatic complexity

6.11 Which one is not a measure of software science theory?

(a) Vocabulary (b) Volume

(c) Level (d) Logic

6.12 A human mind is capable of making how many number of elementary mental discriminations per second (i.e., stroud number)?

(a) 5 to 20 (b) 20 to 40

(c) 1 to 10 (d) 40 to 80

Multiple Choice Questions

6.13 Minimal implementation of any algorithm was given the following name by Halstead:

(a) Volume
(b) Potential volume
(c) Effective volume
(d) None of the above

6.14 Program volume of a software product is

(a) $V=N \log_2 n$
(b) $V=(N/2) \log_2 n$
(c) $V=2N \log_2 n$
(d) $V=N \log_2 n+1$

6.15 Which one is the international standard for size measure?

(a) LOC
(b) Function count
(c) Program length
(d) None of the above

6.16 Which one is not an object oriented metric?

(a) RFC
(b) CBO
(c)DAC
(d) OBC

Multiple Choice Questions

6.17 Which metric also consider indirect connected methods?

(a) TCC
(b) LCC
(c) Both of the above
(d) None of the above

6.18 depth of inheritance tree (DIT) can be measured by:

(a) Number of ancestors classes
(b) Number of successor classes
(c) Number of failure classes
(d) Number of root classes

6.19 A dynamic page is:

(a) where contents are not dependent on the actions of the user
(b) where contents are dependent on the actions of the user
(c) where contents cannot be displayed
(d) None of the above

6.20 Which of the following is not a size metric?

(a) LOC
(b) FP
(c) Cyclomatic Complexity
(d) program length

Exercises

6.1 Define software metrics. Why do we really need metrics in software?

6.2 Discus the areas of applications of software metrics? What are the problems during implementation of metrics in any organizations?

6.3 What are the various categories of software metrics? Discuss with the help of suitable example.

6.4 Explain the Halstead theory of software science. Is it significant in today's scenario of component based software development?

6.5 What is the importance of language level in Halstead theory of software science?

6.6 Give Halstead's software science measure for:

(i) Program Length
(ii) Program volume
(iii) Program level
(iv) Effort
(v) Language level

Exercises

6.7 For a program with number of unique operators $\eta_1 = 20$ and number of unique operands $\eta_2 = 40$, Compute the following:

(i) Program volume (ii) Effort and time

(iii) Program length (iv) Program level

6.8 Develop a small software tool that will perform a Halstead analysis on a programming language source code of your choice.

6.9 Write a program in C and also PASCAL for the calculation of the roots of a quadratic equation, Find out all software science metrics for both the programs. Compare the outcomes and comment on the efficiency and size of both the source codes.

6.10 How should a procedure identifier be considered, both when declared and when called/ What about the identifier of a procedure that is passed as a parameter to another procedure?

Exercises

6.11 Assume that the previous payroll program is expected to read a file containing information about all the cheques that have been printed. The file is supposed to be printed and also used by the program next time it is run, to produce a report that compares payroll expenses of the current month with those of the previous month. Compute functions points for this program. Justify the difference between the function points of this program and previous one by considering how the complexity of the program is affected by adding the requirement of interfacing with another application (in this case, itself).

6.12 Define data structure metrics. How can we calculate amount of data in a program?

6.13 Describe the concept of module weakness. Is it applicable to programs also.

6.14 Write a program for the calculation of roots of a quadratic equation. Generate cross reference list for the program and also calculate for this program.

Exercises

6.15 Show that the value of SP at a particular statement is also the value of LV at that point.

6.16 Discuss the significance of data structure metrics during testing.

6.17 What are information flow metrics? Explain the basic information flow model.

6.18 Discuss the problems with metrics data. Explain two methods for the analysis of such data.

6.19 Show why and how software metrics can improve the software process. Enumerate the effect of metrics on software productivity.

6.20 Why does lines of code (LOC) not measure software nesting and control structures?

6.21 Several researchers in software metrics concentrate on data structure to measure complexity. Is data structure a complexity or quality issue, or both?

Exercises

6.22 List the benefits and disadvantages of using Library routines rather than writing own code.

6.23 Compare software science measure and function points as measure of complexity. Which do you think more useful as a predictor of how much particular software's development will cost?

6.24 Some experimental evidence suggests that the initial size estimate for a project affects the nature and results of the project. Consider two different managers charged with developing the same application. One estimates that the size of the application will be 50,000 lines, while the other estimates that it will be 100,000 lines. Discuss how these estimates affect the project throughout its life cycle.

6.25 Which one is the most appropriate size estimation technique and why?

6.26 Discuss the object oriented metrics. What is the importance of metrics in object oriented software development ?

Exercises

6.27 Define the following: RFC, CBO, DAC, TCC, LCC & DIT.

6.28 What is the significance of use case metrics? Is it really important to design such metrics?

Software Reliability

Software Reliability

Basic Concepts

There are three phases in the life of any hardware component i.e., burn-in, useful life & wear-out.

In **burn-in phase,** failure rate is quite high initially, and it starts decreasing gradually as the time progresses.

During **useful life period**, failure rate is approximately constant.

Failure rate increase in **wear-out phase** due to wearing out/aging of components. The best period is useful life period. The shape of this curve is like a “bath tub” and that is why it is known as bath tub curve. The “bath tub curve” is given in Fig.7.1.

Software Reliability

Fig. 7.1: Bath tub curve of hardware reliability.

Software Reliability

We do not have wear out phase in software. The expected curve for software is given in fig. 7.2.

Fig. 7.2: Software reliability curve (failure rate versus time)

Software Reliability

Software may be retired only if it becomes obsolete. Some of contributing factors are given below:

- ✓ change in environment
- ✓ change in infrastructure/technology
- ✓ major change in requirements
- ✓ increase in complexity
- ✓ extremely difficult to maintain
- ✓ deterioration in structure of the code
- ✓ slow execution speed
- ✓ poor graphical user interfaces

Software Reliability

What is Software Reliability?

"Software reliability means operational reliability. Who cares how many bugs are in the program?

As per IEEE standard: "Software reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time".

Software Reliability

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the inputs are free of error.

"It is the probability of a failure free operation of a program for a specified time in a specified environment".

Software Reliability

- Failures and Faults

A fault is the defect in the program that, when executed under particular conditions, causes a failure.

The execution time for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time is calendar time. It is the familiar time that we normally experience.

Software Reliability

There are four general ways of characterising failure occurrences in time:

1. time of failure,
2. time interval between failures,
3. cumulative failure experienced up to a given time,
4. failures experienced in a time interval.

Software Reliability

Failure Number	Failure Time (sec)	Failure interval (sec)
1	8	8
2	18	10
3	25	7
4	36	11
5	45	9
6	57	12
7	71	14
8	86	15
9	104	18
10	124	20
11	143	19
12	169	26
13	197	28
14	222	25
15	250	28

Table 7.1: Time based failure specification

Software Reliability

Time (sec)	Cumulative Failures	Failure in interval (30 sec)
30	3	3
60	6	3
90	8	2
120	9	1
150	11	2
180	12	1
210	13	1
240	14	1

Table 7.2: Failure based failure specification

Software Reliability

Value of random variable (failures in time period)	Probability	
	Elapsed time t_A = 1 hr	Elapsed time t_B = 5 hr
0	0.10	0.01
1	0.18	0.02
2	0.22	0.03
3	0.16	0.04
4	0.11	0.05
5	0.08	0.07
6	0.05	0.09
7	0.04	0.12
8	0.03	0.16
9	0.02	0.13

Table 7.3: Probability distribution at times t_A and t_B

Software Reliability

Value of random variable (failures in time period)	Probability	
	Elapsed time t_A = 1 hr	Elapsed time t_B = 5 hr
10	0.01	0.10
11	0	0.07
12	0	0.05
13	0	0.03
14	0	0.02
15	0	0.01
Mean failures	3.04	7.77

Table 7.3: Probability distribution at times t_A and t_B

Software Reliability

A random process whose probability distribution varies with time to time is called non-homogeneous. Most failure processes during test fit this situation. Fig. 7.3 illustrates the mean value and the related failure intensity functions at time t_A and t_B. Note that the mean failures experienced increases from 3.04 to 7.77 between these two points, while the failure intensity decreases.

Failure behavior is affected by two principal factors:

- ✓ the number of faults in the software being executed.
- ✓ the execution environment or the operational profile of execution.

Software Reliability

Fig. 7.3: Mean Value & failure intensity functions.

Software Reliability

Environment

The environment is described by the operational profile. The proportion of runs of various types may vary, depending on the functional environment. Examples of a run type might be:

1. a particular transaction in an airline reservation system or a business data processing system,
2. a specific cycle in a closed loop control system (for example, in a chemical process industry),
3. a particular service performed by an operating system for a user.

Software Reliability

The run types required of the program by the environment can be viewed as being selected randomly. Thus, we define the operational profile as the set of run types that the program can execute along with possibilities with which they will occur. In fig. 7.4, we show two of many possible input states A and B, with their probabilities of occurrence.

The part of the operational profile for just these two states is shown in fig. 7.5. A realistic operational profile is illustrated in fig.7.6.

Software Reliability

Fig. 7.4: Input Space

Software Reliability

Fig. 7.5: Portion of operational profile

Software Reliability

Fig. 7.6: Operational profile

Software Reliability

Fig.7.7 shows how failure intensity and reliability typically vary during a test period, as faults are removed.

Fig. 7.7: Reliability and failure intensity

Software Reliability

Uses of Reliability Studies

There are at least four other ways in which software reliability measures can be of great value to the software engineer, manager or user.

1. you can use software reliability measures to evaluate software engineering technology quantitatively.

2. software reliability measures offer you the possibility of evaluating development status during the test phases of a project.

Software Reliability

3. one can use software reliability measures to monitor the operational performance of software and to control new features added and design changes made to the software.

4. a quantitative understanding of software quality and the various factors influencing it and affected by it enriches into the software product and the software development process.

Software Reliability

Software Quality

Different people understand different meanings of quality like:

- conformance to requirements
- fitness for the purpose
- level of satisfaction

Software Reliability

Software Reliability

Fig 7.8: Software quality attributes

Software Reliability

1	Reliability	The extent to which a software performs its intended functions without failure.
2	Correctness	The extent to which a software meets its specifications.
3	Consistency & precision	The extent to which a software is consistent and give results with precision.
4	Robustness	The extent to which a software tolerates the unexpected problems.
5	Simplicity	The extent to which a software is simple in its operations.
6	Traceability	The extent to which an error is traceable in order to fix it.
7	Usability	The extent of effort required to learn, operate and understand the functions of the software

(Contd.)...

Software Reliability

8	Accuracy	Meeting specifications with precision.
9	Clarity & Accuracy of documentation	The extent to which documents are clearly & accurately written.
10	Conformity of operational environment	The extent to which a software is in conformity of operational environment.
11	Completeness	The extent to which a software has specified functions.
12	Efficiency	The amount of computing resources and code required by software to perform a function.
13	Testability	The effort required to test a software to ensure that it performs its intended functions.
14	Maintainability	The effort required to locate and fix an error during maintenance phase.

(Contd.)...

Software Reliability

15	Modularity	It is the extent of ease to implement, test, debug and maintain the software.
16	Readability	The extent to which a software is readable in order to understand.
17	Adaptability	The extent to which a software is adaptable to new platforms & technologies.
18	Modifiability	The effort required to modify a software during maintenance phase.
19	Expandability	The extent to which a software is expandable without undesirable side effects.
20	Portability	The effort required to transfer a program from one platform to another platform.

Table 7.4: Software quality attributes

Software Reliability

- McCall Software Quality Model

Fig 7.9: Software quality factors

Software Reliability

i. Product Operation

Factors which are related to the operation of a product are combined. The factors are:

- Correctness
- Efficiency
- Integrity
- Reliability
- Usability

These five factors are related to operational performance, convenience, ease of usage and its correctness. These factors play a very significant role in building customer's satisfaction.

Software Reliability

ii. Product Revision

The factors which are required for testing & maintenance are combined and are given below:

- Maintainability
- Flexibility
- Testability

These factors pertain to the testing & maintainability of software. They give us idea about ease of maintenance, flexibility and testing effort. Hence, they are combined under the umbrella of product revision.

Software Reliability

iii. Product Transition

We may have to transfer a product from one platform to an other platform or from one technology to another technology. The factors related to such a transfer are combined and given below:

- Portability
- Reusability
- Interoperability

Software Reliability

Most of the quality factors are explained in table 7.4. The remaining factors are given in table 7.5.

Sr.No.	Quality Factors	Purpose
1	Integrity	The extent to which access to software or data by the unauthorized persons can be controlled.
2	Flexibility	The effort required to modify an operational program.
3	Reusability	The extent to which a program can be reused in other applications.
4	Interoperability	The effort required to couple one system with another.

Table 7.5: Remaining quality factors (other are in table 7.4)

Quality criteria

Fig 7.10: McCall's quality model

Software Reliability

Sr. No.	Quality Criteria	Usability	Integrity	Efficiency	Correctness	Reliability	Maintain-bility	Testability	Flexibility	Reusability	Portability	Interopera-bility
1.	Operability	×										
2.	Training	×										
3.	Communicativeness	×										
4.	I/O volume	×										
5.	I/O rate	×										
6.	Access control		×									
7.	Access Audit		×									
8.	Storage efficiency			×								
9.	Execution Efficiency			×								
10.	Traceability				×							
11.	Completeness				×							
12.	Accuracy					×						
13.	Error tolerance					×						
14.	Consistency				×	×	×					
15.	Simplicity					×	×	×				
16.	Conciseness						×					
17.	Instrumentation							×				
18.	Expandability								×			
19.	Generality								×	×		
20.	Self-descriptiveness						×		×	×	×	
21.	Modularity						×		×	×	×	×
22.	Machine independence									×	×	
23.	S/W system independence									×	×	
24.	Communication commonality											×
25.	Data commonality											×

Table 7.5(a): Relation between quality factors and quality criteria

Software Reliability

1	Operability	The ease of operation of the software.
2	Training	The ease with which new users can use the system.
3	Communicativeness	The ease with which inputs and outputs can be assimilated.
4	I/O volume	It is related to the I/O volume.
5	I/O rate	It is the indication of I/O rate.
6	Access control	The provisions for control and protection of the software and data.
7	Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
8	Storage efficiency	The run time storage requirements of the software.
9	Execution efficiency	The run-time efficiency of the software.

(Contd.)...

Software Reliability

10	Traceability	The ability to link software components to requirements.
11	Completeness	The degree to which a full implementation of the required functionality has been achieved.
12	Accuracy	The precision of computations and output.
13	Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
14	Consistency	The use of uniform design and implementation techniques and notations throughout a project.
15	Simplicity	The ease with which the software can be understood.
16	Conciseness	The compactness of the source code, in terms of lines of code.
17	Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.

(Contd.)...

Software Reliability

18	Expandability	The degree to which storage requirements or software functions can be expanded.
19	Generability	The breadth of the potential application of software components.
20	Self-descriptiveness	The degree to which the documents are self explanatory.
21	Modularity	The provision of highly independent modules.
22	Machine independence	The degree to which software is dependent on its associated hardware.
23	Software system independence	The degree to which software is independent of its environment.
24	Communication commonality	The degree to which standard protocols and interfaces are used.
25	Data commonality	The use of standard data representations.

Table 7.5 (b): Software quality criteria

Software Reliability

- Boehm Software Quality Model

Fig.7.11: The Boehm software quality model

Software Reliability

ISO 9126

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Software Reliability

Characteristic/ Attribute	Short Description of the Characteristics and the concerns Addressed by Attributes
Functionality	Characteristics relating to achievement of the basic purpose for which the software is being engineered
• Suitability	The presence and appropriateness of a set of functions for specified tasks
• Accuracy	The provision of right or agreed results or effects
• Interoperability	Software's ability to interact with specified systems
• Security	Ability to prevent unauthorized access, whether accidental or deliberate, to program and data.
Reliability	Characteristics relating to capability of software to maintain its level of performance under stated conditions for a stated period of time
• Maturity	Attributes of software that bear on the frequency of failure by faults in the software

(Contd.)...

Software Reliability

• Fault tolerance	Ability to maintain a specified level of performance in cases of software faults or unexpected inputs
• Recoverability	Capability and effort needed to reestablish level of performance and recover affected data after possible failure.
Usability	Characteristics relating to the effort needed for use, and on the individual assessment of such use, by a stated implied set of users.
• Understandability	The effort required for a user to recognize the logical concept and its applicability.
• Learnability	The effort required for a user to learn its application, operation, input and output.
• Operability	The ease of operation and control by users.
Efficiency	Characteristic related to the relationship between the level of performance of the software and the amount of resources used, under stated conditions.

(Contd.)...

Software Reliability

• Time behavior	The speed of response and processing times and throughout rates in performing its function.
• Resource behavior	The amount of resources used and the duration of such use in performing its function.
Maintainability	Characteristics related to the effort needed to make modifications, including corrections, improvements or adaptation of software to changes in environment, requirements and functions specifications.
• Analyzability	The effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.
• Changeability	The effort needed for modification, fault removal or for environmental change.
• Stability	The risk of unexpected effect of modifications.
• Testability	The effort needed for validating the modified software.

(Contd.)...

Software Reliability

Portability	Characteristics related to the ability to transfer the software from one organization or hardware or software environment to another.
• Adaptability	The opportunity for its adaptation to different specified environments.
• Installability	The effort needed to install the software in a specified environment.
• Conformance	The extent to which it adheres to standards or conventions relating to portability.
• Replaceability	The opportunity and effort of using it in the place of other software in a particular environment.

Table 7.6: Software quality characteristics and attributes – The ISO 9126 view

Software Reliability

Fig.7.12: ISO 9126 quality model

Software Reliability Models

- Basic Execution Time Model

$$\lambda(\mu) = \lambda_0 \left(1 - \frac{\mu}{V_0}\right) \quad (1)$$

Fig.7.13: Failure intensity λ as a function of μ for basic model

Software Reliability

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} \qquad (2)$$

Fig.7.14: Relationship between τ & μ for basic model

Software Reliability

For a derivation of this relationship, equation 1 can be written as:

$$\frac{d\mu(\tau)}{d\tau} = \lambda_0 \left(1 - \frac{\mu(\tau)}{V_0} \right)$$

The above equation can be solved for $\mu(\tau)$ and result in :

$$\mu(\tau) = V_0 \left(1 - \exp\left(\frac{-\lambda_0 \tau}{V_0} \right) \right) \quad \textbf{(3)}$$

Software Reliability

The failure intensity as a function of execution time is shown in figure given below

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

Fig.7.15: Failure intensity versus execution time for basic model

Software Reliability

- Derived quantities

Fig.7.16: Additional failures required to be experienced to reach the objective

Software Reliability

Fig.7.17: Additional time required to reach the objective

This can be derived in mathematical form as:

$$\Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

Software Reliability

Example- 7.1

Assume that a program will experience 200 failures in infinite time. It has now experienced 100. The initial failure intensity was 20 failures/CPU hr.

(i) Determine the current failure intensity.

(ii) Find the decrement of failure intensity per failure.

(iii) Calculate the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(iv) Compute addition failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

Software Reliability

Solution

Here V_o=200 failures

$$\mu = 100 \text{ failures}$$

$$\lambda_0 = 20 \text{ failures/CPU hr.}$$

(i) Current failure intensity:

$$\lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right)$$

$$= 20\left(1 - \frac{100}{200}\right) = 20(1 - 0.5) = 10 \text{ failures/CPU hr}$$

Software Reliability

(ii) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = \frac{-\lambda_0}{V_0} = -\frac{20}{200} = -0.1/\text{CPU hr.}$$

(iii) (a) Failures experienced & failure intensity after 20 CPU hr:

$$\mu(\tau) = V_0\left(1 - \exp\left(\frac{-\lambda_0\tau}{V_0}\right)\right)$$

$$= 200\left(1 - \exp\left(\frac{-20\times 20}{200}\right)\right) = 200(1 - \exp(1-2))$$

$$= 200(1 - 0.1353) \approx 173\text{failures}$$

Software Reliability

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

$$= 20 \exp\left(\frac{-20 \times 20}{200}\right) = 20 \exp(-2) = 2.71 \, failures / CPU \;\; hr$$

(b) Failures experienced & failure intensity after 100 CPU hr:

$$\mu(\tau) = V_0 \left(1 - \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)\right)$$

$$= 200 \left(1 - \exp\left(\frac{-20 \times 100}{200}\right)\right) = 200 \text{ failures(almost)}$$

$$\lambda(\tau) = \lambda_0 \exp\left(\frac{-\lambda_0 \tau}{V_0}\right)$$

Software Reliability

$$= 20\exp\left(\frac{-20\times 100}{200}\right) = 0.000908 \ failures / CPU \ hr$$

(iv) Additional failures $(\Delta\mu)$ required to reach the failure intensity objective of 5 failures/CPU hr.

$$\Delta\mu = \left(\frac{V_0}{\lambda_0}\right)(\lambda_P - \lambda_F) = \left(\frac{200}{20}\right)(10-5) = 50 \text{ failures}$$

Software Reliability

Additional execution time required to reach failure intensity objective of 5 failures/CPU hr.

$$\Delta\tau = \left(\frac{V_0}{\lambda_0}\right) Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{200}{20} Ln\left(\frac{10}{5}\right) = 6.93 \text{ CPU hr.}$$

Software Reliability

- Logarithmic Poisson Execution Time Model

Failure Intensity

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

Fig.7.18: Relationship between μ & λ

Software Reliability

$$\frac{d\lambda}{d\mu} = -\lambda_0 \theta \exp(-\mu\theta)$$

$$\frac{d\lambda}{d\mu} = -\theta\lambda$$

Fig.7.19: Relationship between

Software Reliability

$$\mu(\tau) = \frac{1}{\theta} Ln(\lambda_0 \theta \tau + 1)$$

$$\lambda(\tau) = \lambda_0 / (\lambda_0 \theta \tau + 1)$$

$$\Delta\mu = \frac{1}{\theta} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right] \quad \textbf{(4)}$$

λ_P = Present failure intensity

λ_F = Failure intensity objective

Example- 7.2

Assume that the initial failure intensity is 20 failures/CPU hr. The failure intensity decay parameter is 0.02/failures. We have experienced 100 failures up to this time.

(i) Determine the current failure intensity.

(ii) Calculate the decrement of failure intensity per failure.

(iii) Find the failures experienced and failure intensity after 20 and 100 CPU hrs. of execution.

(iv) Compute the additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use Logarithmic Poisson execution time model for the above mentioned calculations.

Software Reliability

Solution

$$\lambda_0 = 20 \text{ failures/CPU hr.}$$

$$\mu = 100 \text{ failures}$$

$$\theta = 0.02 / \text{failures}$$

(i) Current failure intensity:

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

= 20 exp (-0.02 x 100)

= 2.7 failures/CPU hr.

Software Reliability

(ii) Decrement of failure intensity per failure can be calculated as:

$$\frac{d\lambda}{d\mu} = -\theta\lambda$$

= -.02 x 2.7 = -.054/CPU hr.

(iii) (a) Failures experienced & failure intensity after 20 CPU hr:

$$\mu(\tau) = \frac{1}{\theta} Ln\left(\lambda_0 \theta \tau + 1\right)$$

$$= \frac{1}{0.02} Ln(20 \times 0.02 \times 20 + 1) = 109 \ failures$$

Software Reliability

$$\lambda(\tau) = \lambda_0 / (\lambda_0 \theta \tau + 1)$$

$$= (20) / (20 \times .02 \times 20 + 1) = 2.22 \ failures / CPU \ hr.$$

(b) Failures experienced & failure intensity after 100 CPU hr:

$$\mu(\tau) = \frac{1}{\theta} Ln(\lambda_0 \theta \tau + 1)$$

$$= \frac{1}{0.02} Ln(20 \times 0.02 \times 100 + 1) = 186 \ failures$$

$$\lambda(\tau) = \lambda_0 / (\lambda_0 \theta \tau + 1)$$

$$= (20) / (20 \times .02 \times 100 + 1) = 0.4878 \ failures / CPU \ hr.$$

Software Reliability

(iv) Additional failures $(\Delta\mu)$ required to reach the failure intensity objective of 2 failures/CPU hr.

$$\Delta\mu = \frac{1}{\theta} Ln \frac{\lambda_P}{\lambda_F} = \frac{1}{0.02} Ln\left(\frac{2.7}{2}\right) = 15 \text{ failures}$$

$$\Delta\tau = \frac{1}{\theta}\left[\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right] = \frac{1}{0.02}\left[\frac{1}{2} - \frac{1}{2.7}\right] = 6.5 \text{ CPU hr.}$$

Example- 7.3

The following parameters for basic and logarithmic Poisson models are given:

(a) Determine the addition failures and additional execution time required to reach the failure intensity objective of 5 failures/CPU hr. for both models.

(b) Repeat this for an objective function of 0.5 failure/CPU hr. Assume that we start with the initial failure intensity only.

Basic execution time model	Logarithmic Poisson execution time model
$\lambda_o = 10$ failures/CPU hr	$\lambda_o = 30$ failures/CPU hr
$V_o = 100$ failures	$\theta = 0.25/\text{failure}$

Solution

(a) (i) Basic execution time model

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

$$= \frac{100}{10}(10-5) = 50 \text{ failures}$$

λ_P (Present failure intensity) in this case is same as λ_0 (initial failure intensity).

Now,

$$\Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

Software Reliability

$$= \frac{100}{10} Ln\left(\frac{10}{5}\right) = 6.93 \text{ CPU hr.}$$

(ii) Logarithmic execution time model

$$\Delta\mu = \frac{1}{\theta} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} Ln\left(\frac{30}{5}\right) = 71.67 \text{ Failures}$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025} Ln\left(\frac{1}{5} - \frac{1}{30}\right) = 6.66 \text{ CPU hr.}$$

Software Reliability

Logarithmic model has calculated more failures in almost some duration of execution time initially.

(b) Failure intensity objective (λ_F) = 0.5 failures/CPU hr.

(i) Basic execution time model

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

$$= \frac{100}{10}(10 - 0.5) = 95 \ failures$$

$$\Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{100}{10} Ln\left(\frac{10}{0.05}\right) = 30 \ CPU/hr$$

Software Reliability

(ii) Logarithmic execution time model

$$\Delta\mu = \frac{1}{\theta} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} Ln\left(\frac{30}{0.5}\right) = 164 \ failures$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025}\left(\frac{1}{0.5} - \frac{1}{30}\right) = 78.66 \ CPU/hr$$

Software Reliability

- Calendar Time Component

The calendar time component is based on a debugging process model. This model takes into account:

1. resources used in operating the program for a given execution time and processing an associated quantity of failure.
2. resources quantities available, and
3. the degree to which a resource can be utilized (due to bottlenecks) during the period in which it is limiting.

Table 7.7 will help in visualizing these different aspects of the resources, and the parameters that result.

Resource usage

	Usage parameters requirements per		Planned parameters	
Resource	**CPU hr**	**Failure**	**Quantities available**	**Utilisation**
Failure identification personnel	θ_I	μ_I	P_I	1
Failure correction personnel	0	μ_f	P_f	P_f
Computer time	θc	μ_c	P_c	P_c

Fig. : Calendar time component resources and parameters

Software Reliability

Hence, to be more precise, we have

$$X_C = \mu_c \Delta\mu + \theta_c \Delta\tau$$ (for computer time)

$$X_f = \mu_f \Delta\mu$$ (for failure correction)

$$X_I = \mu_I \Delta\mu + \theta_I \Delta\tau$$ (for failure identification)

$$dx_T / d\tau = \theta_r + \mu_r \lambda$$

Software Reliability

Calendar time to execution time relationship

$$dt / d\tau = (1 / P_r p_r) dx_T / d\tau$$

$$dt / d\tau = (\theta_r + \mu_r \lambda) / P_r p_r$$

Software Reliability

Fig.7.20: Instantaneous calendar time to execution time ratio

Software Reliability

Fig.7.21: Calendar time to execution time ratio for different limiting resources

Software Reliability

Example- 7.4

A team run test cases for 10 CPU hrs and identifies 25 failures. The effort required per hour of execution time is 5 person hr. Each failure requires 2 hr. on an average to verify and determine its nature. Calculate the failure identification effort required.

Software Reliability

Solution

As we know, resource usage is:

$$X_r = \theta_r \tau + \mu_r \mu$$

Here $\theta_r = 15$ person hr. $\mu = 25$ failures

$\tau = 10$ CPU hrs. $\mu_r = 2$ hrs./failure

Hence, $X_r = 5\ (10) + 2\ (25)$

$= 50 + 50 = 100$ person hr.

Example- 7.5

Initial failure intensity (λ_0) for a given software is 20 failures/CPU hr. The failure intensity objective (λ_F) of 1 failure/CPU hr. is to be achieved. Assume the following resource usage parameters.

Resource Usage	Per hour	Per failure
Failure identification effort	2 Person hr.	1 Person hr.
Failure Correction effort	0	5 Person hr.
Computer time	1.5 CPU hr.	1 CPU hr.

Software Reliability

(a) What resources must be expended to achieve the reliability improvement? Use the logarithmic Poisson execution time model with a failure intensity decay parameter of 0.025/failure.

(b) If the failure intensity objective is cut to half, what is the effect on requirement of resources ?

Solution

(a) $$\Delta\mu = \frac{1}{\theta} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$$

$$= \frac{1}{0.025} Ln\left(\frac{20}{1}\right) = 119 \text{ failures}$$

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right)$$

$$= \frac{1}{0.025}\left(\frac{1}{1} - \frac{1}{20}\right) = \frac{1}{0.025}(1 - 0.05) = 38 \text{ CPU hrs.}$$

Software Reliability

Hence

$$X_1 = \mu_1 \Delta\mu + \theta_1 \Delta\tau$$

= 1 (119) + 2 (38) = 195 Person hrs.

$$X_F = \mu_F \Delta\mu$$

= 5 (119) = 595 Person hrs.

$$X_C = \mu_c \Delta\mu + \theta_c \Delta\tau$$

= 1 (119) + (1.5) (38) = 176 CPU hr.

Software Reliability

(b) $\lambda_F = 0.5$ failures/CPU hr.

$$\Delta\mu = \frac{1}{0.025} Ln\left(\frac{20}{0.5}\right) = 148 \text{ failures}$$

$$\Delta\tau = \frac{1}{0.025}\left(\frac{1}{0.5} - \frac{1}{20}\right) = 78 \text{ CPU hr.}$$

So, $X_I = 1\ (148) + 2\ (78) = 304$ Person hrs.

$X_F = 5\ (148) = 740$ Person hrs.

$X_C = 1\ (148) + (1.5)(78) = 265$ CPU hrs.

Software Reliability

Hence, if we cut failure intensity objective to half, resources requirements are not doubled but they are some what less. Note that $\Delta\tau$ is approximately doubled but increases logarithmically. Thus, the resources increase will be between a logarithmic increase and a linear increase for changes in failure intensity objective.

Software Reliability

Example- 7.6

A program is expected to have 500 faults. It is also assumed that one fault may lead to one failure only. The initial failure intensity was 2 failures/CPU hr. The program was to be released with a failure intensity objective of 5 failures/100 CPU hr. Calculated the number of failure experienced before release.

Software Reliability

Solution

The number of failure experienced during testing can be calculated using the equation mentioned below:

$$\Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$$

Here $V_0 = 500$ because one fault leads to one failure

$\lambda_0 = 2$ failures/CPU hr.

$\lambda_F = 5$ failures/100 CPU hr.

$= 0.05$ failures/CPU hr.

Software Reliability

So $$\Delta\mu = \frac{500}{2}(2 - 0.05)$$

= 487 failures

Hence 13 faults are expected to remain at the release instant of the software.

Software Reliability

- The Jelinski-Moranda Model

$$\lambda(t) = \phi(N - i + 1)$$

where

ϕ = Constant of proportionality

N = Total number of errors present

I = number of errors found by time interval t_i

Software Reliability

Fig.7.22: Relation between t & λ

Example- 7.7

There are 100 errors estimated to be present in a program. We have experienced 60 errors. Use Jelinski-Moranda model to calculate failure intensity with a given value of ϕ=0.03. What will be failure intensity after the experience of 80 errors?

Software Reliability

Solution

N = 100 errors

i = 60 failures

ϕ = 0.03

We know $$\lambda(t) = 0.03(100 - 60 + 1)$$

= 0.03(100-60+1)

= 1.23 failures/CPU hr.

After 80 failures $$\lambda(t) = 0.03(100 - 80 + 1)$$

= 0.63 failures/CPU hr.

Hence, there is continuous decrease in the failure intensity as the number of failure experienced increases.

Software Reliability

- The Bug Seeding Model

The bug seeding model is an outgrowth of a technique used to estimate the number of animals in a wild life population or fish in a pond.

$$\frac{N_t}{N + N_t} = \frac{n_t}{n + n_t}$$

$$\hat{N} = \frac{n}{n_t} N_t$$

$$N = \frac{n}{n_s} N_s$$

Software Reliability

- Capability Maturity Model

It is a strategy for improving the software process, irrespective of the actual life cycle model used.

Fig.7.23: Maturity levels of CMM

Software Reliability

Maturity Levels:

- ✓ Initial (Maturity Level 1)
- ✓ Repeatable (Maturity Level 2)
- ✓ Defined (Maturity Level 3)
- ✓ Managed (Maturity Level 4)
- ✓ Optimizing (Maturity Level 5)

Software Reliability

Maturity Level	Characterization
Initial	Adhoc Process
Repeatable	Basic Project Management
Defined	Process Definition
Managed	Process Measurement
Optimizing	Process Control

Fig.7.24: The five levels of CMM

Software Reliability

- Key Process Areas

The key process areas at level 2 focus on the software project's concerns related to establishing basic project management controls, as summarized below:

Requirements Management (RM)	Establish a common relationship between the customer requirements and the developers in order to understand the requirements of the project.
Software Project Planning (PP)	Establish reasonable plans for performing the software engineering and for managing the software project.
Software Project Tracking and Oversight (PT)	Establish adequate visibility into actual progress so that management can take effective actions when the software project's performance deviates significantly from the software plans.
Software Subcontract Management (SM)	Select qualified software subcontractors and manage them effectively.
Software Quality Assurance (QA)	Provide management with appropriate visibility into the process being used by the software project and of the products being built.
Software Configuration Management (CM)	Establish and maintain the integrity of the products of the software project throughout the project's software life cycle.

Software Reliability

The key process areas at level 3 address both project and organizational issues, as summarized below:

Organization Process Focus (PF)	Establish the organizational responsibility for software process activities that improve the organization's overall software process capability.
Organization Process Definition (PD)	Develop and maintain a usable set of software process assets that improve process performance across the projects and provide a basis for cumulative, long-term benefits to the organization.
Training Program (TP)	Develop the skills and knowledge of individuals so that they can perform their roles effectively and efficiently.
Integrated Software Management (IM)	Integrate the software engineering and management activities into a coherent, defined software process that is tailored from the organization's standard software process and related process assets.

(Contd.)...

Software Reliability

Software Product Engineering (PE)	Consistently perform a well-defined engineering process that integrates all the software engineering activities to produce correct, consistent software products effectively and efficiently.
Inter group Coordination (IC)	Establish a means for the software engineering group to participate actively with the other engineering groups so the project is better able to satisfy the customer's needs effectively and efficiently.
Peer Reviews (PR)	Remove defects from the software work products early and efficiently. An important corollary effect is to develop a better understanding of the software work products and of the defects that can be prevented.

Software Reliability

The key process areas at level 4 focus on establishing a quantitative understanding of both the software process and the software work products being built, as summarized below:

Quantitative Process Management (QP)	Control the process performance of the software project quantitatively.
Software Quality Management (QM)	Develop a quantitative understanding of the quality of the project's software products and achieve specific quality goals.

Software Reliability

The key process areas at level 5 cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement, as summarized below:

Defect Prevention (DP)	Identify the causes of defects and prevent them from recurring.
Technology Change Management (TM)	Identify beneficial new technologies (i.e., tools, methods, and processes) and transfer them into the organization in an orderly manner.
Process Change Management (PC)	Continually improve the software processes used in the organization with the intent of improving software quality, increasing productivity, and decreasing the cycle time for product development.

Software Reliability

- Common Features

Commitment to Perform (CO)	Describes the actions the organizations must take to ensure that the process is established and will endure. It includes practices on policy and leadership.
Ability to Perform (AB)	Describes the preconditions that must exist in the project or organization to implement the software process competently. It includes practices on resources, organizational structure, training, and tools.
Activities Performed (AC)	Describes the role and procedures necessary to implement a key process area. It includes practices on plans, procedures, work performed, tracking, and corrective action.
Measurement and Analysis (ME)	Describes the need to measure the process and analyze the measurements. It includes examples of measurements.
Verifying Implementation (VE)	Describes the steps to ensure that the activities are performed in compliance with the process that has been established. It includes practices on management reviews and audits.

Software Reliability

- ISO 9000

The SEI capability maturity model initiative is an attempt to improve software quality by improving the process by which software is developed.

ISO-9000 series of standards is a set of document dealing with quality systems that can be used for quality assurance purposes. ISO-9000 series is not just software standard. It is a series of five related standards that are applicable to a wide variety of industrial activities, including design/ development, production, installation, and servicing. Within the ISO 9000 Series, standard ISO 9001 for quality system is the standard that is most applicable to software development.

Software Reliability

- Mapping ISO 9001 to the CMM

1. Management responsibility
2. Quality system
3. Contract review
4. Design control
5. Document control
6. Purchasing
7. Purchaser-supplied product

Software Reliability

8. Product identification and traceability
9. Process control
10. Inspection and testing
11. Inspection, measuring and test equipment
12. Inspection and test status
13. Control of nonconforming product
14. Corrective action

Software Reliability

15. Handling, storage, packaging and delivery
16. Quality records
17. Internal quality audits
18. Training
19. Servicing
20. Statistical techniques

Software Reliability

- Contrasting ISO 9001 and the CMM

There is a strong correlation between ISO 9001 and the CMM, although some issues in ISO 9001 are not covered in the CMM, and some issues in the CMM are not addressed in ISO 9001.

The biggest difference, however, between these two documents is the emphasis of the CMM on continuous process improvement.

The biggest similarity is that for both the CMM and ISO 9001, the bottom line is **"Say what you do; do what you say"**.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

7.1 Which one is not a phase of "bath tub curve" of hardware reliability

(a) Burn-in (b) Useful life

(c) Wear-out (d) Test-out

7.2 Software reliability is

(a) the probability of failure free operation of a program for a specified time in a specified environment

(b) the probability of failure of a program for a specified time in a specified environment

(c) the probability of success of a program for a specified time in any environment

(d) None of the above

7.3 Fault is

(a) Defect in the program (b) Mistake in the program

(c) Error in the program (d) All of the above

7.4 One fault may lead to

(a) one failure (b) two failures

(c) many failures (d) all of the above

Multiple Choice Questions

7.5 Which 'time' unit is not used in reliability studies

(a) Execution time (b) Machine time

(c) Clock time (d) Calendar time

7.6 Failure occurrences can be represented as

(a) time to failure (b) time interval between failures

(c) failures experienced in a time interval (d) All of the above

7.7 Maximum possible value of reliability is

(a) 100 (b) 10

(c) 1 (d) 0

7.8 Minimum possible value of reliability is

(a) 100 (b) 10

(c) 1 (d) 0

7.9 As the reliability increases, failure intensity

(a) decreases (b) increases

(c) no effect (d) None of the above

Multiple Choice Questions

7.10 If failure intensity is 0.005 failures/hour during 10 hours of operation of a software, its reliability can be expressed as

(a) 0.10 (b) 0.92

(c) 0.95 (d) 0.98

7.11 Software Quality is

(a) Conformance to requirements (b) Fitness for the purpose

(c) Level of satisfaction (d) All of the above

7.12 Defect rate is

(a) number of defects per million lines of source code

(b) number of defects per function point

(c) number of defects per unit of size of software

(d) All of the above

7.13 How many product quality factors have been proposed in McCall quality model?

(a) 2 (b) 3

(c) 11 (d) 6

Multiple Choice Questions

7.14 Which one is not a product quality factor of McCall quality model?

(a) Product revision (b) Product operation
(c) Product specification (d) Product transition

7.15 The second level of quality attributes in McCall quality model are termed as

(a) quality criteria (b) quality factors
(c) quality guidelines (d) quality specifications

7.16 Which one is not a level in Boehm software quality model ?

(a) Primary uses (b) Intermediate constructs
(c) Primitive constructs (d) Final constructs

7.17 Which one is not a software quality model?

(a) McCall model (b) Boehm model
(c) ISO 9000 (d) ISO 9126

7.18 Basic execution time model was developed by

(a) Bev.Littlewood (b) J.D.Musa
(c) R.Pressman (d) Victor Baisili

Multiple Choice Questions

7.19 NHPP stands for

(a) Non Homogeneous Poisson Process (b) Non Hetrogeneous Poisson Process

(c) Non Homogeneous Poisson Product (d) Non Hetrogeneous Poisson Product

7.20 In Basic execution time model, failure intensity is given by

$(a)\ \lambda(\mu) = \lambda_0\left(1 - \frac{\mu^2}{V_0}\right)$ $(b)\ \lambda(\mu) = \lambda_0\left(1 - \frac{\mu}{V_0}\right)$

$(c)\ \lambda(\mu) = \lambda_0\left(1 - \frac{V_0}{\mu}\right)$ $(d)\ \lambda(\mu) = \lambda_0\left(1 - \frac{V_0}{\mu^2}\right)$

7.21 In Basic execution time model, additional number of failures required to achieve a failure intensity objective $(\Delta\mu)$ is expressed as

$(a)\ \Delta\mu = \frac{V_0}{\lambda_0}(\lambda_P - \lambda_F)$ $(b)\ \Delta\mu = \frac{V_0}{\lambda_0}(\lambda_F - \lambda_P)$

$(c)\ \Delta\mu = \frac{\lambda_0}{V_0}(\lambda_F - \lambda_P)$ $(d)\ \Delta\mu = \frac{\lambda_0}{V_0}(\lambda_P - \lambda_F)$

Multiple Choice Questions

7.22 In Basic execution time model, additional time required to achieve a failure intensity objective $(\Delta\tau)$ is given as

$(a)\ \Delta\tau = \frac{\lambda_0}{V_0} Ln\left(\frac{\lambda_F}{\lambda_P}\right)$

$(b)\ \Delta\tau = \frac{\lambda_0}{V_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$

$(c)\ \Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_F}{\lambda_P}\right)$

$(d)\ \Delta\tau = \frac{V_0}{\lambda_0} Ln\left(\frac{\lambda_P}{\lambda_F}\right)$

7.23 Failure intensity function of Logarithmic Poisson execution model is given as

$(a)\ \lambda(\mu) = \lambda_0 LN(-\theta\mu)$

$(b)\ \lambda(\mu) = \lambda_0 \exp(\theta\mu)$

$(c)\ \lambda(\mu) = \lambda_0 \exp(-\theta\mu)$

$(d)\ \lambda(\mu) = \lambda_0 \log(-\theta\mu)$

7.24 In Logarithmic Poisson execution model, 'θ' is known as

(a) Failure intensity function parameter (b) Failure intensity decay parameter

(c) Failure intensity measurement (d) Failure intensity increment parameter

Multiple Choice Questions

7.25 In jelinski-Moranda model, failure intensity is defined aseneous Poisson Product

$(a)\ \lambda(t) = \phi(N - i + 1)$ $(b)\ \lambda(t) = \phi(N + i + 1)$

$(c)\ \lambda(t) = \phi(N + i - 1)$ $(d)\ \lambda(t) = \phi(N - i - 1)$

7.26 CMM level 1 has

(a) 6 KPAs (b) 2 KPAs

(c) 0 KPAs (d) None of the above

7.27 MTBF stands for

(a) Mean time between failure (b) Maximum time between failures

(c) Minimum time between failures (d) Many time between failures

7.28 CMM model is a technique to

(a) Improve the software process (b) Automatically develop the software

(c) Test the software (d) All of the above

7.29 Total number of maturing levels in CMM are

(a) 1 (b) 3

(c) 5 (d) 7

Multiple Choice Questions

7.30 Reliability of a software is dependent on number of errors

(a) removed
(b) remaining
(c) both (a) & (b)
(d) None of the above

7.31 Reliability of software is usually estimated at

(a) Analysis phase
(b) Design phase
(c) Coding phase
(d) Testing phase

7.32 CMM stands for

(a) Capacity maturity model
(b) Capability maturity model
(c) Cost management model
(d) Comprehensive maintenance model

7.33 Which level of CMM is for basic project management?

(a) Initial
(b) Repeatable
(c) Defined
(d) Managed

7.34 Which level of CMM is for process management?

(a) Initial
(b) Repeatable
(c) Defined
(d) Optimizing

Multiple Choice Questions

7.35 Which level of CMM is for process management?

(a) Initial
(b) Defined
(c) Managed
(d) Optimizing

7.36 CMM was developed at

(a) Harvard University
(b) Cambridge University
(c) Carnegie Mellon University
(d) Maryland University

7.37 McCall has developed a

(a) Quality model
(b) Process improvement model
(c) Requirement model
(d) Design model

7.38 The model to measure the software process improvement is called

(a) ISO 9000
(b) ISO 9126
(c) CMM
(d) Spiral model

7.39 The number of clauses used in ISO 9001 are

(a) 15
(b) 25
(c) 20
(d) 10

Multiple Choice Questions

7.40 ISO 9126 contains definitions of

(a) quality characteristics
(b) quality factors
(c) quality attributes
(d) All of the above

7.41 In ISO 9126, each characteristics is related to

(a) one attributes
(b) two attributes
(c) three attributes
(d) four attributes

7.42 In McCall quality model; product revision quality factor consist of

(a) Maintainability
(b) Flexibility
(c) Testability
(d) None of the above

7.43 Which is not a software reliability model ?

(a) The Jelinski-Moranda Model
(b) Basic execution time model
(c) Spiral model
(d) None of the above

7.44 Each maturity model is CMM has

(a) One KPA
(b) Equal KPAs
(c) Several KPAs
(d) no KPA

Multiple Choice Questions

7.45 KPA in CMM stands for

(a) Key Process Area
(b) Key Product Area
(c) Key Principal Area
(d) Key Performance Area

7.46 In reliability models, our emphasis is on

(a) errors
(b) faults
(c) failures
(d) bugs

7.47 Software does not break or wear out like hardware. What is your opinion?

(a) True
(b) False
(c) Can not say
(d) not fixed

7.48 Software reliability is defined with respect to

(a) time
(b) speed
(c) quality
(d) None of the above

7.49 MTTF stands for

(a) Mean time to failure
(b) Maximum time to failure
(c) Minimum time to failure
(d) None of the above

Multiple Choice Questions

7.50 ISO 9000 is a series of standards for quality management systems and has

(a) 2 related standards
(b) 5 related standards
(c) 10 related standards
(d) 25 related standards

Exercises

7.1 What is software reliability? Does it exist?

7.2 Explain the significance of bath tube curve of reliability with the help of a diagram.

7.3 Compare hardware reliability with software reliability.

7.4 What is software failure? How is it related with a fault?

7.5 Discuss the various ways of characterising failure occurrences with respect to time.

7.6 Describe the following terms:

(i) Operational profile (ii) Input space

(iii) MTBF (iv) MTTF

(v) Failure intensity.

Exercises

7.7 What are uses of reliability studies? How can one use software reliability measures to monitor the operational performance of software?

7.8 What is software quality? Discuss software quality attributes.

7.9 What do you mean by software quality standards? Illustrate their essence as well as benefits.

7.10 Describe the McCall software quality model. How many product quality factors are defined and why?

7.11 Discuss the relationship between quality factors and quality criteria in McCall's software quality model.

7.12 Explain the Boehm software quality model with the help of a block diagram.

7.13 What is ISO9126 ? What are the quality characteristics and attributes?

Exercises

7.14 Compare the ISO9126 with McCall software quality model and highlight few advantages of ISO9126.

7.15 Discuss the basic model of software reliability. How $\Delta\mu$ *and* $\Delta\tau$ can be calculated.

7.16 Assume that the initial failure intensity is 6 failures/CPU hr. The failure intensity decay parameter is 0.02/failure. We assume that 45 failures have been experienced. Calculate the current failure intensity.

7.17 Explain the basic & logarithmic Poisson model and their significance in reliability studies.

Exercises

7.18 Assume that a program will experience 150 failures in infinite time. It has now experienced 80. The initial failure intensity was 10 failures/CPU hr.

(i) Determine the current failure intensity

(ii) Calculate the failures experienced and failure intensity after 25 and 40 CPU hrs. of execution.

(iii) Compute additional failures and additional execution time required to reach the failure intensity objective of 2 failures/CPU hr.

Use the basic execution time model for the above mentioned calculations.

7.19 Write a short note on Logarithmic Poisson Execution time model. How can we calculate $\Delta\mu$ & $\Delta\tau$?

7.20 Assume that the initial failure intensity is 10 failures/CPU hr. The failure intensity decay parameter is 0.03/failure. We have experienced 75 failures upto this time. Find the failures experienced and failure intensity after 25 and 50 CPU hrs. of execution.

Exercises

7.21 The following parameters for basic and logarithmic Poisson models are given:

Basic execution time model	*Logarithmic Poisson execution time model*
λ_0 = 5 failures/CPU hr	λ_0 = 25 failures/CPU hr
V_0 = 125 failures	θ = 0.3/failure

Determine the additional failures and additional execution time required to reach the failure intensity objective of 0.1 failure/CPU hr. for both models.

7.22 Quality and reliability are related concepts but are fundamentally different in a number of ways. Discuss them.

7.23 Discuss the calendar time component model. Establish the relationship between calendar time to execution time.

Exercises

7.24 A program is expected to have 250 faults. It is also assumed that one fault may lead to one failure. The initial failure intensity is 5 failure/CPU hr. The program is released with a failure intensity objective of 4 failures/10 CPU hr. Calculate the number of failures experienced before release.

7.25 Explain the Jelinski-Moranda model of reliability theory. What is the relation between 't' and 'λ'?

7.26 Describe the Mill's bug seeding model. Discuss few advantages of this model over other reliability models.

7.27 Explain how the CMM encourages continuous improvement of the software process.

7.28 Discuss various key process areas of CMM at various maturity levels.

7.29 Construct a table that correlates key process areas (KPAs) in the CMM with ISO9000.

7.30 Discuss the 20 clauses of ISO9001 and compare with the practices in the CMM.

Exercises

7.31 List the difference of CMM and ISO9001. Why is it suggested that CMM is the better choice than ISO9001?

7.32 Explain the significance of software reliability engineering. Discuss the advantage of using any software standard for software development?

7.33 What are the various key process areas at defined level in CMM? Describe activities associated with one key process area.

7.34 Discuss main requirements of ISO9001 and compare it with SEI capability maturity model.

7.35 Discuss the relative merits of ISO9001 certification and the SEI CMM based evaluation. Point out some of the shortcomings of the ISO9001 certification process as applied to the software industry.

Software Testing

Software Testing

- What is Testing?

Many people understand many definitions of testing **:**

1. Testing is the process of demonstrating that errors are not present.
2. The purpose of testing is to show that a program performs its intended functions correctly.
3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect.

Software Testing

A more appropriate definition is:

"*Testing is the process of executing a program with the intent of finding errors*."

Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Software Testing

- Who should Do the Testing ?
  - Testing requires the developers to find errors from their software.
  - It is difficult for software developer to point out errors from own creations.
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

Software Testing

- What should We Test ?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 x 2^8$. If only one second it required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

Software Testing

Fig. 1: Control flow graph

Software Testing

The number of paths in the example of Fig. 1 are 10^{14} or 100 trillions. It is computed from $5^{20} + 5^{19} + 5^{18} + \ldots\ldots + 5^{1}$; where 5 is the number of paths through the loop body. If only 5 minutes are required to test one test path, it may take approximately one billion years to execute every path.

Some Terminologies

- **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

- Verification and Validation

Verification is 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.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

Software Testing

➢ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Functional Testing

Fig. 3: Black box testing

Boundary Value Analysis

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \leq x \leq b$$

$$c \leq y \leq d$$

Fig.4: Input domain for program having two input variables

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 8.5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Example- 8.I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

Software Testing

The boundary value test cases are :

Test Case	a	b	c	Expected output
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example – 8.2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

Software Testing

The boundary value test cases are:

Test Case	Month	Day	Year	Expected output
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Example – 8.3

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:

[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected Output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Robustness testing

It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This extended form of boundary value analysis is called robustness testing and shown in Fig. 6

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are 6n+1, where n is the number of input variables. So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200)

Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

Worst-case testing

If we reject "single fault" assumption theory of reliability and may like to see what happens when more than one variable has an extreme value. In electronic circuits analysis, this is called "worst case analysis". It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of n variables generate 5^n test cases as opposed to $4n+1$ test cases for boundary value analysis. Our two variables example will have $5^2=25$ test cases and are given in table 1.

Table 1: Worst cases test inputs for two variables example

Test case number	Inputs		Test case number	Inputs	
	x	*y*		*x*	*y*
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	--		

Example - 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Design the Robust test case and worst test cases for this program.

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Software Testing

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

Software Testing

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	a	b	c	Expected output
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic

(Contd.)...

Software Testing

Test Case	A	b	c	Expected output
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44`	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
66	50	99	0	Real Roots
67	50	99	1	Real Roots
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
101	100	0	0	Equal Roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real Roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real Roots
112	100	50	1	Real Roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real Roots
117	100	99	1	Real Roots
118	100	99	50	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Example – 8.5

Consider the program for the determination of previous date in a calendar as explained in example 8.2. Design the robust and worst test cases for this program.

Solution

Robust test cases are $6n+1$. Hence total 19 robust test cases are designed and are given on next slide.

Software Testing

Test case	Month	Day	Year	Expected Output
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Software Testing

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	Month	Day	Year	Expected output
1	1	1	1900	31 December, 1899
2	1	1	1901	31 December, 1900
3	1	1	1962	31 December, 1961
4	1	1	2024	31 December, 2023
5	1	1	2025	31 December, 2024
6	1	2	1900	1 January, 1900
7	1	2	1901	1 January, 1901
8	1	2	1962	1 January, 1962
9	1	2	2024	1 January, 2024
10	1	2	2025	1 January, 2025
11	1	15	1900	14 January, 1900
12	1	15	1901	14 January, 1901
13	1	15	1962	14 January, 1962
14	1	15	2024	14 January, 2024

(Contd.)...

Software Testing

Test Case	A	b	c	Expected output
15	1	15	2025	14 January, 2025
16	1	30	1900	29 January, 1900
17	1	30	1901	29 January, 1901
18	1	30	1962	29 January, 1962
19	1	30	2024	29 January, 2024
20	1	30	2025	29 January, 2025
21	1	31	1900	30 January, 1900
22	1	31	1901	30 January, 1901
23	1	31	1962	30 January, 1962
24	1	31	2024	30 January, 2024
25	1	31	2025	30 January, 2025
26	2	1	1900	31 January, 1900
27	2	1	1901	31 January, 1901
28	2	1	1962	31 January, 1962
29	2	1	2024	31 January, 2024
30	2	1	2025	31 January, 2025
31	2	2	1900	1 February, 1900

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
32	2	2	1901	1 February, 1901
33	2	2	1962	1 February, 1962
34	2	2	2024	1 February, 2024
35	2	2	2025	1 February, 2025
36	2	15	1900	14 February, 1900
37	2	15	1901	14 February, 1901
38	2	15	1962	14 February, 1962
39	2	15	2024	14 February, 2024
40	2	15	2025	14 February, 2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	31 May, 1900
52	6	1	1901	31 May, 1901
53	6	1	1962	31 May, 1962
54	6	1	2024	31 May, 2024
55	6	1	2025	31 May, 2025
56	6	2	1900	1 June, 1900
57	6	2	1901	1 June, 1901
58	6	2	1962	1 June, 1962
59	6	2	2024	1 June, 2024
60	6	2	2025	1 June, 2025
61	6	15	1900	14 June, 1900
62	6	15	1901	14 June, 1901
63	6	15	1962	14 June, 1962
64	6	15	2024	14 June, 2024
65	6	15	2025	14 June, 2025

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
66	6	30	1900	29 June, 1900
67	6	30	1901	29 June, 1901
68	6	30	1962	29 June, 1962
69	6	30	2024	29 June, 2024
70	6	30	2025	29 June, 2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	31 October, 1900
77	11	1	1901	31 October, 1901
78	11	1	1962	31 October, 1962
79	11	1	2024	31 October, 2024
80	11	1	2025	31 October, 2025
81	11	2	1900	1 November, 1900
82	11	2	1901	1 November, 1901

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
83	11	2	1962	1 November, 1962
84	11	2	2024	1 November, 2024
85	11	2	2025	1 November, 2025
86	11	15	1900	14 November, 1900
87	11	15	1901	14 November, 1901
88	11	15	1962	14 November, 1962
89	11	15	2024	14 November, 2024
90	11	15	2025	14 November, 2025
91	11	30	1900	29 November, 1900
92	11	30	1901	29 November, 1901
93	11	30	1962	29 November, 1962
94	11	30	2024	29 November, 2024
95	11	30	2025	29 November, 2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
101	12	1	1900	30 November, 1900
102	12	1	1901	30 November, 1901
103	12	1	1962	30 November, 1962
104	12	1	2024	30 November, 2024
105	12	1	2025	30 November, 2025
106	12	2	1900	1 December, 1900
107	12	2	1901	1 December, 1901
108	12	2	1962	1 December, 1962
109	12	2	2024	1 December, 2024
110	12	2	2025	1 December, 2025
111	12	15	1900	14 December, 1900
112	12	15	1901	14 December, 1901
113	12	15	1962	14 December, 1962
114	12	15	2024	14 December, 2024
115	12	15	2025	14 December, 2025
116	12	30	1900	29 December, 1900
117	12	30	1901	29 December, 1901
118	12	30	1962	29 December, 1962

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
119	12	30	2024	29 December, 2024
120	12	30	2025	29 December, 2025
121	12	31	1900	30 December, 1900
122	12	31	1901	30 December, 1901
123	12	31	1962	30 December, 1962
124	12	31	2024	30 December, 2024
125	12	31	2025	30 December, 2025

Example – 8.6

Consider the triangle problem as given in example 8.3. Generate robust and worst test cases for this problem.

Solution

Robust test cases are given on next slide.

Software Testing

`	x	y	z	Expected Output
1	50	50	0	Invalid input`
2	50	50	1	Isosceles
3	50	50	2	Isosceles
4	50	50	50	Equilateral
5	50	50	99	Isosceles
6	50	50	100	Not a triangle
7	50	50	101	Invalid input
8	50	0	50	Invalid input
9	50	1	50	Isosceles
10	50	2	50	Isosceles
11	50	99	50	Isosceles
12	50	100	50	Not a triangle
13	50	101	50	Invalid input
14	0	50	50	Invalid input
15	1	50	50	Isosceles
16	2	50	50	Isosceles
17	99	50	50	Isosceles
18	100	50	50	Not a triangle
19	100	50	50	Invalid input

Software Testing

Worst test cases are 125 and are given below:

Test Case	x	y	z	Expected output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	50	Not a triangle
4	1	1	99	Not a triangle
5	1	1	100	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	50	Not a triangle
9	1	2	99	Not a triangle
10	1	2	100	Not a triangle
11	1	50	1	Not a triangle
12	1	50	2	Not a triangle
13	1	50	50	Isosceles
14	1	50	99	Not a triangle

(Contd.)...

Software Testing

Test Case	A	b	c	Expected output
15	1	50	100	Not a triangle
16	1	99	1	Not a triangle
17	1	99	2	Not a triangle
18	1	99	50	Not a triangle
19	1	99	99	Isosceles
20	1	99	100	Not a triangle
21	1	100	1	Not a triangle
22	1	100	2	Not a triangle
23	1	100	50	Not a triangle
24	1	100	99	Not a triangle
25	1	100	100	Isosceles
26	2	1	1	Not a triangle
27	2	1	2	Isosceles
28	2	1	50	Not a triangle
29	2	1	99	Not a triangle
30	2	1	100	Not a triangle
31	2	2	1	Isosceles

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
32	2	2	2	Equilateral
33	2	2	50	Not a triangle
34	2	2	99	Not a triangle
35	2	2	100	Not a triangle
36	2	50	1	Not a triangle
37	2	50	2	Not a triangle
38	2	50	50	Isosceles
39	2	50	99	Not a triangle
40	2	50	100	Not a triangle
41	2	99	1	Not a triangle
42	2	99	2	Not a triangle
43	2	99	50	Not a triangle
44	2	99	99	Isosceles
45	2	99	100	Scalene
46	2	100	1	Not a triangle
47	2	100	2	Not a triangle
48	2	100	50	Not a triangle

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
49	2	100	50	Scalene
50	2	100	99	Isosceles
51	50	1	100	Not a triangle
52	50	1	1	Not a triangle
53	50	1	2	Isosceles
54	50	1	50	Not a triangle
55	50	1	99	Not a triangle
56	50	2	100	Not a triangle
57	50	2	1	Not a triangle
58	50	2	2	Isosceles
59	50	2	50	Not a triangle
60	50	2	99	Not a triangle
61	50	50	100	Isosceles
62	50	50	1	Isosceles
63	50	50	2	Equilateral
64	50	50	50	Isosceles
65	50	50	99	Not a triangle

(Contd.)...

Software Testing

Test Case	A	B	C	Expected output
66	50	99	1	Not a triangle
67	50	99	2	Not a triangle
68	50	99	50	Isosceles
69	50	99	99	Isosceles
70	50	99	100	Scalene
71	50	100	1	Not a triangle
72	50	100	2	Not a triangle
73	50	100	50	Not a triangle
74	50	100	99	Scalene
75	50	100	100	Isosceles
76	50	1	1	Not a triangle
77	99	1	2	Not a triangle
78	99	1	50	Not a triangle
79	99	1	99	Isosceles
80	99	1	100	Not a triangle
81	99	2	1	Not a triangle
82	99	2	2	Not a triangle

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
83	99	2	50	Not a triangle
84	99	2	99	Isosceles
85	99	2	100	Scalene
86	99	50	1	Not a triangle
87	99	50	2	Not a triangle
88	99	50	50	Isosceles
89	99	50	99	Isosceles
90	99	50	100	Scalene
91	99	99	1	Isosceles
92	99	99	2	Isosceles
93	99	99	50	Isosceles
94	99	99	99	Equilateral
95	99	99	100	Isosceles
96	99	100	1	Not a triangle
97	99	100	2	Scalene
98	99	100	50	Scalene
99	99	100	99	Isosceles
100	99	100	100	Isosceles

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
101	100	1	1	Not a triangle
102	100	1	2	Not a triangle
103	100	1	50	Not a triangle
104	100	1	99	Not a triangle
105	100	1	100	Isosceles
106	100	2	1	Not a triangle
107	100	2	2	Not a triangle
108	100	2	50	Not a triangle
109	100	2	99	Scalene
110	100	2	100	Isosceles
111	100	50	1	Not a triangle
112	100	50	2	Not a triangle
113	100	50	50	Not a triangle
114	100	50	99	Scalene
115	100	50	100	Isosceles
116	100	99	1	Not a triangle
117	100	99	2	Scalene
118	100	99	50	Scalene

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
119	100	99	99	Isosceles
120	100	99	100	Isosceles
121	100	100	1	Isosceles
122	100	100	2	Isosceles
123	100	100	50	Isosceles
124	100	100	99	Isosceles
125	100	100	100	Equilateral

Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

Fig. 7: Equivalence partitioning

Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

Example 8.7

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Identify the equivalence class test cases for output and input domains.

Solution

Output domain equivalence class test cases can be identified as follows:

O_1={<a,b,c>:Not a quadratic equation if a = 0}

O_1={<a,b,c>:Real roots if (b^2-4ac)>0}

O_1={<a,b,c>:Imaginary roots if (b^2-4ac)<0}

O_1={<a,b,c>:Equal roots if (b^2-4ac)=0}`

The number of test cases can be derived form above relations and shown below:

Test case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

Software Testing

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

Test Case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Example 8.8

Consider the program for determining the previous date in a calendar as explained in example 8.3. Identify the equivalence class test cases for output & input domains.

Solution

Output domain equivalence class are:

O_1={<D,M,Y>: Previous date if all are valid inputs}

O_1={<D,M,Y>: Invalid date if any input makes the date invalid}

Test case	M	D	Y	Expected output
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

Software Testing

We may have another set of test cases which are based on input domain.

I_1={month: $1 \leq m \leq 12$}

I_2={month: $m < 1$}

I_3={month: $m > 12$}

I_4={day: $1 \leq D \leq 31$}

I_5={day: $D < 1$}

I_6={day: $D > 31$}

I_7={year: $1900 \leq Y \leq 2025$}

I_8={year: $Y < 1900$}

I_9={year: $Y > 2025$}

Software Testing

Inputs domain test cases are :

Test Case	M	D	Y	Expected output
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	invalid input
6	6	32	1962	invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	invalid input (Value out of range)
9	6	15	2026	invalid input (Value out of range)

Example – 8.9

Consider the triangle problem specified in a example 8.3. Identify the equivalence class test cases for output and input domain.

Solution

Output domain equivalence classes are:

O_1={<*x*,*y*,*z*>: Equilateral triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Isosceles triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Scalene triangle with sides *x*,*y*,*z*}
O_1={<*x*,*y*,*z*>: Not a triangle with sides *x*,*y*,*z*}

The test cases are:

Test case	*x*	*y*	*z*	*Expected Output*
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Software Testing

Input domain based classes are:

$I_1=\{x: x < 1\}$

$I_2=\{x: x > 100\}$

$I_3=\{x: 1 \leq x \leq 100\}$

$I_4=\{y: y < 1\}$

$I_5=\{y: y > 100\}$

$I_6=\{y: 1 \leq y \leq 100\}$

$I_7=\{z: z < 1\}$

$I_8=\{z: z > 100\}$

$I_9=\{z: 1 \leq z \leq 100\}$

Software Testing

Some inputs domain test cases can be obtained using the relationship amongst x,y and z.

$I_{10}=\{< x,y,z >: x = y = z\}$

$I_{11}=\{< x,y,z >: x = y, x \neq z\}$

$I_{12}=\{< x,y,z >: x = z, x \neq y\}$

$I_{13}=\{< x,y,z >: y = z, x \neq y\}$

$I_{14}=\{< x,y,z >: x \neq y, x \neq z, y \neq z\}$

$I_{15}=\{< x,y,z >: x = y + z\}$

$I_{16}=\{< x,y,z >: x > y + z\}$

$I_{17}=\{< x,y,z >: y = x + z\}$

$I_{18}=\{< x,y,z >: y > x + z\}$

$I_{19}=\{< x,y,z >: z = x + y\}$

$I_{20}=\{< x,y,z >: z > x + y\}$

Test cases derived from input domain are:

Test case	*x*	*y*	*z*	*Expected Output*
1	0	50	50	Invalid input
2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles

(Contd.)...

Software Testing

Test case	x	y	z	Expected Output
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

Decision Table Based Testing

Condition Stub		Entry						
	C_1	True				False		
	C_2	True			False	True		False
	C_3	True	False	True	False	True	False	---
Action Stub	a_1	X	X			X		
	a_2	X		X			X	
	a_3		X			X		
	a_4				X		X	X

Table 2: Decision table terminology

Test case design

C_1:x,y,z are sides of a triangle?	N	Y							
C_2:x = y?	--	Y				N			
C_3:x = z?	--	Y		N		Y		N	
C_4:y = z?	--	Y	N	Y	N	Y	N	Y	N
a_1: Not a triangle	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

Table 3: Decision table for triangle problem

Conditions C_1 : x < y + z ?	F	T	T	T	T	T	T	T	T	T	T
C_2 : y < x + z ?	--	F	T	T	T	T	T	T	T	T	T
C_3 : z < x + y ?	--	--	F	T	T	T	T	T	T	T	T
C_4 : x = y ?	--	--	--	T	T	T	T	F	F	F	F
C_5 : x = z ?	--	--	--	T	T	F	F	T	T	F	F
C_6 : y = z ?	--	--	--	T	F	T	F	T	F	T	F
a_1 : Not a triangle	X	X	X								
a_2 : Scalene											X
a_3 : Isosceles							X		X	X	
a_4 : Equilateral				X							
a_5 : Impossible					X	X		X			

Table 4: Modified decision table

Example 8.10

Consider the triangle program specified in example 8.3. Identify the test cases using the decision table of Table 4.

Solution

There are eleven functional test cases, three to fail triangle property, three impossible cases, one each to get equilateral, scalene triangle cases, and three to get on isosceles triangle. The test cases are given in Table 5.

Test case	*x*	*y*	*z*	*Expected Output*
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Test cases of triangle problem using decision table

Example 8.11

Consider a program for the determination of Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs are “Previous date” and “Invalid date”. Design the test cases using decision table based testing.

Solution

The input domain can be divided into following classes:

I_1= {M_1: month has 30 days}

I_2= {M_2: month has 31 days except March, August and January}

I_3= {M_3: month is March}

I_4= {M_4: month is August}

I_5= {M_5: month is January}

I_6= {M_6: month is February}

I_7= {D_1: day = 1}

I_8= {D_2: 2 ≤ day ≤ 28}

I_9= {D_3: day = 29}

I_{10}={D_4: day = 30}

I_{11}={D_5: day = 31}

I_{12}={Y_1: year is a leap year}

I_{13}={Y_2: year is a common year}

The decision table is given below:

Sr.No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_1: Months in	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_2	M_2	M_2	M_2	M_2
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3
C_3: year in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1
a_1: Impossible									X	X					
a_2: Decrement day			X	X	X	X	X	X					X	X	X
a_3: Reset day to 31	X	X													
a_4: Reset day to 30											X	X			
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month	X	X									X	X			
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Sr.No.	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
C_1: Months in	M_2	M_2	M_2	M_2	M_2	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3
C_2: days in	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: year in	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible															
a_2: Decrement day	X	X	X	X	X			X	X	X	X	X	X	X	X
a_3: Reset day to 31															
a_4: Reset day to 30															
a_5: Reset day to 29						X									
a_6: Reset day to 28							X								
a_7: decrement month						X	X								
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Sr.No.	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
C_1: Months in	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_5	M_5	M_5	M_5	M_5
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3
C_3: year in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1
a_1: Impossible															
a_2: Decrement day			X	X	X	X	X	X	X	X			X	X	X
a_3: Reset day to 31	X	X									X	X			
a_4: Reset day to 30															
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month	X	X													
a_8: Reset month to December											X	X			
a_9: Decrement year											X	X			

Software Testing

Sr.No.	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
C_1: Months in	M_5	M_5	M_5	M_5	M_5	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6
C_2: days in	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: year in	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible											X	X	X	X	X
a_2: Decrement day	X	X	X	X	X			X	X	X					
a_3: Reset day to 31						X	X								
a_4: Reset day to 30															
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month						X	X								
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Test case	Month	Day	Year	Expected output
1	June	1	1964	31 May, 1964
2	June	1	1962	31 May, 1962
3	June	15	1964	14 June, 1964
4	June	15	1962	14 June, 1962
5	June	29	1964	28 June, 1964
6	June	29	1962	28 June, 1962
7	June	30	1964	29 June, 1964
8	June	30	1962	29 June, 1962
9	June	31	1964	Impossible
10	June	31	1962	Impossible
11	May	1	1964	30 April, 1964
12	May	1	1962	30 April, 1962
13	May	15	1964	14 May, 1964
14	May	15	1962	14 May, 1962
15	May	29	1964	28 May, 1964

Software Testing

Test case	Month	Day	Year	Expected output
16	May	29	1962	28 May, 1962
17	May	30	1964	29 May, 1964
18	May	30	1962	29 May, 1962
19	May	31	1964	30 May, 1964
20	May	31	1962	30 May, 1962
21	March	1	1964	29 February, 1964
22	March	1	1962	28 February, 1962
23	March	15	1964	14 March, 1964
24	March	15	1962	14 March, 1962
25	March	29	1964	28 March, 1964
26	March	29	1962	28 March, 1962
27	March	30	1964	29 March, 1964
28	March	30	1962	29 March, 1962
29	March	31	1964	30 March, 1964
30	March	31	1962	30 March, 1962

Software Testing

Test case	Month	Day	Year	Expected output
31	August	1	1964	31 July, 1962
32	August	1	1962	31 July, 1964
33	August	15	1964	14 August, 1964
34	August	15	1962	14 August, 1962
35	August	29	1964	28 August, 1964
36	August	29	1962	28 August, 1962
37	August	30	1964	29 August, 1964
38	August	30	1962	29 August, 1962
39	August	31	1964	30 August, 1964
40	August	31	1962	30 August, 1962
41	January	1	1964	31 December, 1964
42	January	1	1962	31 December, 1962
43	January	15	1964	14 January, 1964
44	January	15	1962	14 January, 1962
45	January	29	1964	28 January, 1964

Software Testing

Test case	Month	Day	Year	Expected output
46	January	29	1962	28 January, 1962
47	January	30	1964	29 January, 1964
48	January	30	1962	29 January, 1962
49	January	31	1964	30 January, 1964
50	January	31	1962	30 January, 1962
51	February	1	1964	31 January, 1964
52	February	1	1962	31 January, 1962
53	February	15	1964	14 February, 1964
54	February	15	1962	14 February, 1962
55	February	29	1964	28 February, 1964
56	February	29	1962	Impossible
57	February	30	1964	Impossible
58	February	30	1962	Impossible
59	February	31	1964	Impossible
60	February	31	1962	Impossible

Cause Effect Graphing Technique

- Consider single input conditions
- do not explore combinations of input circumstances

Steps

1. Causes & effects in the specifications are identified.

   A cause is a distinct input condition or an equivalence class of input conditions.

   An effect is an output condition or a system transformation.

2. The semantic content of the specification is analysed and transformed into a boolean graph linking the causes & effects.

3. Constraints are imposed

4. graph – limited entry decision table

   Each column in the table represent a test case.

5. The columns in the decision table are converted into test cases.

The basic notation for the graph is shown in fig. 8

Fig.8. 8 : Basic cause effect graph symbols

Software Testing

Myers explained this effectively with following example. “The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message *x* is issued. If the character in column 2 is not a digit, message y is issued”.

The causes are

c_1: character in column 1 is A

c_2: character in column 1 is B

c_3: character in column 2 is a digit

and the effects are

e_1: update made

e_2: message *x* is issued

e_3: message *y* is issued

Fig. 9: Sample cause effect graph

The **E** constraint states that it must always be true that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). The **I** constraint states that at least one of c_1, c_2 and c_3 must always be 1 (c_1, c_2 and c_3 cannot be 0 simultaneously). The **O** constraint states that one, and only one, of c_1 and c_2 must be 1. The constraint **R** states that, for c_1 to be 1, c_2 must be 1 (i.e. it is impossible for c_1 to be 1 and c_2 to be 0),

Software Testing

Fig. 10: Constraint symbols

Fig. 11: Symbol for masks constraint

Fig. 12 : Sample cause effect graph with exclusive constraint

Example 8.12

Consider the triangle problem specified in the example 8.3. Draw the Cause effect graph and identify the test cases.

Solution

The causes are

c_1: side x is less than sum of sides *y* and *z*

c_2: side y is less than sum of sides *x* and *y*

c_3: side z is less than sum of sides *x* and *y*

c_4: side x is equal to side *y*

c_5: side x is equal to side *z*

c_6: side y is equal to side *z*

and effects are

e_1: Not a triangle

e_2: Scalene triangle

e_3: Isosceles triangle

e_4: Equilateral triangle

e_5: Impossible stage

The cause effect graph is shown in fig. 13 and decision table is shown in table 6. The test cases for this problem are available in Table 5.

Conditions C_1: x < y + z ?	0	1	1	1	1	1	1	1	1	1	1
C_2: y < x + z ?	X	0	1	1	1	1	1	1	1	1	1
C_3: z < x + y ?	X	X	0	1	1	1	1	1	1	1	1
C_4: x = y ?	X	X	X	1	1	1	1	0	0	0	0
C_5: x = z ?	X	X	X	1	1	0	0	1	1	0	0
C_6: y = z ?	X	X	X	1	0	1	0	1	0	1	0
e_1: Not a triangle	1	1	1								
e_2: Scalene											1
e_3: Isosceles							1		1	1	
e_4: Equilateral				1							
e_5: Impossible					1	1		1			

Table 6: Decision table

Fig. 13: Cause effect graph of triangle problem

Structural Testing

A complementary approach to functional testing is called structural / white box testing. It permits us to examine the internal structure of the program.

Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness.

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in the set of program paths.

Flow Graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.

Fig. 14: The basic construct of the flow graph

Software Testing

```
/* Program to generate the previous date given a date, assumes data
given as dd mm yyyy separated by space and performs error checks on the
validity of the current date entered. */

#include <stdio.h>
#include <conio.h>

int main()
{
  int day, month, year, validDate = 0;
/*Date Entry*/
  printf("Enter the day value: ");
  scanf("%d", &day);
  printf("Enter the month value: ");
  scanf("%d", &month);
  printf("Enter the year value: ");
  scanf("%d",&year);
/*Check Date Validity */
  if (year >= 1900 && year <= 2025) {
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
      month == 8 || month == 10 || month == 12) {
```

(Contd.)...

```
        if (day >= 1 && day <= 31) {
            validDate = 1;
        }
        else {
          validDate = 0;
        }
      }
      else if (month == 2) {
        int rVal=0;
        if (year%4 == 0) {
          rVal=1;
          if ((year%100)==0 && (year % 400) !=0) {
            rVal=0;
          }
        }
        if (rVal ==1 && (day >=1 && day <=29) ) {
          validDate = 1;
        }
        else if (day >=1 && day <= 28 ) {
          validDate = 1;
        }
```

(Contd.)...

```
33          else {
34            validDate = 0;
35          }
36        }
37        else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
38          validDate = 1;
39        }
40        else {
41          validDate = 0;
42        }
43      }
        /*Prev Date Calculation*/
44      if (validDate) {
45        if (day == 1) {
46          if (month == 1) {
47            year--;
48            day=31;
49            month=12;
50          }
51          else if (month == 3) {
52            int rVal=0;
```

(Contd.)...

Software Testing

```
53          if (year%4 == 0) {
54            rVal=1;
55            if ((year%100)==0 && (year % 400) !=0) {
56          rVal=0;
57        }
58          }
59          if (rVal ==1) {
60            day=29;
61            month--;
62          }
63          else {
64            day=28;
65            month--;
66          }
67        }
68        else if (month == 2 || month == 4 || month == 6 || month == 9 ||
          month == 11) {
69          day = 31;
70          month--;
```

(Contd.)...

```
        }
        else {
          day=30;
          month--;
        }
      }
      else {
        day--;
      }
      printf("The next date is: %d-%d-%d",day,month,year);
    }
    else {
      printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
    }
    getche ();
    return 1;
  }
```

Fig. 15: Program for previous date problem

Fig. 16: Flow graph of previous date problem

DD Path Graph

Table 7: Mapping of flow graph nodes and DD path nodes

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 9	n_1	There is a sequential flow from node 1 to 9
10	n_2	Decision node, if true go to 13 else go to 44
11	n_3	Decision node, if true go to 12 else go to 19
12	n_4	Decision node, if true go to 13 else go to 15
13,14	n_5	Sequential nodes and are combined to form new node n_5
15,16,17	n_6	Sequential nodes
18	n_7	Edges from node 14 to 17 are terminated here
19	n_8	Decision node, if true go to 20 else go to 37
20	n_9	Intermediate node with one input edge and one output edge
21	n_{10}	Decision node, if true go to 22 else go to 27
22	n_{11}	Intermediate node
23	n_{12}	Decision node, if true go to 24 else go to 26

Cont.....

Flow graph nodes	DD Path graph corresponding node	Remarks
24,25	n_{13}	Sequential nodes
26	n_{14}	Two edges from node 25 & 23 are terminated here
27	n_{15}	Two edges from node 26 & 21 are terminated here. Also a decision node
28,29	n_{16}	Sequential nodes
30	n_{17}	Decision node, if true go to 31 else go to 33
31,32	n_{18}	Sequential nodes
33,34,35	n_{19}	Sequential nodes
36	n_{20}	Three edge from node 29,32 and 35 are terminated here
37	n_{21}	Decision node, if true go to 38 else go to 40
38,39	n_{22}	Sequential nodes
40,41,42	n_{23}	Sequential nodes
43	n_{24}	Three edge from node 36,39 and 42 are terminated here

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
44	n_{25}	Decision node, if true go to 45 else go to 82. Three edges from 18,43 & 10 are also terminated here.
45	n_{26}	Decision node, if true go to 46 else go to 77
46	n_{27}	Decision node, if true go to 47 else go to 51
47,48,49,50	n_{28}	Sequential nodes
51	n_{29}	Decision node, if true go to 52 else go to 68
52	n_{30}	Intermediate node with one input edge & one output ege
53	n_{31}	Decision node, if true go to 54 else go to 59
54	n_{32}	Intermediate node
55	n_{33}	Decision node, if true go to 56 else go to 58
56,57	n_{34}	Sequential nodes
58	n_{35}	Two edge from node 57 and 55 are terminated here
59	n_{36}	Decision node, if true go to 60 else go to 63. Two edge from nodes 58 and 53 are terminated.

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
60,61,62	n_{37}	Sequential nodes
63,64,65,66	n_{38}	Sequential nodes
67	n_{39}	Two edge from node 62 and 66 are terminated here
68	n_{40}	Decision node, if true go to 69 else go to 72
69,70,71	n_{41}	Sequential nodes
72,73,74,75	n_{42}	Sequential nodes
76	n_{43}	Four edges from nodes 50, 67, 71 and 75 are terminated here.
77,78,79	n_{44}	Sequential nodes
80	n_{45}	Two edges from nodes 76 & 79 are terminated here
81	n_{46}	Intermediate node
82,83,84	n_{47}	Sequential nodes
85	n_{48}	Two edges from nodes 81 and 84 are terminated here
86,87	n_{49}	Sequential nodes with exit node

Fig. 17: DD path graph of previous date problem

	Independent paths of previous date problem
1	$n_1, n_2, n_{25}, n_{47}, n_{48}, n_{49}$
2	$n_1, n_2, n_3, n_4, n_5, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
3	$n_1, n_2, n_3, n_4, n_6, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
4	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
5	$n_1, n_2, n_3, n_8, n_{21}, n_{23}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
6	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{19}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
7	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
8	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
9	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
10	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
11	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{44}, n_{45}, n_{46}, n_{48}, n_{49}$
12	$n_1, n_2, n_3, n_8, n_9, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{28}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
13	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{41}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
14	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{42}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
15	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{38}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
16	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
17	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{34}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
18	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$

Fig. 18: Independent paths of previous date problem

Example 8.13

Consider the problem for the determination of the nature of roots of a quadratic equation. Its input a triple of positive integers (say a,b,c) and value may be from interval [0,100].

The program is given in fig. 19. The output may have one of the following words:

[Not a quadratic equation; real roots; Imaginary roots; Equal roots]

Draw the flow graph and DD path graph. Also find independent paths from the DD Path graph.

Software Testing

```
#include <conio.h>
#include <math.h>
1       int main()
2     {
3       int a,b,c,validInput=0,d;
4       double D;
5       printf("Enter the 'a' value: ");
6       scanf("%d",&a);
7       printf("Enter the 'b' value: ");
8       scanf("%d",&b);
9       printf("Enter the 'c' value: ");
10      scanf("%d",&c);
11      if ((a >= 0) && (a <= 100) && (b >= 0) && (b <= 100) && (c >= 0)
          && (c <= 100)) {
12        validInput = 1;
13        if (a == 0) {
14          validInput = -1;
15        }
16      }
17      if (validInput==1) {
18        d = b*b - 4*a*c;
19        if (d == 0) {
20          printf("The roots are equal and are r1 = r2 = %f\n",
                  -b/(2*(float) a));
```

Cont….

```
21        }
22        else if ( d > 0 ) {
23          D=sqrt(d);
24          printf("The roots are real and are r1 = %f and r2 = %f\n",
                    (-b-D)/(2* a), (-b+D)/(2* a));
25        }
26        else {
27          D=sqrt(-d)/(2*a);
28          printf("The roots are imaginary and are r1 = (%f,%f) and
                    r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
29        }
30      }
31      else if (validInput == -1) {
32        printf("The vlaues do not constitute a Quadratic equation.");
33      }
34      else {
35        printf("The inputs belong to invalid range.");
36      }
37      getche();
38      return 1;
39    }
```

Fig. 19: Code of quadratic equation problem

Solution

Fig. 19 (a) : Program flow graph

Fig. 19 (b) : DD Path graph

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14,15	E	Sequential node
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20,21	J	Sequential node
22	K	Decision node
23,24,25	L	Sequential node

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
26,27,28,29	M	Sequential nodes
30	N	Three edges are combined
31	O	Decision node
32,33	P	Sequential node
34,35,36	Q	Sequential node
37	R	Three edges are combined here
38,39	S	Sequential nodes with exit node

Independent paths are:

(i) ABGOQRS
(ii) ABGOPRS
(iii) ABCDFGOQRS
(iv) ABCDEFGOPRS
(v) ABGHIJNRS
(vi) ABGHIKLNRS
(vi) ABGHIKMNRS

Example 8.14

Consider a program given in Fig.8.20 for the classification of a triangle. Its input is a triple of positive integers (say a,b,c) from the interval [1,100]. The output may be [Scalene, Isosceles, Equilateral, Not a triangle].

Draw the flow graph & DD Path graph. Also find the independent paths from the DD Path graph.

Software Testing

```
#include <stdio.h>
#include <conio.h>
1    int main()
2    {
3      int a,b,c,validInput=0;
4      printf("Enter the side 'a' value: ");
5      scanf("%d",&a);
6      printf("Enter the side 'b' value: ")
7      scanf("%d",&b);
8      printf("Enter the side 'c' value:");
9      scanf("%d",&c);
10     if ((a > 0) && (a <= 100) && (b > 0) && (b <= 100) && (c > 0)
           && (c <= 100)) {
11       if ( (a + b) > c) && ((c + a) > b) && ((b + c) > a)) {
12         validInput = 1;
13       }
14     }
15     else {
16       validInput = -1;
17     }
18     If (validInput==1) {
19       If ((a==b) && (b==c)) {
20         printf("The trinagle is equilateral");
21       }
22       else if ( (a == b) || (b == c) || (c == a) ) {
```

(Contd.)...

```
23          printf("The triangle is isosceles");
24        }
25        else {
26          printf("The trinagle is scalene");
27        }
28      }
29      else if (validInput == 0) {
30        printf("The values do not constitute a Triangle");
31      }
32      else {
33        printf("The inputs belong to invalid range");
34      }
35      getche();
36      return 1;
37    }
```

Fig. 20 : Code of triangle classification problem

Solution :

Flow graph of triangle problem is:

Fig.8. 20 (a): Program flow graph

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 TO 9	A	Sequential nodes
10	B	Decision node
11	C	Decision node
12, 13	D	Sequential nodes
14	E	Two edges are joined here
15, 16, 17	F	Sequential nodes
18	G	Decision nodes plus joining of two edges
19	H	Decision node
20, 21	I	Sequential nodes
22	J	Decision node
23, 24	K	Sequential nodes
25, 26, 27	L	Sequential nodes

Cont….

Flow graph nodes	DD Path graph corresponding node	Remarks
28	M	Three edges are combined here
29	N	Decision node
30, 31	O	Sequential nodes
32, 33, 34	P	Sequential nodes
35	Q	Three edges are combined here
36, 37	R	Sequential nodes with exit node

Fig. 20 (b): DD Path graph

DD Path graph is given in Fig. 20 (b)

Independent paths are:

(i) ABFGNPQR
(ii) ABFGNOQR
(iii) ABCEGNPQR
(iv) ABCDEGNOQR
(v) ABFGHIMQR
(vi) ABFGHJKMQR
(vii) ABFGHJMQR

Fig. 20 (b): DD Path graph

Cyclomatic Complexity

McCabe's cyclomatic metric V(G) = $e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*'.

Fig. 21: Flow graph

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 21.

Path 1 : *a c f*

Path 2 : *a b e f*

Path 3 : *a d c f*

Path 4 : *a b e a c f* **or** *a b e a b e f*

Path 5 : *a b e b e f*

Several properties of cyclomatic complexity are stated below:

1. V(G) ≥1
2. V (G) is the maximum number of independent paths in graph G.
3. Inserting & deleting functional statements to G does not affect V(G).
4. G has only one path if and only if V(G)=1.
5. Inserting a new row in G increases V(G) by unity.
6. V(G) depends only on the decision structure of G.

The role of P in the complexity calculation V(G)=e-n+2P is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 22.

Fig. 22

Let us denote the total graph above with 3 connected components as

$$V(M \cup A \cup B) = e - n + 2P$$

$$= 13-13+2*3$$

$$= 6$$

This method with P ≠ 1 can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

Notice that $V(M \cup A \cup B) = V(M) + V(A) + V(B) = 6$. In general, the complexity of a collection C of flow graphs with K connected components is equal to the summation of their complexities. To see this let C_i,1 ≤ I ≤ K denote the k distinct connected component, and let e_i and n_i be the number of edges and nodes in the ith-connected component. Then

$$V(C) = e - n + 2p = \sum_{i=1}^{k} e_i - \sum_{i=1}^{k} n_i + 2K$$

$$= \sum_{i=1}^{k} (e_i - n_i + 2) = \sum_{i=1}^{k} V(C_i)$$

Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (decision) nodes plus one.

   $V(G)=\Pi+1$

   Where Π is the number of predicate nodes contained in the flow graph G.

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

Example 8.15

Consider a flow graph given in Fig. 23 and calculate the cyclomatic complexity by all three methods.

Fig. 23

Solution

Cyclomatic complexity can be calculated by any of the three methods.

1. V(G) $= e - n + 2P$

   $= 13 - 10 + 2 = 5$

2. V(G) $= \pi + 1$

   $= 4 + 1 = 5$

3. V(G) = number of regions

   $= 5$

Therefore, complexity value of a flow graph in Fig. 23 is 5.

Example 8.16

Consider the previous date program with DD path graph given in Fig. 17. Find cyclomatic complexity.

Solution

Number of edges (e) = 65

Number of nodes (n) =49

(i) $V(G) = e - n + 2P = 65 - 49 + 2 = 18$

(ii) $V(G) = \pi + 1 = 17 + 1 = 18$

(iii) V(G) = Number of regions = 18

The cyclomatic complexity is 18.

Example 8.17

Consider the quadratic equation problem given in example 8.13 with its DD Path graph. Find the cyclomatic complexity:

Solution

Number of nodes (n) = 19

Number of edges (e) = 24

(i) $V(G) = e - n + 2P = 24 - 19 + 2 = 7$

(ii) $V(G) = \pi + 1 = 6 + 1 = 7$

(iii) V(G) = Number of regions = 7

Hence cyclomatic complexity is 7 meaning thereby, seven independent paths in the DD Path graph.

Example 8.18

Consider the classification of triangle problem given in example 8.14. Find the cyclomatic complexity.

Solution

Number of edges (e) = 23

Number of nodes (n) =18

(i) $V(G) = e - n + 2P = 23 - 18 + 2 = 7$

(ii) $V(G) = \pi + 1 = 6 + 1 = 7$

(iii) V(G) = Number of regions = 7

The cyclomatic complexity is 7. Hence, there are seven independent paths as given in example 8.14.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig. 24.

Flow graph

	1	2	3	4
1			a	c
2				
3		b		
4		d		

Graph Matrix

(a)

Fig. 24 (a): Flow graph and graph matrices

(Contd.)...

Flow graph

(b)

	1	2	3
1	a		d
2	b + e		
3		c	

Graph Matrix

Fig. 24 (b): Flow graph and graph matrices

(Contd.)...

Flow graph

	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Graph Matrix

(c)

Fig. 24 (c): Flow graph and graph matrices

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Fig. 25 : Connection matrix of flow graph shown in Fig. 24 (c)

	1	2	3	4
1			a	c
2				
3		b		
4		d		

[A]

	1	2	3	4
1		ab + cd		
2				
3				
4				

$[A]^2$

The square matrix represent that there are two path *ab* and *cd* from node 1 to node 2.

Example 8.19

Consider the flow graph shown in the Fig. 26 and draw the graph & connection matrices. Find out cyclomatic complexity and two / three link paths from a node to any other node.

Fig. 26 : Flow graph

Solution

The graph & connection matrices are given below :

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

Graph Matrix (A)

	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1		1	3 – 1 = 2
						4 + 1 = 5

Connection Matrix

To find two link paths, we have to generate a square of graph matrix [A] and for three link paths, a cube of matrix [A] is required.

Software Testing

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed + hg	he	eb	h^2

$[A^2]$

	1	2	3	4	5
1		abc			afb
2					
3		bfg	bfe		bfh
4		fed + fhg	fhe	feb	fh^2
5		ebc + hed + h^2g	h^2e	heb	ebf + h^3

$[A^3]$

Data Flow Testing

Data flow testing is another from of structural testing. It has nothing to do with data flow diagrams.

i. Statements where variables receive values.

ii. Statements where these values are used or referenced.

As we know, variables are defined and referenced throughout the program. We may have few define/ reference anomalies:

i. A variable is defined but not used/ referenced.

ii. A variable is used but never defined.

iii. A variable is defined twice before it is used.

Definitions

The definitions refer to a program P that has a program graph G(P) and a set of program variables V. The G(P) has a single entry node and a single exit node. The set of all paths in P is PATHS(P)

(i) **Defining Node:** Node n ϵ G(P) is a defining node of the variable *v* ϵ V, written as DEF (*v*, *n*), if the value of the variable *v* is defined at the statement fragment corresponding to node *n*.

(ii) **Usage Node:** Node n ϵ G(P) is a usage node of the variable *v* ϵ V, written as USE (*v*, *n*), if the value of the variable *v* is used at statement fragment corresponding to node *n*. A usage node USE (*v*, n) is a predicate use (denote as p) if statement n is a predicate statement otherwise USE (*v*, n) is a computation use (denoted as *c*).

(iii) Definition use: A definition use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some $v \in V$, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are initial and final nodes of the path.

(iv) Definition clear : A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in PATHS(P) with initial and final nodes DEF (*v*, m) and USE (*v*, n), such that no other node in the path is a defining node of *v*.

The du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. The du-paths that are not definition clear are potential trouble spots.

Hence, our objective is to find all du-paths and then identity those du-paths which are not dc-paths. The steps are given in Fig. 27. We may like to generate specific test cases for du-paths that are not dc-paths.

Fig. 27 : Steps for data flow testing

Example 8.20

Consider the program of the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values for each of these may be from interval [0,100]. The program is given in Fig. 19. The output may have one of the option given below:

(i) Not a quadratic program

(ii) real roots

(iii) imaginary roots

(iv) equal roots

(v) invalid inputs

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 19 (a). The variables used in the program are a,b,c,d, validinput, D.

Step II: DD Path graph is given in Fig. 19(b). The cyclomatic complexity of this graph is 7 indicating there are seven independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	11,13,18,20,24,27,28
b	8	11,18,20,24,28
c	10	11,18
d	18	19,22,23,27
D	23, 27	24,28
Validinput	3, 12, 14	17,31

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 19 (a).

Variable	Path (beginning, end) nodes	Definition clear ?
a	6, 11 6, 13 6, 18 6, 20 6, 24 6, 27 6, 28	Yes Yes Yes Yes Yes Yes Yes
b	8, 11 8, 18 8, 20 8, 24 8, 28	Yes Yes Yes Yes Yes

(Contd.)...

Variable	Path (beginning, end) nodes	Definition clear ?
c	10, 11 10, 18	Yes Yes
d	18, 19 18, 22 18, 23 18, 27	Yes Yes Yes Yes
D	23, 24 23, 28 27, 24 27, 28	Yes Path not possible Path not possible Yes
validinput	3, 17 3, 31 12, 17 12, 31 14, 17 14, 31	no no no no yes yes

Example 8.21

Consider the program given in Fig. 20 for the classification of a triangle. Its input is a triple of positive integers (say a,b,c) from the interval [1,100]. The output may be:

[Scalene, Isosceles, Equilateral, Not a triangle, Invalid inputs].

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 20 (a). The variables used in the program are a,b,c, valid input.

Step II: DD Path graph is given in Fig. 20(b). The cyclomatic complexity of this graph is 7 and thus, there are 7 independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	10, 11, 19, 22
b	7	10, 11, 19, 22
c	9	10, 11, 19, 22
valid input	3, 13, 16	18, 29

Step IV: The du-paths are identified and are named by their beginning and ending nodes using Fig. 20 (a).

Variable	Path (beginning, end) nodes	Definition clear ?
a	5, 10 5, 11 5, 19 5, 22	Yes Yes Yes Yes
b	7, 10 7, 11 7, 19 7, 22	Yes Yes Yes Yes

(Contd.)...

Variable	Path (beginning, end) nodes	Definition clear ?
c	9, 10 9, 11 9, 19 9, 22	Yes Yes Yes Yes
valid input	3, 18 3, 29 12, 18 12, 29 16, 18 16, 29	no no no no Yes Yes

Hence total du-paths are 18 out of which four paths are not definition clear

Mutation Testing

Mutation testing is a fault based technique that is similar to fault seeding, except that mutations to program statements are made in order to determine properties about test cases. it is basically a fault simulation technique.

Multiple copies of a program are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program.

A mutant that is detected by a test case is termed “killed” and the goal of mutation procedure is to find a set of test cases that are able to kill groups of mutant programs.

When we mutate code there needs to be a way of measuring the degree to which the code has been modified. For example, if the original expression is x+1 and the mutant for that expression is x+2, that is a lesser change to the original code than a mutant such as (c*22), where both the operand and the operator are changed. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. High order mutants becomes intractable and thus in practice only low order mutants are used.

One difficulty associated with whether mutants will be killed is the problem of reaching the location; if a mutant is not executed, it cannot be killed. Special test cases are to be designed to reach a mutant. For example, suppose, we have the code.

```
Read (a,b,c);
If(a>b) and (b=c) then
x:=a*b*c; (make mutants; m1, m2, m3 .......)
```

To execute this, input domain must contain a value such that a is greater than b and b equals c. If input domain does not contain such a value, then all mutants made at this location should be considered equivalent to the original program, because the statement x:=a*b*c is dead code (code that cannot be reached during execution). If we make the mutant x+y for x+1, then we should take care about the value of y which should not be equal to 1 for designing a test case.

The manner by which a test suite is evaluated (scored) via mutation testing is as follows: for a specified test suite and a specific set of mutants, there will be three types of mutants in the code i.e., killed or dead, live, equivalent. The sum of the number of live, killed, and equivalent mutants will be the total number of mutants created. The score associated with a test suite T and mutants M is simply.

$$\frac{\#killed}{\#total - \#equivalent} \times 100\%$$

Levels of Testing

There are 3 levels of testing:

i. Unit Testing

ii. Integration Testing

iii. System Testing

UNIT TESTING

INTEGRATION TESTING

SYSTEM TESTING

Unit Testing

There are number of reasons in support of unit testing than testing the entire product.

1. The size of a single module is small enough that we can locate an error fairly easily.

2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.

3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

Software Testing

There are problems associated with testing a module in isolation. How do we run a module without anything to call it, to be called by it or, possibly, to output intermediate values obtained during execution? One approach is to construct an appropriate driver routine to call if and, simple stubs to be called by it, and to insert output statements in it.

Stubs serve to replace modules that are subordinate to (called by) the module 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.

This overhead code, called scaffolding represents effort that is import to testing, but does not appear in the delivered product as shown in Fig. 29.

Fig. 29 : Scaffolding required testing a program unit (module)

Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization.

Fig. 30 : Three different integration approaches

System Testing

Of the three levels of testing, the system level is closet to everyday experiences. We test many things; a used car before we buy it, an on-line cable network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, goal is not to find faults, but to demonstrate performance. Because of this we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and is compounded by the reduced testing interval that usually remains before a delivery deadline.

Petschenik gives some guidelines for choosing test cases during system testing.

Software Testing

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 31. These represent the operational correctness of the product and may be part of the software specifications.

Usable	Is the product convenient, clear, and predictable?
Secure	Is access to sensitive data restricted to those with authorization?
Compatible	Will the product work correctly in conjunction with existing data, software, and procedures?
Dependable	Do adequate safeguards against failure and methods for recovery exist in the product?
Documented	Are manuals complete, correct, and understandable?

Fig. 31 : Attributes of software to be tested during system testing

Validation Testing

- It refers to test the software as a complete product.
- This should be done after unit & integration testing.
- Alpha, beta & acceptance testing are nothing but the various ways of involving customer during testing.

Validation Testing

- IEEE has developed a standard (IEEE standard 1059-1993) entitled " IEEE guide for software verification and validation " to provide specific guidance about planning and documenting the tasks required by the standard so that the customer may write an effective plan.
- Validation testing improves the quality of software product in terms of functional capabilities and quality attributes.

The Art of Debugging

The goal of testing is to identify errors (bugs) in the program. The process of testing generates symptoms, and a program's failure is a clear symptom of the presence of an error. After getting a symptom, we begin to investigate the cause and place of that error. After identification of place, we examine that portion to identify the cause of the problem. This process is called debugging.

Debugging Techniques

Pressman explained few characteristics of bugs that provide some clues.

1. "The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located in other part. Highly coupled program structures may complicate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.

Software Testing

3. The symptom may actually be caused by non errors (e.g. round off inaccuracies).

4. The symptom may be caused by a human error that is not easily traced.

5. The symptom may be a result of timing problems rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded system that couple hardware with software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors".

Induction approach

- Locate the pertinent data
- Organize the data
- Devise a hypothesis
- Prove the hypothesis

Fig. 32 : The inductive debugging process

Deduction approach

- Enumerate the possible causes or hypotheses
- Use the data to eliminate possible causes
- Refine the remaining hypothesis
- Prove the remaining hypothesis

Fig. 33 : The inductive debugging process

Testing Tools

One way to improve the quality & quantity of testing is to make the process as pleasant as possible for the tester. This means that tools should be as concise, powerful & natural as possible.

The two broad categories of software testing tools are :

- Static
- Dynamic

Software Testing

There are different types of tools available and some are listed below:

1. Static analyzers, which examine programs systematically and automatically.
2. Code inspectors, who inspect programs automatically to make sure they adhere to minimum quality standards.
3. standards enforcers, which impose simple rules on the developer.
4. Coverage analysers, which measure the extent of coverage.
5. Output comparators, used to determine whether the output in a program is appropriate or not.

Software Testing

6. Test file/ data generators, used to set up test inputs.

7. Test harnesses, used to simplify test operations.

8. Test archiving systems, used to provide documentation about programs.

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

8.1 Software testing is:

(a) the process of demonstrating that errors are not present

(b) the process of establishing confidence that a program does what it is supposed to do

(c) the process of executing a program to show it is working as per specifications

(d) the process of executing a program with the intent of finding errors

8.2 Software mistakes during coding are known as:

(a) failures (b) defects

(c) bugs (d) errors

8.3 Functional testing is known as:

(a) Structural testing (b) Behavior testing

(c) Regression testing (d) None of the above

8.4 For a function of n variables, boundary value analysis yields:

(a) 4n+3 test cases (b) 4n+1 test cases

(c) n+4 test cases (d) None of the above

Multiple Choice Questions

8.5 For a function of two variables, how many cases will be generated by robustness testing?

(a) 9 (b) 13

(c) 25 (d) 42

8.6 For a function of n variables robustness testing of boundary value analysis yields:

(a) 4n+1 (b) 4n+3

(c) 6n+1 (d) None of the above

8.7 Regression testing is primarily related to:

(a) Functional testing (b) Data flow testing

(c) Development testing (d) Maintenance testing

8.8 A node with indegree=0 and out degree $\neq 0$ is called

(a) Source node (b) Destination node

(c) Transfer node (d) None of the above

8.9 A node with indegree $\neq 0$ and out degree=0 is called

(a) Source node (b) Predicate node

(c) Destination node (d) None of the above

Multiple Choice Questions

8.10 A decision table has

(a) Four portions (b) Three portions
(c) Five portions (d) Two portions

8.11 Beta testing is carried out by

(a) Users (b) Developers
(c) Testers (d) All of the above

8.12 Equivalence class partitioning is related to

(a) Structural testing (b) Blackbox testing
(c) Mutation testing (d) All of the above

8.13 Cause effect graphing techniques is one form of

(a) Maintenance testing (b) Structural testing
(c) Function testing (d) Regression testing

8.14 During validation

(a) Process is checked (b) Product is checked
(c) Developer's performance is evaluated (d) The customer checks the product

Multiple Choice Questions

8.15 Verification is

(a) Checking the product with respect to customer's expectation

(b) Checking the product with respect to specifications

(c) Checking the product with respect to the constraints of the project

(d) All of the above

8.16 Validation is

(a) Checking the product with respect to customer's expectation

(b) Checking the product with respect to specifications

(c) Checking the product with respect to the constraints of the project

(d) All of the above

8.17 Alpha testing is done by

(a) Customer (b) Tester

(c) Developer (d) All of the above

8.18 Site for Alpha testing is

(a) Software company (b) Installation place

(c) Any where (d) None of the above

Multiple Choice Questions

8.19 Site for Beta testing is

(a) Software company (b) User's site

(c) Any where (d) All of the above

8.20 Acceptance testing is done by

(a) Developers (b) Customers

(c) Testers (d) All of the above

8.21 One fault may lead to

(a) One failure (b) No failure

(c) Many failure (d) All of the above

8.22 Test suite is

(a) Set of test cases (b) Set of inputs

(c) Set of outputs (d) None of the above

8.23 Behavioral specification are required for:

(a) Modeling (b) Verification

(c) Validation (d) None of the above

Multiple Choice Questions

8.24 During the development phase, the following testing approach is not adopted

(a) Unit testing
(b) Bottom up testing
(c) Integration testing
(d) Acceptance testing

8.25 Which is not a functional testing technique?

(a) Boundary value analysis
(b) Decision table
(c) Regression testing
(d) None of the above

8.26 Decision table are useful for describing situations in which:

(a) An action is taken under varying sets of conditions.
(b) Number of combinations of actions are taken under varying sets of conditions
(c) No action is taken under varying sets of conditions
(d) None of the above

8.27 One weakness of boundary value analysis and equivalence partitioning is

(a) They are not effective
(b) They do not explore combinations of input circumstances
(c) They explore combinations of input circumstances
(d) None of the above

Multiple Choice Questions

8.28 In cause effect graphing technique, cause & effect are related to

(a) Input and output
(b) Output and input
(c) Destination and source
(d) None of the above

8.29 DD path graph is called as

(a) Design to Design Path graph
(b) Defect to Defect Path graph
(c) Destination to Destination Path graph
(d) Decision to decision Path graph

8.30 An independent path is

(a) Any path through the DD path graph that introduce at least one new set of processing statements or new conditions

(b) Any path through the DD path graph that introduce at most one new set of processing statements or new conditions

(c) Any path through the DD path graph that introduce at one and only one new set of processing statements or new conditions

(d) None of the above

8.31 Cyclomatic complexity is developed by

(a) B.W.Boehm
(b) T.J.McCabe
(c) B.W.Lettlewood
(d) Victor Basili

Multiple Choice Questions

8.32 Cyclomatic complexity is denoted by

(a) V(G)=e-n+2P (b) V(G)= ∏ +1

(c) V(G)=Number of regions of the graph (d) All of the above

8.33 The equation V(G)= ∏ +1 of cyclomatic complexity is applicable only if every predicate node has

(a) two outgoing edges (b) three or more outgoing edges

(c) no outgoing edges (d) none of the above

8.34 The size of the graph matrix is

(a) Number of edges in the flow graph

(b) Number of nodes in the flow graph

(c) Number of paths in the flow graph

(d) Number of independent paths in the flow graph

Multiple Choice Questions

8.35 Every node is represented by

(a) One row and one column in graph matrix

(b) Two rows and two columns in graph matrix

(c) One row and two columns in graph matrix

(d) None of the above

8.36 Cyclomatic complexity is equal to

(a) Number of independent paths (b) Number of paths

(c) Number of edges (d) None of the above

8.37 Data flow testing is related to

(a) Data flow diagrams (b) E-R diagrams

(c) Data dictionaries (d) none of the above

8.38 In data flow testing, objective is to find

(a) All dc-paths that are not du-paths (b) All du-paths

(c) All du-paths that are not dc-paths (d) All dc-paths

Multiple Choice Questions

8.39 Mutation testing is related to

(a) Fault seeding
(b) Functional testing
(c) Fault checking
(d) None of the above

8.40 The overhead code required to be written for unit testing is called

(a) Drivers
(b) Stubs
(c) Scaffolding
(d) None of the above

8.41 Which is not a debugging techniques

(a) Core dumps
(b) Traces
(c) Print statements
(d) Regression testing

8.42 A break in the working of a system is called

(a) Defect
(b) Failure
(c) Fault
(d) Error

8.43 Alpha and Beta testing techniques are related to

(a) System testing
(b) Unit testing
(c) acceptance testing
(d) Integration testing

Multiple Choice Questions

8.44 Which one is not the verification activity

(a) Reviews
(b) Path testing
(c) Walkthrough
(d) Acceptance testing

8.45 Testing the software is basically

(a) Verification
(b) Validation
(c) Verification and validation
(d) None of the above

8.46 Integration testing techniques are

(a) Topdown
(b) Bottom up
(c) Sandwich
(d) All of the above

8.47 Functionality of a software is tested by

(a) White box testing
(b) Black box testing
(c) Regression testing
(d) None of the above

8.48 Top down approach is used for

(a) Development
(b) Identification of faults
(c) Validation
(d) Functional testing

Multiple Choice Questions

8.49 Thread testing is used for testing

(a) Real time systems
(b) Object oriented systems
(c) Event driven systems
(d) All of the above

8.50 Testing of software with actual data and in the actual environment is called

(a) Alpha testing
(b) Beta testing
(c) Regression testing
(d) None of the above

Exercises

8.1 What is software testing? Discuss the role of software testing during software life cycle and why is it so difficult?

8.2 Why should we test? Who should do the testing?

8.3 What should we test? Comment on this statement. Illustrate the importance of testing

8.4 Defined the following terms:

(i) fault (ii) failure

(iii) bug (iv) mistake

8.5 What is the difference between

(i) Alpha testing & beta testing

(ii) Development & regression testing

(iii) Functional & structural testing

8.6 Discuss the limitation of testing. Why do we say that complete testing is impossible?

Exercises

8.7 Briefly discuss the following

(i) Test case design, Test & Test suite

(ii) Verification & Validation

(iii) Alpha, beta & acceptance testing

8.8 Will exhaustive testing (even if possible for every small programs) guarantee that the program is 100% correct?

8.9 Why does software fail after it has passed from acceptance testing? Explain.

8.10 What are various kinds of functional testing? Describe any one in detail.

8.11 What is a software failure? Explain necessary and sufficient conditions for software failure. Mere presence of faults means software failure. Is it true? If not, explain through an example, a situation in which a failure will definitely occur.

8.12 Explain the boundary value analysis testing techniques with the help of an example.

Exercises

8.13 Consider the program for the determination of next date in a calendar. Its input is a triple of day, month and year with the following range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \; 1 \leq \text{year} \leq 2025$

The possible outputs would be Next date or invalid date. Design boundary value, robust and worst test cases for this programs.

8.14 Discuss the difference between worst test case and adhoc test case performance evaluation by means of testing. How can we be sure that the real worst case has actually been observed?

8.15 Describe the equivalence class testing method. Compare this with boundary value analysis techniques

Exercises

8.16 Consider a program given below for the selection of the largest of numbers

```
main()
   {
     float A,B,C;
     printf("Enter three values\n");
     scanf("%f%f%f",  &A,&B,&c);
     printf("\n Largest value is");
     if (A>B)
          {
               if(A>C)
                    printf("%f\n",A);
               else
                    printf("%f\n",c);
          }
          else
          {
               if(C>B)
                    printf("%f\n",C);
               else
                    printf("%f\n",B);
          }
     }
```

Exercises

(i) Design the set of test cases using boundary value analysis technique and equivalence class testing technique.

(ii) Select a set of test cases that will provide 100% statement coverage.

(iii) Develop a decision table for this program.

8.17 Consider a small program and show, why is it practically impossible to do exhaustive testing?

8.18 Explain the usefulness of decision table during testing. Is it really effective? Justify your answer.

8.19 Draw the cause effect graph of the program given in exercise 8.16.

8.20 Discuss cause effect graphing technique with an example.

8.21 Determine the boundary value test cases the extended triangle problem that also considers right angle triangles.

Exercises

8.22 Why does software testing need extensive planning? Explain.

8.23 What is meant by test case design? Discuss its objectives and indicate the steps involved in test case design.

8.24 Let us consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Marks obtained	*Grade*
80–100	Distinction
60–79	First division
50–59	Second division
40–49	Third division
0–39	Fail

Generate test cases using equivalence class testing technique

Exercises

8.25 Consider a program to determine whether a number is 'odd' or 'even' and print the message

NUMBER IS EVEN

Or

NUMBER IS ODD

The number may be any valid integer.

Design boundary value and equivalence class test cases.

8.26 Admission to a professional course is subject to the following conditions:

(*a*) Marks in Mathematics >=	60
(*b*) Marks in Physics >=	50
(*c*) Marks in Chemistry >=	40
(*d*) Total in all three subjects >=	200
Or	
Total in Mathematics and Physics >=	150

Exercises

If aggregate marks of an eligible candidate are more than 225, he/she will be eligible for honors course, otherwise he/she will be eligible for pass course. The program reads the marks in the three subjects and generates the following outputs:

(a) Not Eligible

(b) Eligible to Pass Course

(c) Eligible to Honors Course

Design test cases using decision table testing technique.

8.27 Draw the flow graph for program of largest of three numbers as shown in exercise 8.16. Find out all independent paths that will guarantee that all statements in the program have been tested.

8.28 Explain the significance of independent paths. Is it necessary to look for a tool for flow graph generation, if program size increases beyond 100 source lines?

8.29 Discuss the structure testing. How is it different form functional testing?

Exercises

8.30 What do you understand by structural testing? Illustrate important structural testing techniques.

8.31 Discuss the importance of path testing during structural testing.

8.32 What is cyclomatic complexity? Explain with the help of an example.

8.33 Is it reasonable to define "thresholds" for software modules? For example, is a module acceptable if its $V(G) \leq 10$? Justify your answer.

8.34 Explain data flow testing. Consider an example and show all "du" paths. Also identify those "du" paths that are not "dc" paths.

8.35 Discuss the various steps of data flow testing.

8.36 If we perturb a value, changing the current value of 100 by 1000, what is the effect of this change? What precautions are required while designing the test cases?

Exercises

8.37 What is the difference between white and black box testing? Is determining test cases easier in back or white box testing? Is it correct to claim that if white box testing is done properly, it will achieve close to 100% path coverage?

8.38 What are the objectives of testing? Why is the psychology of a testing person important?

8.39 Why does software fail after it has passed all testing phases? Remember, software, unlike hardware does not wear out with time.

8.40 What is the purpose of integration testing? How is it done?

8.41 Differentiate between integration testing and system testing.

8.42 Is unit testing possible or even desirable in all circumstances? Provide examples to Justify your answer?

8.43 Peteschenik suggested that a different team than the one that does integration testing should carry out system testing. What are some good reasons for this?

Exercises

8.44 Test a program of your choice, and uncover several program errors. Localise the main route of these errors, and explain how you found the courses. Did you use the techniques of Table 8? Explain why or why not.

8.45 How can design attributes facilitate debugging?

8.46 List some of the problem that could result from adding debugging statements to code. Discuss possible solutions to these problems.

8.47 What are various debugging approaches? Discuss them with the help of examples.

8.48 Researchers and practitioners have proposed several mixed testing strategies intended to combine advantages of various techniques discussed in this chapter. Propose your own combination, perhaps also using some kind of random testing at selected points.

8.49 Design a test set for a spell checker. Then run it on a word processor having a spell checker, and report on possible inadequacies with respect to your requirements.

Exercises

8.50 4 GLs represent a major step forward in the development of automatic program generation. Explain the major advantage & disadvantage in the use of 4 GLs. What are the cost impact of applications of testing and how do you justify expenditures for these activities.

Software Maintenance

Software Maintenance

What is Software Maintenance?

Software Maintenance is a very broad activity that includes error corrections, enhancements of capabilities, deletion of obsolete capabilities, and optimization.

Software Maintenance

Categories of Maintenance

- Corrective maintenance

This refer to modifications initiated by defects in the software.

- Adaptive maintenance

It includes modifying the software to match changes in the ever changing environment.

- Perfective maintenance

It means improving processing efficiency or performance, or restructuring the software to improve changeability. This may include enhancement of existing system functionality, improvement in computational efficiency etc.

Software Maintenance

- Other types of maintenance

There are long term effects of corrective, adaptive and perfective changes. This leads to increase in the complexity of the software, which reflect deteriorating structure. The work is required to be done to maintain it or to reduce it, if possible. This work may be named as preventive maintenance.

Software Maintenance

Fig. 1: Distribution of maintenance effort

Software Maintenance

Problems During Maintenance

- Often the program is written by another person or group of persons.
- Often the program is changed by person who did not understand it clearly.
- Program listings are not structured.
- High staff turnover.
- Information gap.
- Systems are not designed for change.

Maintenance is Manageable

A common misconception about maintenance is that it is not manageable.

Report of survey conducted by Lientz & Swanson gives some interesting observations:

1	Emergency debugging	12.4%
2	Routine debugging	9.3%
3	Data environment adaptation	17.3%
4	Changes in hardware and OS	6.2%
5	Enhancements for users	41.8%
6	Documentation Improvement	5.5%
7	Code efficiency improvement	4.0%
8	Others	3.5%

Table 1: Distribution of maintenance effort

Software Maintenance

Kinds of maintenance requests

1	New reports	40.8%
2	Add data in existing reports	27.1%
3	Reformed reports	10%
4	Condense reports	5.6%
5	Consolidate reports	6.4%
6	Others	10.1%

Table 2: Kinds of maintenance requests

Software Maintenance

Potential Solutions to Maintenance Problems

- Budget and effort reallocation
- Complete replacement of the system
- Maintenance of existing system

Software Maintenance

The Maintenance Process

Fig. 2: The software maintenance process

Software Maintenance

- Program Understanding

The first phase consists of analyzing the program in order to understand.

- Generating Particular Maintenance Proposal

The second phase consists of generating a particular maintenance proposal to accomplish the implementation of the maintenance objective.

- Ripple Effect

The third phase consists of accounting for all of the ripple effect as a consequence of program modifications.

Software Maintenance

- Modified Program Testing

The fourth phase consists of testing the modified program to ensure that the modified program has at least the same reliability level as before.

- Maintainability

Each of these four phases and their associated software quality attributes are critical to the maintenance process. All of these factors must be combined to form maintainability.

Maintenance Models

- Quick-fix Model

This is basically an adhoc approach to maintaining software. It is a fire fighting approach, waiting for the problem to occur and then trying to fix it as quickly as possible.

Fig. 3: The quick-fix model

Software Maintenance

- Iterative Enhancement Model
  - Analysis
  - Characterization of proposed modifications
  - Redesign and implementation

Software Maintenance

Fig. 4: The three stage cycle of iterative enhancement

Software Maintenance

- Reuse Oriented Model

The reuse model has four main steps:

1. Identification of the parts of the old system that are candidates for reuse.
2. Understanding these system parts.
3. Modification of the old system parts appropriate to the new requirements.
4. Integration of the modified parts into the new system.

Software Maintenance

Fig. 5: The reuse model

Software Maintenance

- Boehm's Model

Boehm proposed a model for the maintenance process based upon the economic models and principles.

Boehm represent the maintenance process as a closed loop cycle.

Software Maintenance

Fig. 6: Boehm's model

Software Maintenance

- Taute Maintenance Model

It is a typical maintenance model and has eight phases in cycle fashion. The phases are shown in Fig. 7

Fig. 7: Taute maintenance model

Software Maintenance

Phases :

1. Change request phase
2. Estimate phase
3. Schedule phase
4. Programming phase
5. Test phase
6. Documentation phase
7. Release phase
8. Operation phase

Estimation of maintenance costs

Phase	Ratio
Analysis	1
Design	10
Implementation	100

Table 3: Defect repair ratio

Software Maintenance

- Belady and Lehman Model

$$M = P + Ke^{(c-d)}$$

where

M : Total effort expended

P : Productive effort that involves analysis, design, coding, testing and evaluation.

K : An empirically determined constant.

c : Complexity measure due to lack of good design and documentation.

d : Degree to which maintenance team is familiar with the software.

Software Maintenance

Example – 9.1

The development effort for a software project is 500 person months. The empirically determined constant (K) is 0.3. The complexity of the code is quite high and is equal to 8. Calculate the total effort expended (M) if

(i) maintenance team has good level of understanding of the project (d=0.9)

(ii) maintenance team has poor understanding of the project (d=0.1)

Software Maintenance

Solution

Development effort (P) = 500 PM

$K = 0.3$

$C = 8$

(i) maintenance team has good level of understanding of the project (d=0.9)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.9)}$$

$$= 500 + 363.59 = 863.59 \text{ PM}$$

(ii) maintenance team has poor understanding of the project (d=0.1)

$$M = P + Ke^{(c-d)}$$

$$= 500 + 0.3e^{(8-0.1)}$$

$$= 500 + 809.18 = 1309.18 \text{ PM}$$

Software Maintenance

- Boehm Model

Boehm used a quantity called Annual Change Traffic (ACT).

"The fraction of a software product's source instructions which undergo change during a year either through addition, deletion or modification".

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

AME = ACT x SDE

Where, **SDE :** Software development effort in person months

ACT : Annual change Traffic

EAF : Effort Adjustment Factor

AME = ACT * SDE * EAF

Software Maintenance

Example – 9.2

Annual Change Traffic (ACT) for a software system is 15% per year. The development effort is 600 PMs. Compute estimate for Annual Maintenance Effort (AME). If life time of the project is 10 years, what is the total effort of the project ?

Software Maintenance

Solution

The development effort = 600 PM

Annual Change Traffic (ACT) = 15%

Total duration for which effort is to be calculated = 10 years

The maintenance effort is a fraction of development effort and is assumed to be constant.

AME = ACT x SDE

= 0.15 x 600 = 90 PM

Maintenance effort for 10 years = 10 x 90 = 90 PM

Total effort = 600 + 900 = 1500 PM

Software Maintenance

Example – 9.3

A software project has development effort of 500 PM. It is assumed that 10% code will be modified per year. Some of the cost multipliers are given as:

1. Required software Reliability (RELY) : high
2. Date base size (DATA) : high
3. Analyst capability (ACAP) : high
4. Application experience (AEXP) : Very high
5. Programming language experience (LEXP) : high

Other multipliers are nominal. Calculate the Annual Maintenance Effort (AME).

Solution

Annual change traffic (ACT) = 10%

Software development effort (SDE) = 500 Pm

Using Table 5 of COCOMO model, effort adjustment factor can be calculated given below :

RELY = 1.15

ACAP = 0.86

AEXP = 0.82

LEXP = 0.95

DATA = 1.08

Software Maintenance

Other values are nominal values. Hence,

EAF = 1.15 x 0.86 x 0.82 x 0.95 x 1.08 = 0.832

AME = ACT * SDE * EAF

= 0.1 * 500 * 0.832 = 41.6 PM

AME = 41.6 PM

Regression Testing

Regression testing is the process of retesting the modified parts of the software and ensuring that no new errors have been introduced into previously test code.

"Regression testing tests both the modified code and other parts of the program that may be affected by the program change. It serves many purposes :

- increase confidence in the correctness of the modified program
- locate errors in the modified program
- preserve the quality and reliability of software
- ensure the software's continued operation

Software Maintenance

- Development Testing Versus Regression Testing

Sr. No.	Development testing	Regression testing
1.	We create test suites and test plans	We can make use of existing test suite and test plans
2.	We test all software components	We retest affected components that have been modified by modifications.
3.	Budget gives time for testing	Budget often does not give time for regression testing.
4.	We perform testing just once on a software product	We perform regression testing many times over the life of the software product.
5.	Performed under the pressure of release date of the software	Performed in crisis situations, under greater time constraints.

- Regression Test Selection

Regression testing is very expensive activity and consumes significant amount of effort / cost. Many techniques are available to reduce this effort/ cost.

1. Reuse the whole test suite

2. Reuse the existing test suite, but to apply a regression test selection technique to select an appropriate subset of the test suite to be run.

Software Maintenance

Fragment A		Fragment B (modified form of A)	
S_1	y = (x - 1) * (x + 1)	S_1'	y = (x -1) * (x + 1)
S_2	if (y = 0)	S_2'	if (y = 0)
S_3	return (error)	S_3'	return (error)
S_4	else	S_4'	else
S_5	return $\left(\frac{1}{y}\right)$	S_5'	return $\left(\frac{1}{y-3}\right)$

Fig. 8: code fragment A and B

Test cases		
Test number	Input	Execution History
t_1	x = 1	S_1, S_2, S_3
t_2	x = -1	S_1, S_2, S_3
t_3	x = 2	S_1, S_2, S_5
t_4	x = 0	S_1, S_2, S_5

Fig. 9: Test cases for code fragment A of Fig. 8

Software Maintenance

If we execute all test cases, we will detect this divide by zero fault. But we have to minimize the test suite. From the fig. 9, it is clear that test cases t_3 and t_4 have the same execution history i.e. S_1, S_2, S_5. If few test cases have the same execution history; minimization methods select only one test case. Hence, either t_3 or t_4 will be selected. If we select t_4 then fine otherwise fault not found.

Minimization methods can omit some test cases that might expose fault in the modified software and so, they are not safe.

A safe regression test selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program.

- Selective Retest Techniques

Selective retest techniques may be more economical than the "retest-all" technique.

Selective retest techniques are broadly classified in three categories :

1. **Coverage techniques :** They are based on test coverage criteria. They locate coverable program components that have been modified, and select test cases that exercise these components.

2. **Minimization techniques:** They work like coverage techniques, except that they select minimal sets of test cases.

3. **Safe techniques:** They do not focus on coverage criteria; instead they select every test case that cause a modified program to produce different output than its original version.

Software Maintenance

Rothermal identified categories in which regression test selection techniques can be compared and evaluated. These categories are:

Inclusiveness measures the extent to which a technique chooses test cases that will cause the modified program to produce different output than the original program, and thereby expose faults caused by modifications.

Precision measures the ability of a technique to avoid choosing test cases that will not cause the modified program to produce different output than the original program.

Efficiency measures the computational cost, and thus, practically, of a technique.

Generality measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex modifications, and realistic testing applications.

Reverse Engineering

Reverse engineering is the process followed in order to find difficult, unknown and hidden information about a software system.

Software Maintenance

- Scope and Tasks

The areas there reverse engineering is applicable include (but not limited to):

1. Program comprehension
2. Redocumentation and/ or document generation
3. Recovery of design approach and design details at any level of abstraction
4. Identifying reusable components
5. Identifying components that need restructuring
6. Recovering business rules, and
7. Understanding high level system description

Software Maintenance

Reverse Engineering encompasses a wide array of tasks related to understanding and modifying software system. This array of tasks can be broken into a number of classes.

- Mapping between application and program domains

Fig. 10: Mapping between application and domains program

Software Maintenance

- Mapping between concrete and abstract levels
- Rediscovering high level structures
- Finding missing links between program syntax and semantics
- To extract reusable component

Software Maintenance

- Levels of Reverse Engineering

Reverse Engineers detect low level implementation constructs and replace them with their high level counterparts.

The process eventually results in an incremental formation of an overall architecture of the program.

Software Maintenance

Fig. 11: Levels of abstraction

Software Maintenance

Redocumentation

Redocumentation is the recreation of a semantically equivalent representation within the same relative abstraction level.

Design recovery

Design recovery entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code. This may be achieved from a combination of code, existing design documentation, personal experience, and knowledge of the problem and application domains.

Software RE-Engineering

Software re-engineering is concerned with taking existing legacy systems and re-implementing them to make them more maintainable.

The critical distinction between re-engineering and new software development is the starting point for the development as shown in Fig.12.

Software Maintenance

Fig. 12: Comparison of new software development with re-engineering

Software Maintenance

The following suggestions may be useful for the modification of the legacy code:

- ✓ Study code well before attempting changes
- ✓ Concentrate on overall control flow and not coding
- ✓ Heavily comment internal code
- ✓ Create Cross References
- ✓ Build Symbol tables
- ✓ Use own variables, constants and declarations to localize the effect
- ✓ Keep detailed maintenance document
- ✓ Use modern design techniques

Software Maintenance

- Source Code Translation

1. **Hardware platform update:** The organization may wish to change its standard hardware platform. Compilers for the original language may not be available on the new platform.

2. **Staff Skill Shortages:** There may be lack of trained maintenance staff for the original language. This is a particular problem where programs were written in some non standard language that has now gone out of general use.

3. **Organizational policy changes:** An organization may decide to standardize on a particular language to minimize its support software costs. Maintaining many versions of old compilers can be very expensive.

Software Maintenance

- Program Restructuring

1. **Control flow driven restructuring:** This involves the imposition of a clear control structure within the source code and can be either inter modular or intra modular in nature.

2. **Efficiency driven restructuring:** This involves restructuring a function or algorithm to make it more efficient. A simple example is the replacement of an IF-THEN-ELSE-IF-ELSE construct with a CASE construct.

Software Maintenance

```
IF Score > = 75 THEN Grade: = 'A'
ELSE IF Score > = 60 THEN Grade: = 'B'
ELSE IF Score > = 50 THEN Grade: = 'C'
ELSE IF Score > = 40 THEN Grade: = 'D'
ELSE IF Grade = 'F'
END
```

(a)

```
CASE Score of
75, 100: Grade: = 'A'
60, 74: Grade: = 'B';
50, 59: Grade: = 'C';
40, 49: Grade: = 'D';
ELSE Grade: = 'F'
END
```

(b)

Fig. 13: Restructuring a program

Software Maintenance

3. **Adaption driven restructuring:** This involves changing the coding style in order to adapt the program to a new programming language or new operating environment, for instance changing an imperative program in PASCAL into a functional program in LISP.

Configuration Management

The process of software development and maintenance is controlled is called configuration management. The configuration management is different in development and maintenance phases of life cycle due to different environments.

- Configuration Management Activities

The activities are divided into four broad categories.

1. The identification of the components and changes
2. The control of the way by which the changes are made
3. Auditing the changes
4. Status accounting recording and documenting all the activities that have take place

Software Maintenance

The following documents are required for these activities

- ✓ Project plan
- ✓ Software requirements specification document
- ✓ Software design description document
- ✓ Source code listing
- ✓ Test plans / procedures / test cases
- ✓ User manuals

Software Maintenance

- Software Versions

Two types of versions namely revisions (replace) and variations (variety).

Version Control :

A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This comprises the

- ✓ Name of each source code component, including the variations and revisions
- ✓ The versions of the various compilers and linkers used
- ✓ The name of the software staff who constructed the component
- ✓ The date and the time at which it was constructed

Software Maintenance

- Change Control Process

Change control process comes into effect when the software and associated documentation are delivered to configuration management change request form (as shown in fig. 14), which should record the recommendations regarding the change.

Software Maintenance

CHANGE REQUEST FORM

Project ID:

Change Requester with date:

Requested change with date:

Change analyzer:

Components affected:

Associated components:

Estimated change costs:

Change priority:

Change assessment:

Change implementation:

Date submitted to CCA:

Date of CCA decision:

CCA decision:

Change implementer:

Date submitted to QA:

Date of implementation:

Date submitted to CM:

QA decision:

Fig. 14: Change request form

Software Maintenance

Documentation

Software documentation is the written record of the facts about a software system recorded with the intent to convey purpose, content and clarity.

Software Maintenance

- User Documentation

S.No.	Document	Function
1.	System Overview	Provides general description of system's functions.
2.	Installation Guide	Describes how to set up the system, customize it to local hardware needs and configure it to particular hardware and other software systems.
3.	Beginner's Guide	Provides simple explanations of how to start using the system.
4.	Reference Guide	Provides in depth description of each system facility and how it can be used.
5.	Enhancement	Booklet Contains a summary of new features.
6.	Quick reference card	Serves as a factual lookup.
7.	System administration	Provides information on services such as net-working, security and upgrading.

Table 5: User Documentation

Software Maintenance

- System Documentation

It refers to those documentation containing all facets of system, including analysis, specification, design, implementation, testing, security, error diagnosis and recovery.

Software Maintenance

- System Documentation

S.No.	Document	Function
1.	System Rationale	Describes the objectives of the entire system.
2.	SRS	Provides information on exact requirements of system as agreed between user and developers.
3.	Specification/ Design	Provides description of: (i) How system requirements are implemented. (ii) How the system is decomposed into a set of interacting program units. (iii) The function of each program unit.
4.	Implementation	Provides description of: (i) How the detailed system design is expressed in some formal programming language. (ii) Program actions in the form of intra program comments.

Software Maintenance

S.No.	Document	Function
5.	System Test Plan	Provides description of how program units are tested individually and how the whole system is tested after integration.
6.	Acceptance Test Plan	Describes the tests that the system must pass before users accept it.
7.	Data Dictionaries	Contains description of all terms that relate to the software system in question.

Table 6: System Documentation

Multiple Choice Questions

Note: Choose most appropriate answer of the following questions:

9.1 Process of generating analysis and design documents is called

(a) Inverse Engineering (b) Software Engineering
(c) Reverse Engineering (d) Re-engineering

9.2 Regression testing is primarily related to

(a) Functional testing (b) Data flow testing
(c) Development testing (d) Maintenance testing

9.3 Which one is not a category of maintenance ?

(a) Corrective maintenance (b) Effective maintenance
(c) Adaptive maintenance (d) Perfective maintenance

9.4 The maintenance initiated by defects in the software is called

(a) Corrective maintenance (b) Adaptive maintenance
(c) Perfective maintenance (d) Preventive maintenance

9.5 Patch is known as

(a) Emergency fixes (b) Routine fixes
(c) Critical fixes (d) None of the above

Multiple Choice Questions

9.6 Adaptive maintenance is related to

(a) Modification in software due to failure

(b) Modification in software due to demand of new functionalities

(c) Modification in software due to increase in complexity

(d) Modification in software to match changes in the ever-changing environment.

9.7 Perfective maintenance refers to enhancements

(a) Making the product better

(b) Making the product faster and smaller

(c) Making the product with new functionalities

(d) All of the above

9.8 As per distribution of maintenance effort, which type of maintenance has consumed maximum share?

(a) Adaptive (b) Corrective

(c) Perfective (d) Preventive

9.9 As per distribution of maintenance effort, which type of maintenance has consumed minimum share?

(a) Adaptive (b) Corrective

(c) Perfective (d) Preventive

Multiple Choice Questions

9.10 Which one is not a maintenance model ?

(a) CMM
(b) Iterative Enhancement model
(c) Quick-fix model
(d) Reuse-Oriented model

9.11 In which model, fixes are done without detailed analysis of the long-term effects?

(a) Reuse oriented model
(b) Quick-fix model
(c) Taute maintenance model
(d) None of the above

9.12 Iterative enhancement model is a

(a) three stage model
(b) two stage model
(c) four stage model
(d) seven stage model

9.13 Taute maintenance model has

(a) Two phases
(b) six phases
(c) eight phases
(d) ten phases

9.14 In Boehm model, ACT stands for

(a) Actual change time
(b) Actual change traffic
(c) Annual change traffic
(d) Annual change time

Multiple Choice Questions

9.15 Regression testing is known as

(a) the process of retesting the modified parts of the software
(b) the process of testing the design documents
(c) the process of reviewing the SRS
(d) None of the above

9.16 The purpose of regression testing is to

(a) increase confidence in the correctness of the modified program
(b) locate errors in the modified program
(c) preserve the quantity and reliability of software
(d) All of the above

9.17 Regression testing is related to

(a) maintenance of software
(b) development of software
(c) both (a) and (b)
(d) none of the above.

9.18 Which one is not a selective retest technique

(a) coverage technique
(b) minimization technique
(c) safe technique
(d) maximization technique

Multiple Choice Questions

9.19 Purpose of reverse engineering is to

(a) recover information from the existing code or any other intermediate document

(b) redocumentation and/or document generation

(c) understand the source code and associated documents

(d) All of the above

9.20 Legacy systems are

(a) old systems (b) new systems

(c) undeveloped systems (d) None of the above

9.21 User documentation consists of

(a) System overview (b) Installation guide

(c) Reference guide (d) All of the above

9.22 Which one is not a user documentations ?

(a) Beginner's Guide (b) Installation guide

(c) SRS (d) System administration

Multiple Choice Questions

9.23 System documentation may not have

(a) SRS (b) Design document

(c) Acceptance Test Plan (d) System administration

9.24 The process by which existing processes and methods are replaced by new techniques is:

(a) Reverse engineering (b) Business process re-engineering

(c) Software configuration management (d) Technical feasibility

9.25 The process of transforming a model into source code is

(a) Reverse Engineering (b) Forward engineering

(c) Re-engineering (d) Restructuring

Exercises

9.1 What is software maintenance? Describe various categories of maintenance. Which category consumes maximum effort and why?

9.2 What are the implication of maintenance for a one person software production organisation?

9.3 Some people feel that "maintenance is manageable". What is your opinion about this issue?

9.4 Discuss various problems during maintenance. Describe some solutions to these problems.

9.5 Why do you think that the mistake is frequently made of considering software maintenance inferior to software development?

9.6 Explain the importance of maintenance. Which category consumes maximum effort and why?

9.7 Explain the steps of software maintenance with help of a diagram.

9.8 What is self descriptiveness of a program? Explain the effect of this parameter on maintenance activities.

Exercises

9.9 What is ripple effect? Discuss the various aspects of ripple effect and how does it affect the stability of a program?

9.10 What is maintainability? What is its role during maintenance?

9.11 Describe Quick-fix model. What are the advantage and disadvantage of this model?

9.12 How iterative enhancement model is helpful during maintenance? Explain the various stage cycles of this model.

9.13 Explain the Boehm's maintenance model with the help of a diagram.

9.14 State the various steps of reuse oriented model. Is it a recommended model in object oriented design?

9.15 Describe the Taute maintenance model. What are various phases of this model?

9.16 Write a short note on Boledy and Lehman model for the calculation of maintenance effort.

Exercises

9.17 Describe various maintenance cost estimation model.s

9.18 The development effort for a project is 600 PMs. The empirically determined constant (K) of Belady and Lehman model is 0.5. The complexity of code is quite high and is equal to 7. Calculate the total effort expended (M) if maintenance team has reasonable level of understanding of the project (d=0.7).

9.19 Annual change traffic (ACT) in a software system is 25% per year. The initial development cost was Rs. 20 lacs. Total life time for software is 10 years. What is the total cost of the software system?

9.20 What is regression testing? Differentiate between regression and development testing?

9.21 What is the importance of regression test selection? Discuss with help of examples.

9.22 What are selective retest techniques? How are they different from "retest-all" techniques?

Exercises

9.23 Explain the various categories of retest techniques. Which one is not useful and why?

9.24 What are the categories to evaluate regression test selection techniques? Why do we use such categorisation?

9.25 What is reverse engineering? Discuss levels of reverse engineering.

9.26 What are the appropriate reverse engineering tools? Discuss any two tools in detail.

9.27 Discuss reverse engineering and re-engineering.

9.28 What is re-engineering? Differentiate between re-engineering and new development.

9.29 Discuss the suggestions that may be useful for the modification of the legacy code.

9.30 Explain various types of restructuring techniques. How does restructuring help in maintaining a program?

Exercises

9.31 Explain why single entry, single exit modules make testing easier during maintenance.

9.32 What are configuration management activities? Draw the performa of change request form.

9.33 Explain why the success of a system depends heavily on the quantity of the documentation generated during system development.

9.34 What is an appropriate set of tools and documents required to maintain large software product/

9.35 Explain why a high degree of coupling among modules can make maintenance very difficult.

9.36 Is it feasible to specify maintainability in the SRS? If yes, how would we specify it?

9.37 What tools and techniques are available for software maintenance? Discuss any two of them.

Exercises

9.38 Why is maintenance programming becoming more challenging than new development? What are desirable characteristics of a maintenance programmer?

9.39 Why little attention is paid to maintainability during design phase?

9.40 List out system documentation and also explain their purpose.