SOFTWARE ENGINEERING

SYLLABUS

Objective:

This course introduces the concepts and methods required for the construction of large software intensive systems.

UNIT-I:

Introduction - Evolving Role of Software - Changing Nature of Software – Software Myths; A Generic View of Process: Layered Technology - Process Models: Waterfall Model - Evolutionary Process Models.

UNIT-II:

Requirements Engineering: Tasks - Initiating the Requirements Engineering Process - Eliciting Requirements - Building the Analysis Model - Requirements Analysis - Data Modelling Concepts.

UNIT-III:

Data Engineering: Design Process and Design Quality - Design Concepts - The Design Model Creating an Architectural Design: Software Architecture - Data Design -Architectural Design - Mapping Data Flow into Software Architecture; Performing User Interface Design: Golden Rules.

UNIT-IV:

Testing Strategies: Strategic Approach to Software Testing- Test Strategies for Conventional and Object Oriented Software - Validation Testing - System Testing -Art of Debugging. Testing Tactics: Fundamentals - White Box- Basis Path - Control Structure - Black Box Testing Methods

UNIT-V:

Project Management: Management Spectrum - People - Product - Process - Project. Estimation: Project Planning Process - Resources - Software Project Estimation - Project Scheduling - Quality Concepts - Software Quality Assurance - Formal Technical Reviews.

TEXT BOOK:

Roger S Pressman, "Software Engineering - A Practitioner's Approach", Sixth Edition, McGraw Hill International Edition and New York: 2005.

REFERENCES:

1. Ian Somerville, "Software Engineering", 7th Edition, Pearson Education, 2006.

2. Mall Rajib," Software Engineering", 2/E, PHI, 2006.

SOFTWARE ENGINEERING

Unit -1:

Introduction - The Evolving Role of Software - Changing Nature of Software - Software Myths; A Generic View of Process: Layered Technology - Process models: Waterfall model-Evolutionary Process Models.

1.1 Introduction to Software Engineering

Software

Software is a set of instructions or programs instructing a computer to do specific task. Software is considered to be a collection of executable programming code, associated libraries and documentations.

Engineering

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software Engineering

Software Engineering is about teams and it is about quality. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software Engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget.

Software engineering is also about communication on a team and with internal and external stakeholders. Teams do not consist only of developers, but also of quality assurance testers, systems architects, system/platform engineers, customers, project managers and other stakeholders. The Primary goals of software engineering are to improve the quality of software products and to increase the productivity and job satisfaction of software engineers.

Definition

"Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.**"**

"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".

Program versus Software

Software is more than programs. It consists of programs; documentation of any facet of the program and the procedure used to setup and operate the software system.

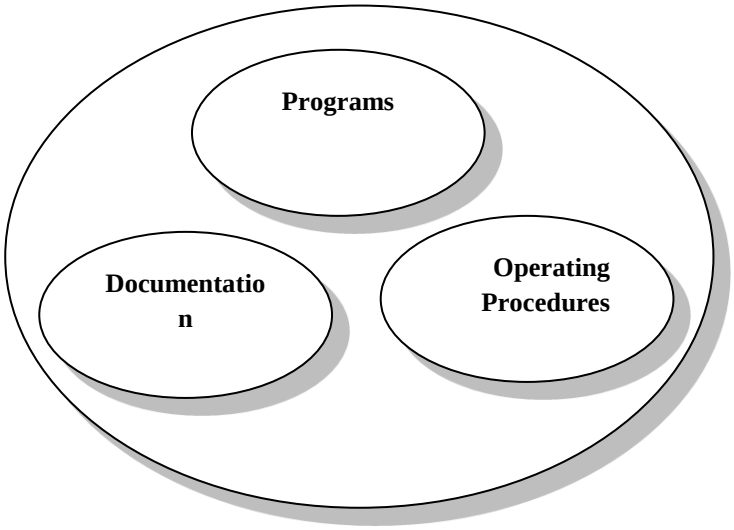


Fig. 1.1 Components of Software

Software = Programs + Documentation + Operating Procedures

1.2 Evolving Role of Software

In the late 1960s, it was clear that software development was unlike the construction of physical structures. This was because in software development, more programmers could not be added simply to speed up a lagging development project.

Software had become a critical component of many systems, yet it was too complex to develop with any certainty of schedule or quality. This problem imposed financial and public safety concerns. Software errors have caused large-scale financial losses as well as inconvenience to many. Disasters, such as Y2K problem have affected economic, political, and administrative system of various countries around the world. This situation where catastrophic failures have occurred is known as **Software crisis**.

Software crisis is a term that has been used since the early days of software engineering to describe the impact of the rapid increases in computer power and its complexity. Software crisis occurs due to problems associated with poor quality software. This includes problems arising from malfunctioning of software systems, inefficient development of software, and most importantly, dissatisfaction among users of the software.

Role of software

The role of software has undergone drastic change in the last few decades. These improvements range through hardware, computing architecture, memory, storage capacity and a wide range of unusual input and output conditions.

All these significant improvements have lead to the development of more complex and sophisticated computer-based systems. Sophistication leads to better results but can cause problems for those who build these systems

Lone programmer has been replaced by a team of software experts. These experts focus on individual parts of technology in order to deliver a complex application. However, the experts still face the same questions as that by a lone programmer:

- Why does it take long to finish software?
- Why are the costs of development so high?
- Why aren't all the errors discovered before delivering the software to customers?
- Why is it difficult to measure the progress of developing software?

All these questions and many more have lead to the manifestation of the concern about software and the manner in which it is developed – a concern which lead to the evolution of the software engineering practices.

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware.

Whether it resides within a cellular phone or operates inside a mainframe computer, software is information transformer — producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments). Software delivers the most important product of our time — information.

Software transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context; it manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

1.3 The Changing Nature of Software

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

- System Software
- Real Time Software
- Embedded Software
- Business Software
- Personal Computer Software
- Artificial Intelligence Software
- Web Based Software
- Engineering And Scientific Software

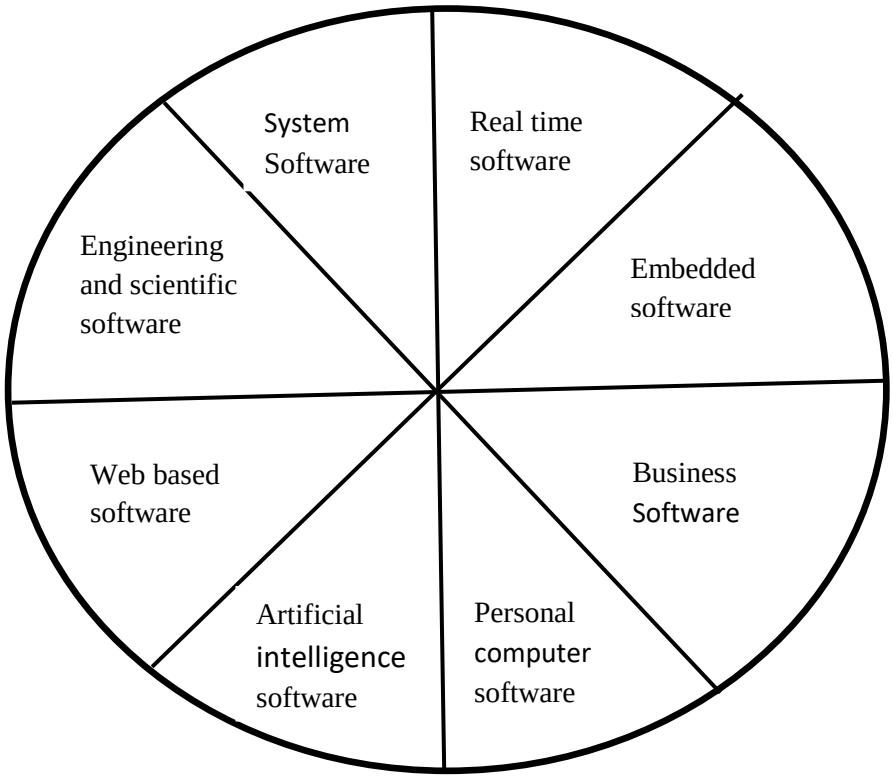


Fig. 1.2 Software Applications

1. System Software

System Software is a collection of programs written to service other programs. Infrastructure software comes under this category like compilers, operating systems, editors, drivers, etc.

System software is software that provides platform to other software's. Some examples can be operating systems, antivirus software, disk formatting software as, computer language translator etc. These are commonly prepared by the computer manufactures.

2. Real Time Software

This Software is used to monitor, control and analyze real world events as they occur. Real time software deals with changing environment.

An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather.

3. Embedded Software

Embedded Software resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. This type of software is placed in **"Read Only Memory (ROM)"** of the product and controls the various functions of the product.

The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software.

4. Business Software

This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system employee management, and account management. It may also be a data warehousing tool which helps us to take decisions based on available data.

Management information system, Enterprise Resource Planning (ERP) and such other software are popular example of business software.

5. Personal Computer Software

The Software used in personal computers is covered in this category. Examples are word processors, computer graphics, and multimedia and animating tools, database management, personal and financial applications, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

6. Artificial Intelligence Software

AI Software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert system, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

7. Web Based Software

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

8. Engineering and Scientific Software

Formerly characterized by "number crunching" algorithms, engineering and scientific software applications range from astronomy to volcano logy, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

Scientific and engineering application software is grouped in this category. Huge computing is normally required to process data. Examples are CAD/CAM packages, SPSS, MATLAB, Engineering pro, Circuit analyzers etc.

1.4 Software Myths

Myth is defined as "widely held but false notation" by the oxford dictionary, so as in other fields software arena also has some myths to demystify. Pressman insists "Software myths- beliefs about software and the process used to build it- can be traced to earliest days of computing.

Myths have a number of attributes that have made them insidious." So software myths prevail but though they do are not clearly visible they have the potential to harm all the parties involved in the software development process mainly the developer team.

Here are number of myths associated with software development community. Software myths-beliefs about software and process used to build it –can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious.

In developing software, the developers put their extreme dedication and hard work. A great majority of software related problems arises because of the myths that formed during the software developments initial stages.

- Software is easy to change
- Computers provide greater reliability than the devices they replace.

- Testing software or 'proving' software correct can remove all the errors.
- Reusing software increases safety.
- Software can work right the first time.
- Software can be designed thoroughly enough to avoid most integrations problems.
- Software with more features is better software.
- Addition of more software engineers will make up the delay.
- Aim is to develop working programs.

1. Software is easy to change

It is true that source code files area easy to edit, but that is quite different than saying that ***software is easy to change.*** This is deceptive precisely because source code is so easy to alter. But making changes without introducing errors is extremely difficult, particularly in organization with poor process maturity.

Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.

2. Computers provide greater reliability than the devices they replace.

It is true that software does not fail in the traditional sense. There are no limits to how many times a given piece of code can be exhausted before it "wears out". In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have been computerized.

Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.

3. Testing software or 'proving' software correct can remove all the errors

Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.

4. Reusing software increases safety

This myth is particularly troubling because of the false sense of security that code re-use can create. Code reuse is a very powerful tool that can yield dramatic improvement in development efficiency, but it's still requires analyses to determine its suitability and testing tool determine if it works.

5. Software can work right the first time

If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote as a price.

If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineer's area often asked to do precisely this sort of work, and they often accept the job.

6. Software can be designed thoroughly enough to avoid most integrations problems

There is an old saying among software designers: "Too bad, there is no compiler for specifications": These points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these.

Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.

7. Software with more features is better software

This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.

8. Addition of more software engineers will make up the delay

This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

9. Aim is to develop working programs

The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community.

These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. Primarily, there are three types of software myths, all the three are stated below:

- Management Myth
- Customer Myth
- Practitioner/Developer Myth

Management Myth

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if those beliefs will lessen the pressure (even temporarily). Some common managerial myths stated by Roger Pressman include:

- We have standards and procedures for building software, so developers have everything they need to know.
- We have state-of-the-art software development tools; after all, we buy the latest computers.
- If we're behind schedule, we can add more programmers to catch up.
- A good manger can manage any project.

The managers completely ignore that fact that they are working on something intangible but very important to the clients which invites more trouble than solution.

So a software project manger must have worked well with the software development process analyzing the minute deals associated with the field learning the nitty-gritty and the tips and trick of the trade.

Customer Myths

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract.

In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer. Commonly held myths by the clients are:

- A general statement of objectives is sufficient to begin writing programs - we can fill in the details later.
- Requirement changes are easy to accommodate because software is flexible.
- I know what my problem is; therefore I know how to solve it.

This primarily is seen evidently because the clients do not have a firsthand experience in software development and they think that it's an easy process.

Practitioner/ Developer Myths

Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

A malpractice seen is developers are that they think they know everything and neglect the peculiarity of each problem.

- If I miss something now, I can fix it later.
- Once the program is written and running, my job is done.
- Until a program is running, there's no way of assessing its quality.
- The only deliverable for a software project is a working program.

- Every developer should try to get all requirement is relevant detail to effectively design and code the system.

1.5 A Generic View of Process

1.5.1 Introduction

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

A software process as a framework for the tasks that are required to build high-quality software. The work associated with software engineering can be categorized into three generic phases, regardless of application area, project size, or complexity.

A number of process models have been proposed for developing software. In fact, each organization that follows a process has its own customized version.

Different processes can also be divided into different activities and activities into sub-activities. However, in general, any problem solving in software must consist of such activities as

- Requirement specification for understanding and clearly stating the problem
- Deciding a plan for a solution
- Coding for implementing the planned solution, and
- Testing for verifying the programs.

These activities may not be done explicitly in small size projects.

1.5.2 Software Engineering – A Layered Technology

Software Engineering can be viewed as a layered technology. To develop software, we need to go from one layer to another. Software development is totally a layered technology. All this layers are related to each other and each layer demands the fulfilment of the previous layer. The layers technology consists of

- Quality Focus
- Process
- Method
- Tools

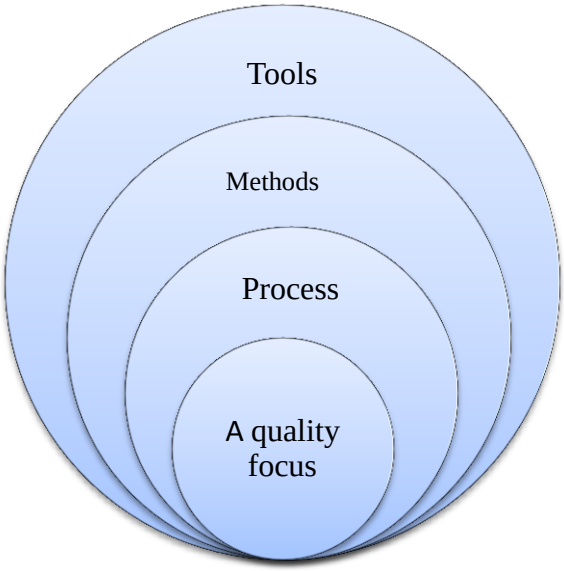


Fig. 1.3 Layered Technology

1. Quality Focus

Any engineering approach (including software engineering) must rest on an organizational commitment to quality.

Total quality management, six sigma, and similar philosophies foster a continues process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering.

2. Process layer

The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.

Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed.

It is an adhesive that enables rational and timely development of computer software. It defines an outline for a set of key process areas that must be acclaimed for effective delivery of software engineering technology.

3. Method layer

Software engineering methods provide the technical "how to's" for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modelling, program construction, testing, and support.

It provides technical knowledge for developing software. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modelling activities and other descriptive techniques.

4. Tools layer

Software engineering tools provide automated or semi automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering or CASE is established.

CASE combines software, hardware and software engineering database to create software engineering analogous to computer-aided design or CAD hardware. CASE helps in application development including analysis, design, code generation, and debugging and testing etc.

1.5.3 A Process Framework

A process framework establishes the foundation for a complete software process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

In additional, the process framework encompasses a set of umbrella activities that are applicable across the entire software process. Software engineering approach pivots around the concept of process.

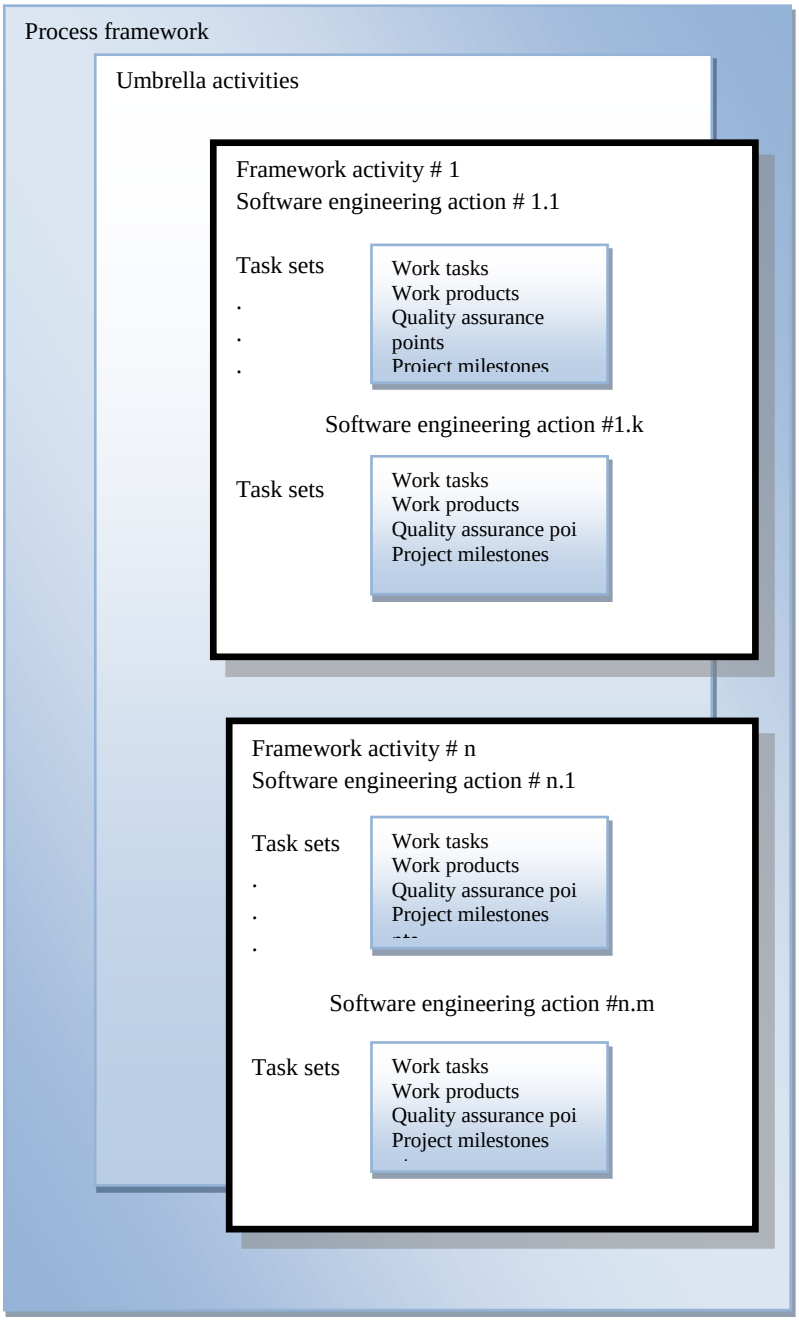


Fig. 1.4 Software Process Framework

A process means "a particular method of doing something, generally involving a number of steps or operations"

.In software engineering; the phase software process refers to the method of developing software. Software process specifies how one can manage and plan a software development project taking constraints and boundaries into consideration.

A software process is a set of activities, together with ordering constraints among them, such that if the activities are performed properly and in accordance with the ordering constraints, the desired result is produced.

The desired result is high-quality software at low cost. The process that deals with the technical and management issues of software development is called a software process.

As different type of activities are being performed, which are frequently done by different people, it is better to view the software process as consisting of many in component processes, each consisting of a certain type of active

. The following generic process framework is applicable to the vast majority of software project.

Communication This framework activity involves heavy communication and coloration with the customer and encompasses requirements gathering and other related activities.

Planning This activity establishes a plan for software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modelling The activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

Construction This activity combines four generation and the testing that is required uncovering errors in the code.

Deployment The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small programs, the creation of large web application, and for the engineering of large, complex computer based systems. This implies that a software engineering action can be adapted to the specific needs of the software projects and the characteristics of the project team.

The framework describe in the generic view of software engineering is complemented by a number of umbrella activities. Typical activities in this category include:

Software project tracking and control allows the software team to assess progress against the project plan and take necessary action to maintain schedule.

Risk management assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance defines and conducts the activities required to ensure software quality.

Formal technical reviews assess software engineering work products in an effort to uncover and remove errors before them or propagated to the next action or activity.

Measurement defines and collects process, projects, and product measures that assist the team in delivering software that needs customers needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management manage the effects of change throughout the software process.

Reusability management defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

Work product preparation and production encompasses the activity required to create work product such as models, documents, logs, forms, and lists. Intelligent applications of any software process model must recognize that adaptation is essential for success. But process models do differ fundamentally in

- The overall flow of activities and tasks and the interdependencies among activities and tasks.

- The degree to which work tasks are defined within each framework activity.
- The degree which works products are identified and required.
- The manner which quality assurance activities are applied.

1.5.4 The capability maturity model integration (CMMI)

The Software engineering institute has developed a comprehensive process meta-model that is predicated on a set of systems and software engineering capabilities that should be a present as organisations reach different levels of process capability and maturity.

The CMMI represents a process meta model in two different ways.

- As a continuous model and
- As a staged model

Each process area is formally assessed against specific goal and practices and is rated according to the following capability levels:

Level 0: incomplete -- To process area (requirements management) is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability

Level 1: performed -- All of the specific goals of the process area have been satisfied. Work tasks required to produce define work products being conducted.

Level 2: managed -- All level1 criteria have been satisfied. In addition, all work associated with the process area conforms to an organizationally defined policy.

Level 3: defined -- All level2 criteria have been achieved. In addition, the process is "tailored from the organizations set of standard processes according to the organizations tailoring guidelines, and contributed work products, measures and other process – improvement information to the organizational process assets".

Level 4: quantitatively managed – All level3 criteria have been achieved. In addition, the process area is and improved using measurement and quantitative assessment. "quantitative objectivise for quality and process performance are established and used as criteria in managing the process.

Level 5: optimized – All capability level4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area and consideration".

1.6 Process Model

1.6.1 The Waterfall Model (*The Linear Sequential Model*)

The Waterfall Model, sometimes called as *the Classic Life Cycle*, suggest a systematic, sequential approach to software development that begins at the system level and progresses through analysis, planning, modelling, construction and deployment, culminating in on-going support of the completed software.

Communication The software development starts with the communication between customer and developer.

Planning It consist of complete estimation, scheduling for project development..

Modelling Software design focuses on four distinct attributes of a program: data structure, software architecture, interface representations and procedural details. The design process translates the requirements into a representation of the software.

Construction (*Coding and Testing*) The design must be translated into machine-readable form programming language.

The Coding is performed based on the Coding Guidelines and Standards. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, test errors and ensures that defined input produces actual result.

Deployment Software will be undergoing many changes after it is delivered to the customer. Software maintenance reapplies each of the preceding life-cycle steps to an existing program

Problems

- The classic life cycle model is the oldest paradigm for software engineering.
- Real project rarely follow sequential flow.
- It is often difficult for the customer to state all requirements explicitly.

- The customer must have patience to get the working version of the programs, which is available only late.

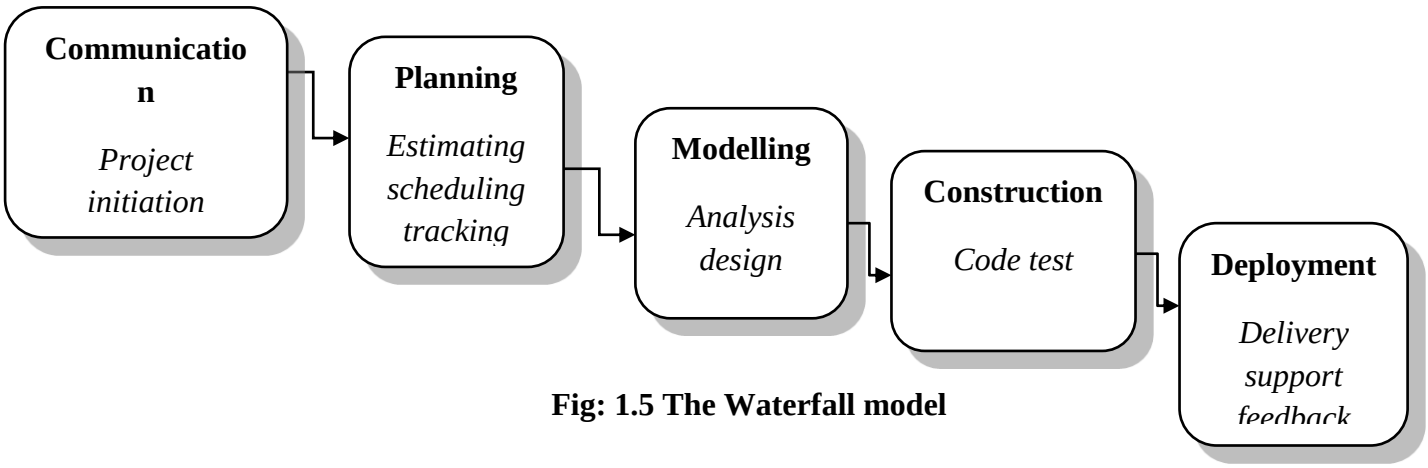


Fig: 1.5 The Waterfall model

1.6.2 The V Model

The V model is known as ***Verification*** and ***Validation*** mode. This model is an extension of the waterfall model. In the life cycle of V- model, processes are executed sequentially. Every phase completes its execution before the execution of next phase begins.

Requirements The requirements of product are understood from the customer's point of view to know their exact requirement and expectation. The requirements are documented and reviewed with the customer.

The acceptance test design planning is completed at requirement stage because business requirements are used as an input for acceptance testing.

System Design In system design high level design of the software is constructed. In the software engineer the phase of system design, we study how the requirements are implemented their technical use.

Architecture Design Here software architecture is created on the basis of high level design. The module relationship and dependencies of module, architectural diagrams, database tables, and technology details are competed.

Module Design In this phase, we separately design every module or the software components. Finalize all the methods, classes, interface, etc. Unit tests are designed in module design based on the internal module designs.

Unit test are the essential part of any development process. They help to separate an extreme faults and errors at the prior stage.

Coding The coding of the system modules designed in the design phase is grabbed in the coding phase. The coding is performed based on the coding guidelines and standards.

Merits

- V-model is easy and simple to use and understood.
- Works well for small projects where the requirements are very well and easy to understand.
- This is highly-disciplined model and it saves more time.

Demerits

- V-model is not suitable for substantial and complex project
- If the requirements are varying then this model is not acceptable

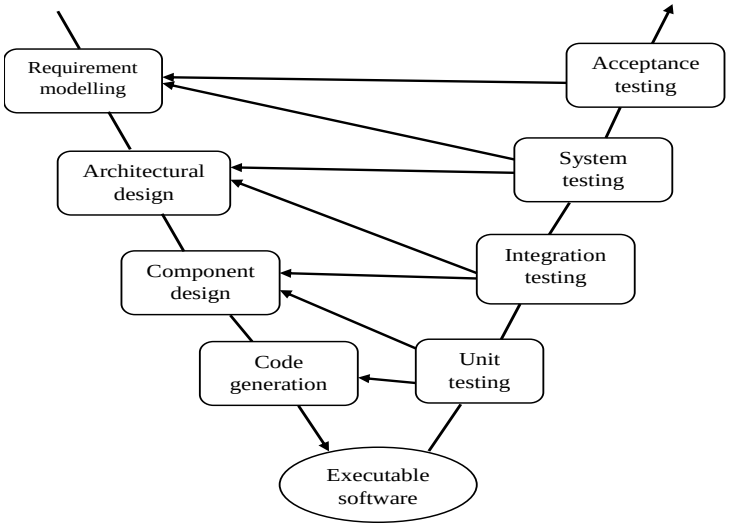


Fig: 1.6 The V-Model

1.6.3 Incremental Model

There are many situations in which initial software requirements are reasonably well- defined but the overall scope of the development effort precludes a purely linear process. The incremental model combines elements of the waterfall model applied in an iterative fashion.

The first incremental model is generally a core product. Each increment builds the product and submits it to the customer for suggesting any modification. Unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

Merits

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug in the smaller iteration.
- The customer can respond to its functionalities after every increment.

Demerits

- The cost of the product may cross the initially estimated.
- This model requires a very clear and complete planning.
- The demands of customer for the additional functionalities after every increment cause problem in the system architecture.

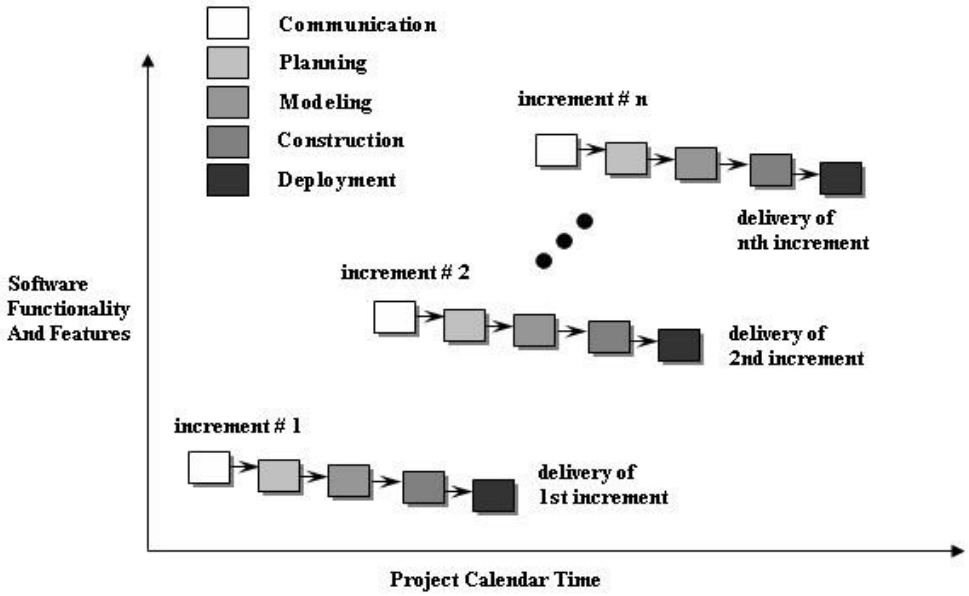


Fig. 1.7 Incremental Models

1.7 Evolutionary Process model

1.7.1 Prototyping

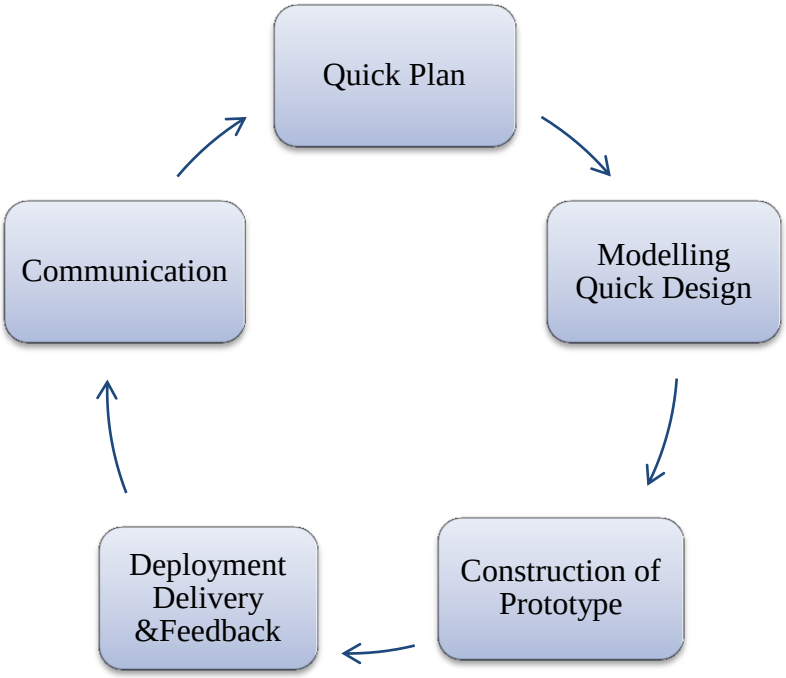
Suppose **customer** defines a set of general objectives for software but does not identify detailed requirements for functions and features. Prototyping can be used as a stand-alone process model.

In other case, the **developer** may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human- machine interaction should take. In this situation a prototype paradigm may offer the best approach.

The prototyping paradigm as shown in the following figure begins with **communication**. Developer and customer meet and define the overall objectives of the software, identity the requirements and outline the system.

Prototyping iteration is planned quickly and quick design occurs. A quick design focuses on a representation of those aspects of the software that will be visible to the customer/user. The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by the customer/user and used to refine requirements for the software to be developed. A process of iteration occurs unit prototype is turned to satisfy the needs of the customer.



.

Fig. 1.8 Incremental Models

Merits

- Customer can see steady progress.
- This is useful when requirements are changing rapidly**.**

Demerits

- It is impossible to know how long it will take.

There is no way to know the no of iteration will be required

1.7.2 The Spiral Model

The spiral model is an evolutionary software process model that combines the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It is risk-driven process model. Using this model, software is developed in a series of evolutionary releases.

During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called ***task regions***. Each of the framework activities represents one segments of the spiral path illustrated in figure.

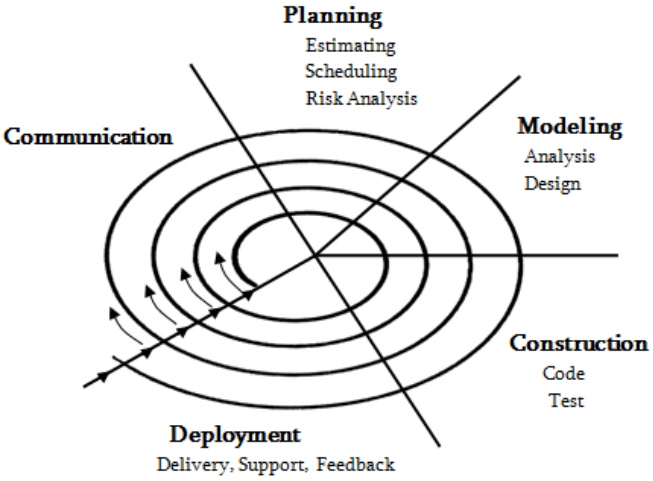


Fig. 1.9 The Spiral Model

The first circuit around the spiral might result in the development of a product specification; subsequently passes around the spiral might be used to develop a prototyping and the progressively more sophisticated version of the software.

Unlike other process models that end when software is delivered, the spiral model can be adapted to apply throughout the life of the computer software.

- Concept development project.
- New product development project.
- Product enhancement project.

Merits

- Cost estimation becomes easy.
- Additional functionally of changes can be done at later stage.
- Development is fast and features are added in a systematic way
- There is always space for customer feedback.

Demerits

- Risk is not meeting the schedule of budget.
- Documentation is more as it has intermediate phases.
- It is not advisable for smaller projects; it might cost them a lot.

1.7.3 Concurrent Models

The concurrent development models also called as concurrent engineering. The following figure provides a schematic representation of one software engineering task within the modelling activity for the concurrent process model. All software engineering activities exit concurrently but reside in different states.

For example, early in a project the communication activity has completed its first iteration and exits in the **awaiting changes** state. The modelling activity which existed in the none state while initial communication was completed now makes a transition into the under development state. The concurrent modelling defines a series of events that will trigger transitions from state to state for each of the software engineering activities.

For example, the event analysis model correction which will trigger the analysis activity from the done state into the awaiting changes state. The concurrent modelling is applicable to all types of software development and provides an accurate picture of the current state of a project. Event generated at one point in the process network trigger transitions among the states associated with each activity.

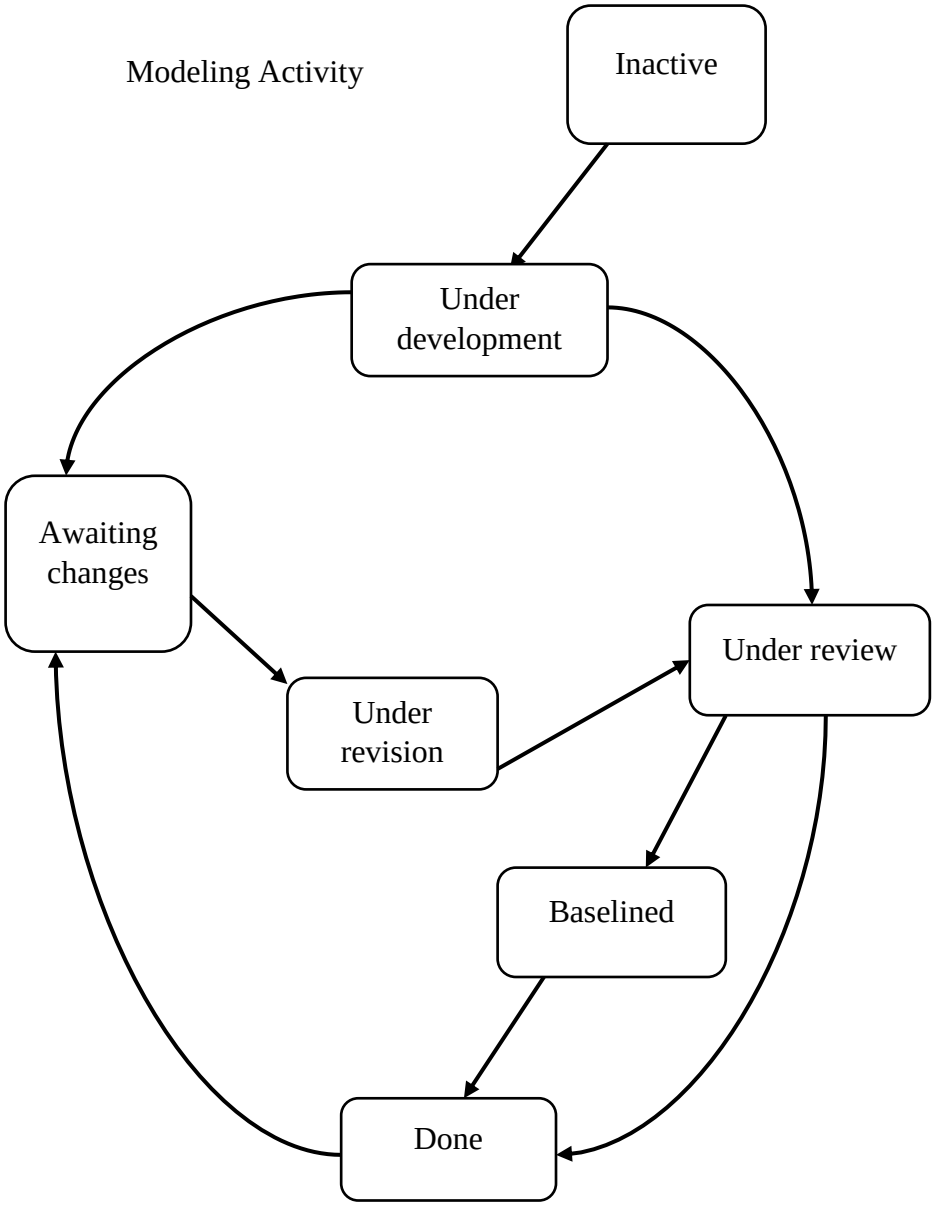


Fig. 1.10 The Concurrent Model

2 Marks

1. Define Software Engineering.

Software Engineering is about teams and it is about quality. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software Engineering has the objective of solving these problems by producing good quality, maintainable software, on time, within budget.

2. Define System Software.

System Software is a collection of programs written to service other programs. Infrastructure software comes under this category like compilers, operating systems, editors, drivers, etc.

System software is software that provides platform to other software's. Some examples can be operating systems, antivirus software, disk formatting software as, computer language translator etc. These are commonly prepared by the computer manufactures.

3. What is Software Myths

Myth is defined as "widely held but false notation" by the oxford dictionary, so as in other fields software arena also has some myths to demystify. Pressman insists "Software myths- beliefs about software and the process used to build it- can be traced to earliest days of computing.

4. What are the different stages to develop Software?

- Software is easy to change
- Computers provide greater reliability than the devices they replace.
- Testing software or 'proving' software correct can remove all the errors.
- Reusing software increases safety.
- Software can work right the first time.
- Software can be designed thoroughly enough to avoid most integrations problems.

5. Define layered technology and its types.

Software Engineering can be viewed as a layered technology. To develop software, we need to go from one layer to another. Software development is totally a layered technology. All this layers are related to each other and each layer demands the fulfilment of the previous layer**.** The layers technology consists of

- Quality Focus
- Process
- Method
- Tools

6. Define Waterfall Model.

The Waterfall Model, sometimes called as *the Classic Life Cycle*, suggest a systematic, sequential approach to software development that begins at the system level and progresses through analysis, planning, modelling, construction and deployment, culminating in on-going support of the completed software.

7. Define V Model.

The V model is known as ***Verification*** and ***Validation*** mode. This model is an extension of the waterfall model. In the life cycle of V- model, processes are executed sequentially. Every phase completes its execution before the execution of next phase begins.

8. Define Incremental Model.

There are many situations in which initial software requirements are reasonably well- defined but the overall scope of the development effort precludes a purely linear process. The incremental model combines elements of the waterfall model applied in an iterative fashion.

The first incremental model is generally a core product. Each increment builds the product and submits it to the customer for suggesting any modification. Unlike prototyping, the incremental model focuses on the delivery of an operational product with each increment.

9. What is Prototype?

Prototyping iteration is planned quickly and quick design occurs. A quick design focuses on a representation of those aspects of the software that will be visible to the customer/user. The quick design leads to the construction of a prototype.

The prototype is deployed and evaluated by the customer/user and used to refine requirements for the software to be developed. A process of iteration occurs unit prototype is turned to satisfy the needs of the customer.

10. What are the different phases of prototyping model?

The different phases of prototyping model are:

1) Communication
2) Quick design
3) Modelling and quick design
4) Construction of prototype
5) Deployment, delivery, feedback

11. Define RAD.

RAD is a Rapid Application Development model. Using the RAD model, Software product is developed in a short period of time. The initial activity starts with the communication between the customer and developer.

12. Define Spiral Model.

The spiral model is an evolutionary software process model that combines the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model. It is risk-driven process model. Using this model, software is developed in a series of evolutionary releases.

During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called ***task regions***.

13. What is Concurrent Model.

The concurrent development models also called as concurrent engineering. The following figure provides a schematic representation of one software engineering task within the modelling activity for the concurrent process model. All software engineering activities exit concurrently but reside in different states.

10 Marks

1. Discuss about changing nature of software.

2. Explain About generic view of software engineering.

3. Explain about software myths in software engineering.

4. Discuss about waterfall models and their activities in each phase.

5. Discuss about evolutionary process model.

SOFTWARE ENGINEERING

Unit -II:

Requirements Engineering: Tasks - Initiating the Requirements Engineering Process - Eliciting Requirements - Building the Analysis Model - Requirements Analysis - Data Modeling Concepts.

Objectives

The objective of this chapter is to introduce software requirements and to discuss the processes involved in discovering and documenting these requirements.

When you have read the chapter you will:

- Understand the concepts of user and system requirements and why these requirements should be written in different ways;
- Understand the differences between functional and non-functional software requirements;
- Understand how requirements may be organized in a software requirements document;

Contents

Requirements Defined

Requirements are the features that your application must provide. At the beginning of the project, you gather requirements from the customers to figure out what you need to build. Throughout development, you use the requirements to guide development and ensure that you're heading in the right direction. At the end of the project, you use the requirements to verify that the finished application actually does what it's supposed to do.

2.1 Requirements Engineering

Requirements engineering helps software engineers understand the problem they are to solve. It involves activities that lead to understanding the business context, what the customer wants, how end-users will interact with the software, and what the business impact will be. Requirements engineering starts with the problem **definition: A software requirement is a description of a software system to be developed. It lays out functional and non-functional requirements, and may include a set of use cases that describe user interactions that the software must provide.**

This is an informal description of what the customers think they need from a software system to do for them. The problem could be identified by management personnel, through market research, by ingenious observation, or some other means. The statement of work captures the perceived needs and, because it is opinion-based, it usually evolves over time, with changing market conditions or better understanding of the problem. Defining the requirements for the system-to-be includes both fact-finding about how the problem is solved in the current practice as well as envisioning how the planned system might work. The final outcome of requirements engineering is a requirements specification document. The key task of requirements engineering is formulating a well-defined problem to solve. A well-defined problem includes

A set of criteria ("requirements") according to which proposed solutions either definitely solve the problem or fail to solve it

The description of the resources and components at disposal to solve the problem. Requirements engineering involves different stakeholders in defining the problem and specifying the solution. A stakeholder is an individual, team, or organization with interests in, or concerns related

to, the system-to-be. Generally, the system-to-be has several types of stakeholders: customers, end users, business analysts, systems architects and developers, testing and quality assurance engineers, project managers, the future maintenance organization, owners of other systems that will interact with the system-to-be, etc. The stakeholders all have a stake, but the stakes may differ. End users will be interested in the requested functionality. Architects and developers will be interested in how to effectively implement this functionality. Customers will be interested in costs and timelines. Often compromises and tradeoffs need to be made to satisfy different stakeholders.

2.2 Requirements process in different methodologies.

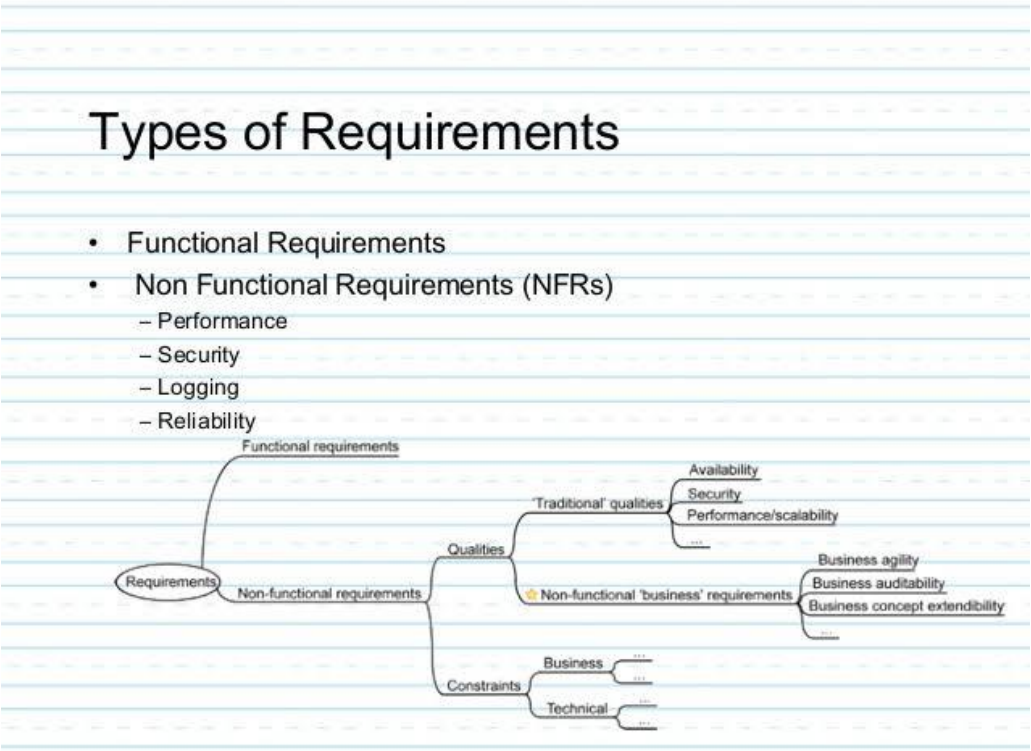
Although different methodologies provide different techniques for requirements engineering, all of them follow the same requirements process: requirements gathering, requirements analysis, and requirements specification. The process starts with customer's requirements or surveying the potential market and ends with a specification document that details how the system-to-be will behave. This is simply a logical ordering of requirements engineering activities, regardless of the methodology that is used. Of course, the logical order does not imply that each step must be perfectly completed before the next is taken.

Requirements gathering (also known as "requirements elicitation") help the developer understand the business context. The customer needs to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis. This turns out to be very hard to achieve, as discussed in Section. The statement of work is rarely precise and completes enough for the development team to start working on the software product.

Requirements analysis involves refining of and reasoning about the requirements received from the customer during requirements gathering. Analysis is driven by the creation and elaboration of user scenarios that describe how the end-user will interact with the system. Negotiation with the customer will be needed to determine the priorities, what is essential, and what is realistic. A popular tool is the use cases. It is important to ensure that the developer's understanding of the problem coincides with the customer's understanding of the problem.

Requirements specification represents the problem statement in a semiformal or formal manner to ensure clarity, consistency, and completeness. It describes the function and quality of the software-to-be and the constraints that will govern its development. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these. The developers could use UML or another symbol language for this purpose.

2.3 Types of Requirements



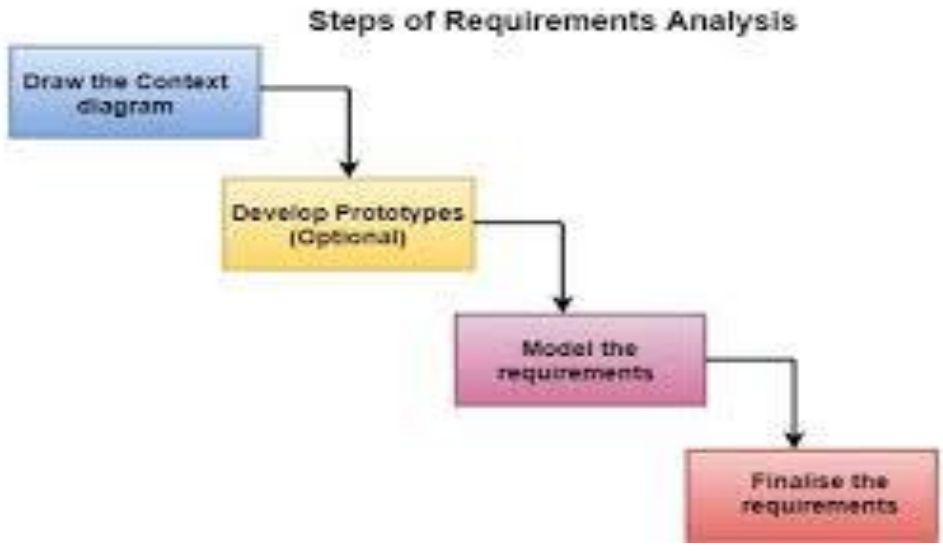
System requirements make explicit the characteristics of the system-to-be. Requirements are usually divided into functional and non-functional. Functional requirements determine the system's expected behavior and the effects it should produce in the problem domain. These requirements generally represent the main product features.

Non-functional requirements describe some quality characteristic that the system-to-be shall exhibit. They are also known as "quality" or "emergent" requirements, or the "laities" of the system-to-be. An example non-functional requirement is: Maintain a persistent data backup, for the cases of power outages.

The non-functional system properties:

- Functionality lists additional functional requirements that might be considered, such as security, which refers to ensuring data integrity and authorized access to information
- Usability refers to the ease of use, esthetics, consistency, and documentation—a system that is difficult and confusing to use will likely fail to accomplish its intended purpose
- Reliability specifies the expected frequency of system failure under certain operating conditions, as well as recoverability, predictability, accuracy, and mean time to failure
- Performance details the computing speed, efficiency, resource consumption, throughput, and response time

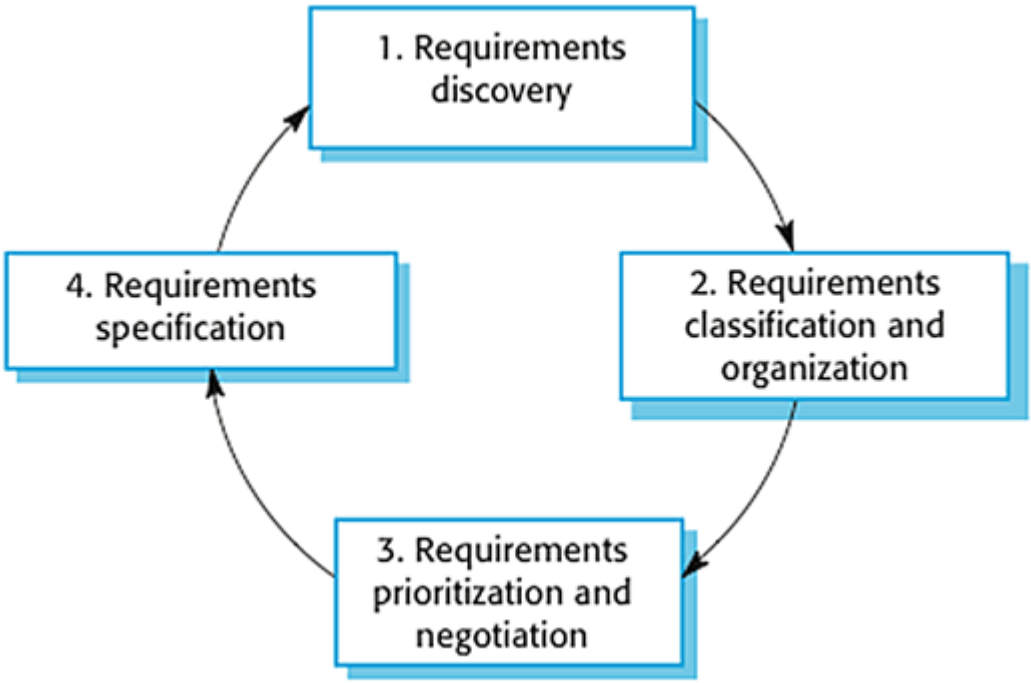
2.4 Requirements elicitation and analysis



After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

A process model of the elicitation and analysis process. Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

2.3.1Requirements discovery



This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.

Requirements classification and organization

This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.

Requirements prioritization and negotiation

Inevitably, when multiple stake-holders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

Requirements specification

The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced, as discussed.

2.5 Building the Analysis Model

We start with the customer statement of work (also known as customer statement of requirements), if the project is sponsored by a specific customer, or the vision statement, if the project does not have a sponsor. The statement of work describes what the envisioned system-to-be is about, followed by a list of features/services it will provide or tasks/activities it will support.

Given the statement of work, **the first step in the software development process is called requirements analysis or systems analysis.** During this activity the developer attempts to understand the problem and delimit its scope. The result is an elaborated statement of requirements.

The goal is to produce the system specification—the document that is an exact description of what the planned system-to-be is to do. Requirements analysis delimits the system and specifies the

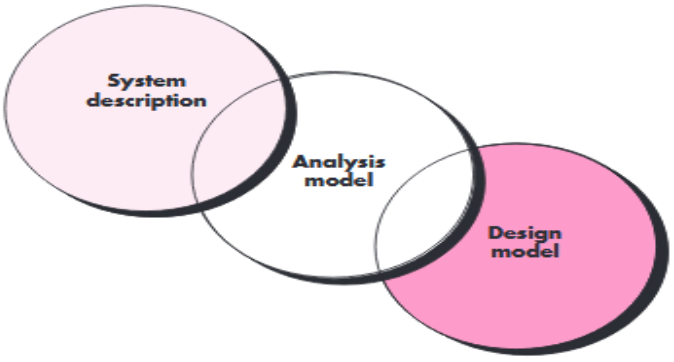
services it offers, identifies the types of users that will interact with the system, and identifies other systems that interact with ours.

The system is at first considered a black box, its services ("push buttons") are identified, and typical interaction scenarios are detailed for each service. Requirement analysis includes both fact-finding of how the problem is solved in the current practice as well as envisioning how the planned system might work. However, this may be too great a leap for a complex system.

A popular technique for requirements analysis is use case modelling. A set of use cases describes the elemental tasks a system is to perform and the relation between these tasks and the outside world. Each use case description represents a dialog between the user and the system, with the aim of helping the user achieve a business goal. In each dialog, the user initiates actions and the system responds with reactions.

Requirements analysis

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you (regardless of whether you're called a software engineer, an analyst, or a modeler) to elaborate on basic requirements established during the inception, elicitation, and



negotiation tasks that are part of requirements engineering (Chapter 5).The requirements modeling action results in one or more of the following types of models:

- ✓ Scenario-based models of requirements from the point of view of various system "actors"

- ✓ Data models that depict the information domain for the problem.

Class-oriented models that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system Behavioral models that depict how the software behaves as a consequence of external "events" These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model (and the software requirements specification) provides the developer and the customer with the means to assess quality once software is built. In this chapter, I focus on scenario-based modeling—a technique that is growing increasingly popular throughout the software engineering community; data modeling—a more specialized technique that is particularly appropriate when an application must create or manipulate a complex.

Modeling a representation of the object-oriented classes and the resultant collaborations that allow a system to function.

- ➢ **Overall Objectives and Philosophy**

Throughout requirements modeling, your primary focus is on what, not how. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the sys-tem exhibit, what interfaces are defined, and what constraints apply?2In earlier chapters, I noted that complete specification of requirements may not be possible at this stage. The customer may be unsure of precisely what is required for certain aspects of the system. The developer may be unsure that a specific approach will properly accomplish function and performance. These realities mitigate in favor of an iterative approach to requirements analysis and modeling. The analyst should model what is known and use that model as the basis for design of the soft-ware increment.3The requirements model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software de-sign, and (3) to define a set of requirements that can be validated once the software is built. The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying soft-ware, hardware, data, human, and other system

elements and a software design(Chapters 8 through 13) that describes the software's application architecture, user interface, and component-level structure.

- It should be noted that as customers become more technologically sophisticated, there is a trend toward the specification of how as well as what. However, the primary focus should remain on what.
- Alternatively, the software team may choose to create a prototype (Chapter 2) in an effort to better understand requirements for the system

It is important to note that all elements of the requirements model will be directly traceable to parts of the design model. A clear division of analysis and design tasks between these two important modeling activities is not always possible. Some design invariably occurs as part of analysis, and some analysis will be conducted during design.

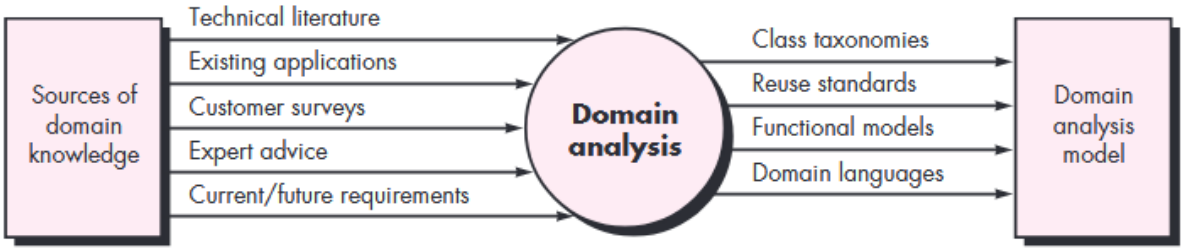
- **Analysis Rules of Thumb**

Arlow and Neustadt [Arl02] suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:•The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high. "Don't get bogged down in details" [Arl02] that try to explain how the system will work. Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system. Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed. Minimize coupling throughout the system. It is important to represent relation-ships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it. Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stake-holders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests. Keep the model as simple as it can be. Don't create

additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

- **Domain Analysis**

In the discussion of requirements engineering (Chapter 5), I noted that analysis pat-terns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs.

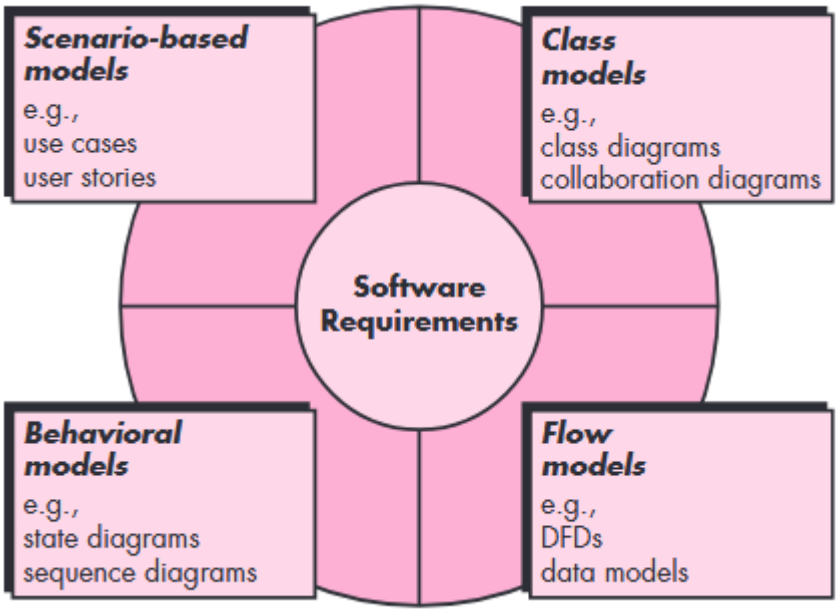


But how are analysis patterns and classes recognized in the first place? Who de-fines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in domain analysis. Firesmith [Fir93] describes domain analysis in the following way Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain typically for reuse on multiple project swithin that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks. The "specific application domain" can range from avionics to banking, from multi-media video games to software embedded within medical devices. The goal of do-main analysis is straightforward: to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.4Using terminology that was introduced earlier in this book, domain analysis maybe viewed as an umbrella activity for the software process. By this I mean that do-main analysis is an ongoing software engineering activity that is not connected to any

one software project. In a way, the role of a domain analyst is similar to the role of a master tool smith in a heavy manufacturing environment. The job of the tool-smith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst5is to discover and de-fine analysis patterns, analysis classes, and related information that may be used by many people working on similar but not necessarily the same applications.

- **Requirements Modeling Approaches**

One view of requirements modeling, called structured analysis, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system. A second approach to analysis modeling, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process is predominantly object oriented. Although the requirements model proposed in this book combines features of both approaches, software teams often choose one approach and exclude all representations from the other. The question is not which is best, but rather, what



Combination of representations will provide stakeholders with the best model of software requirements and the most effective bridge to software design. Each element of the requirements model (Figure 6.3) presents the problem from a different point of view. Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used. Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships (some hierarchical) between the objects, and the collaborations that occur between the classes that are defined. Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally, flow-oriented elements represent the system as information transform, depicting how data objects are transformed as they flow through various system functions. Analysis modeling leads to the derivation of each of these modeling elements. However, the specific content of each element (i.e., the diagrams that are used to construct the element and the model) may differ from project to project. As we have noted a number of times in this book, the software team must work to keep it simple. Only those modeling elements that add value to the model should be used.

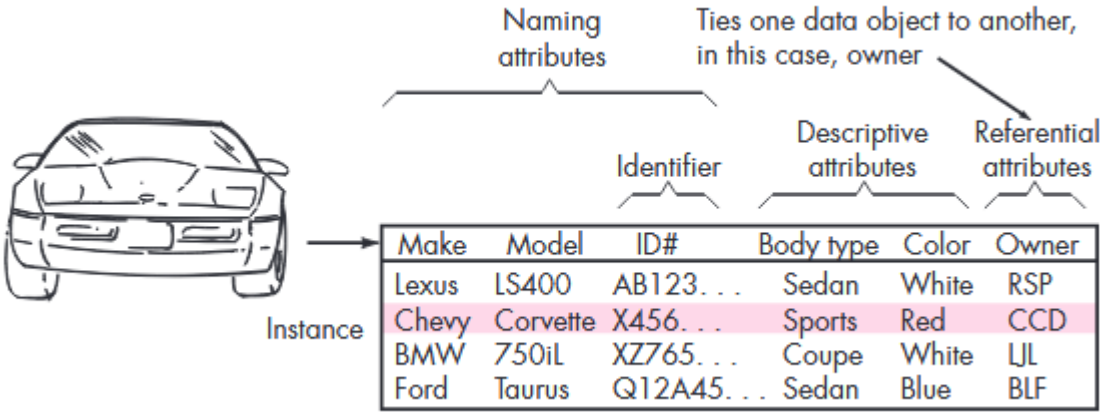
Data modeling concepts

If software requirements include the need to create, extend, or interface with a data-base or if complex data structures must be constructed and manipulated, the soft-ware team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application.

- **Data Objects**

A data object is a representation of composite information that must be understood by software. By composite information, I mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions(incorporating height, width, and depth) could be defined as an object.A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence

(e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes. A data object encapsulates data only—there is no reference within a data object to operations that act on the data.10Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car.6.4.2 Data Attributes Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier



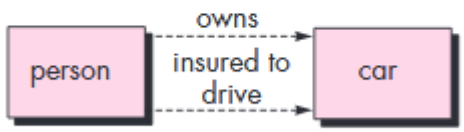
Attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Refer-ring to the data object car, a reasonable identifier might be the ID number. The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for car might serve well for an application that would be used by a department of motor vehicles, but these

attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for car might also include I number, body type, and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car meaningful object in the manufacturing control context.

- **Relationships**

Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the simple notation

(a) A basic connection between data objects



(b) Relationships between data objects

A connection is established between person and car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/relationship pairs that define the relevant relationships. For example, A person owns car. A person is insured to drive car. The relationships own and insured to drive define the relevant connections between person and car. Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations.

Two marks

What is Cost model?

Cost estimation models are mathematical algorithms or parametric equations used to estimate the costs of a product or project. The results of the models are business plans, budgets, and other financial planning and tracking mechanisms.

Define Software Requirements?

A software requirement is a description of a software system to be developed. It lays out functional and non-functional requirements, and may include a set of use cases that describe user interactions that the software must provide.

What are Software estimation techniques?

The Project Estimation Approach that is widely used is Decomposition Technique. Decomposition techniques take a divide and conquer approach. Size, Effort and Cost estimation are performed in a stepwise manner by breaking down a Project into major Functions or related Software Engineering Activities.

Define Parkinson's Law.

Parkinson's Law is the tendency for the amount of work required for something to increase so that it consumes any amount of time that may be allotted to it.

Write the major factors that influence Software cost.

- ✓ Program ability
- ✓ Product complexity
- ✓ Product size
- ✓ Available time
- ✓ Required reliability
- ✓ Level of technology

What is software cost estimation?

It is an algorithmic approach to estimating the cost of a software project. For example: By using COCOMO we can calculate the amount of effort and the time schedule for projects.

List out the techniques used in software cost estimation.

- ✓ Expert judgment
- ✓ Delphi cost estimation
- ✓ Work breakdown structure
- ✓ Algorithm cost models.

What do you know about algorithm cost models?

Algorithmic cost estimators compute the estimated cost of software system as the some of the cost of the module this model is bottom up estimates. The constructive cost model (COCOMO) is an algorithmic cost model described by Boehm estimation.

What is expert judgment?

Expert Judgment is a term that refers a specifically to a technique in which judgment is made based upon a specific set of criteria and/or expertise that has been acquired in a specific knowledge area, or product area, a particular discipline, an industry, etc.

Write the desirable properties of software requirement specification.

- ✓ Correct
- ✓ Complete
- ✓ Consistent
- ✓ Unambiguous
- ✓ Functional
- ✓ Verifiable
- ✓ Easily changed

Define structured system analysis.

Structured analysis is a software engineering technique that uses graphical diagrams to develop and portray system specifications that are easily understood by users.

Write the desirable properties of software specification.

Product overview and summary

- ✓ Development, operating, and maintenance environments
- ✓ External interfaces and data flow Functional requirements Performance requirements Exception handling

What are the basic features of SSA?

- ✓ Data flow diagrams
- ✓ Data dictionaries
- ✓ Procedure logic representation
- ✓ Data store structuring techniques.

List of system directories in SRS?

- ✓ System input output flow
- ✓ System structure
- ✓ Data structure
- ✓ Data derivation
- ✓ System size and volume
- ✓ System dynamics
- ✓ System properties
- ✓ Project management

What is Delphi Cost estimation?

- ✓ This technique was developed at the rand corporation in 1948
- ✓ This technique can be adapted to software estimation
- ✓ Estimators study the document and complete their estimates.

What is SRS?

Software Requirement Specification (SRS) is a technical specification of requirements for the s/w products. The requirements specification will state that what of the s/w product without implying how.

What is Rayleigh curve?

- ✓ Rayleigh curve represents the number of fulltime equivalent personnel required at the instant in time.
- ✓ Rayleigh curve is specified by two parameters
- ✓ Td-the time at which the curve reaches its maximum value
- ✓ k-the total area under the curve (ie) the total effort required for the project

What is state oriented notation?

State oriented notations include

- ✓ Decision tables.
- ✓ Even tables.
- ✓ Transition tables.
- ✓ Finite state mechanism.
- ✓ Pettiness.

What are the various types of reports?

- ✓ Database modification reports.
- ✓ Reference reports.
- ✓ Summary reports
- ✓ Analysis report

Five mark

Write the features of specification techniques? Explain.

Functional characteristics of an s/w product are one of the most important activities to be then during the requirement analysis.

- ✓ The advantage of formal notation is concise and ambiguous.
- ✓ They provide the basis for verification of the resulting s/w product.
- ✓ Two nations are used to specify the functional characteristics.
  - Relational
  - State oriented

Relational notations

- ✓ It is based on the concept of entities and attributes.
- ✓ Entities are named elements in a system.
- ✓ The names are chosen to denote the nature of the elements.
- ✓ E.g.: stack, queue
- ✓ Attributes are specified by applying functions and relations to the named entities
- ✓ Attributes specify permitted operations on entities, relationships among entities and data flow between entities. Relational notations include,
  - Implicit equations
  - Recurrence relations
  - Algebraic axioms
  - Regular Expressions State oriented notations include
  - Decision tables.
  - Even tables.
  - Transition tables.
  - Finite state mechanism.

Write the format of a software requirements specification.

- ✓ Section1: Product overview and summary
- ✓ Section2: Development, operating, and maintenance environments
- ✓ Section3: External interfaces and data flow
- ✓ Section4: Functional requirements
- ✓ Section5: Performance requirements
- ✓ Section6: Exception handling
- ✓ Section7: Early subsets and implementation priorities
- ✓ Section8: Foreseeable modifications and enhancements
- ✓ Section9: Acceptance criteria
- ✓ Section10: design Hints and Guidelines

Why to understand and specifying requirements?

- ➢ A software requirement is defined as a condition to which a system must comply.
- ➢ Software specification or requirements management is the process of understanding and defining what functional and non-functional requirements are required for the system
- ➢ It identifies the constraints on the system's operation and development.
- ➢ The requirements engineering process results in the production of a software requirements document that is the specification for the system.

There are four main phases in the Specifying requirements:

- ➢ **Feasibility study**: To estimate user needs may be satisfied using current software and hardware technologies.
- ➢ **Requirements elicitation and analysis:** This is the process of deriving the system requirements through observation of existing systems, discussions with potential users, requirements workshop, storyboarding, etc.
- ➢ **Requirements specification**: This is the activity of translating the information gathered during the analysis activity into a document that defines a set of requirements. Two types of requirements may be included in this document: user (functional) requirements and system (non-functional) requirements.

UNIT –III

Data Engineering

Introduction

Software design encompasses the set of principles, concepts, and practices that lead to the development of a high-quality system or product. Design concepts must be understood before the mechanics of design practice are applied, and design practice itself leads to the creation of various representations of the software that serve as a guide for the construction activity that follows.

What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. . . .

According to Mitch Kapor? The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.

- ✓ *Firmness:* A program should not have any bugs that inhibit its function.
- ✓ *Commodity:* A program should be suitable for the purposes for which it was intended.
- ✓ *Delight:* The experience of using the program should be a pleasurable one.

To accomplish the goal you must practice diversification and then convergence. Diversification is the acquisition of a components, component solutions, and knowledge, all contained in catalogues, textbooks, and the mind. Once this diverse set of information is assembled, you must pick and choose elements from the repertoire that meet the requirements defined by requirements engineering and the analysis model. As this occurs, alternatives are considered and rejected and you converge on "one particular configuration of components, and thus the creation of the final product".

3.1 Design with the context of Software Engineering

After the software requirements have been analysed and modelled, software design is the last step in the software engineering action within the modelling activity that sets the stage for **construction** (code generation and testing)**.**

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 3.1. The requirements model is manifested by scenario-based, class-based, flow-oriented, and behavioural elements, feed the design task.

The data/class design transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action.

The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which

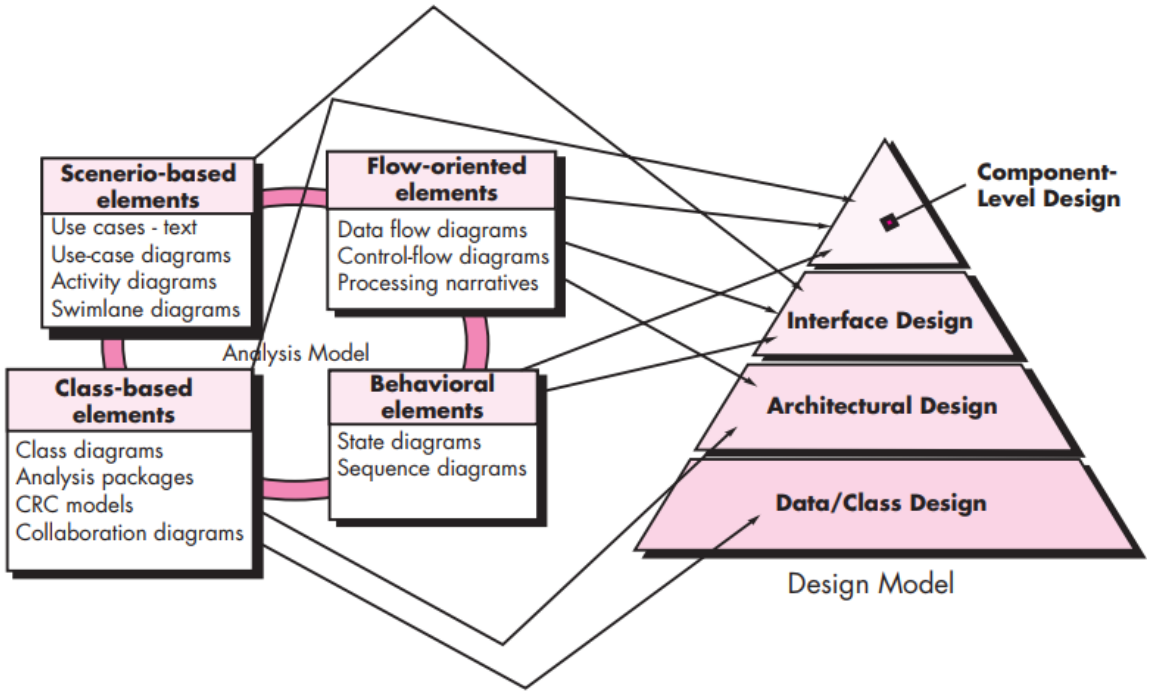


Fig 3.1 Translating the requirements model into the design model

architecture can be implemented.

The interface design describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behaviour. Therefore, usage scenarios and behavioural models provide much of the information required for interface design.

The component-level design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioural models serve as the basis for component design.

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system.

3.2 The Design Process and Design Quality

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews

There are **three characteristics** that serve as a guide for the evaluation of a good design:

- ✓ The design must implement all of the explicit requirements contained in the requirements model, and it must accommodate all of the implicit requirements desired by stakeholders (Firmness).
- ✓ The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software (Delight).
- ✓ The design should provide a complete picture of the software, addressing the data, functional, and behavioural domains from an implementation perspective (Commodity).

Quality Guidelines: In order to evaluate the quality of a design representation, the software team have to consider the following guidelines.

1. A design should exhibit an architecture that,
   (1) has been created using recognizable architectural styles or patterns,
   (2) is composed of components that exhibit good design characteristics, and
   (3) can be implemented in an evolutionary fashion,2 thereby facilitating implementation and
   Testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes: Software quality attributes that has been given the acronym FURPS—functionality, usability, reliability, performance, and supportability.

Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.

Usability is assessed by considering human factors, overall aesthetics, consistency, and documentation.

Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure, and the predictability of the program.

Performance is measured by considering processing speed, response time, resource consumption, throughput, and efficiency.

Supportability combines the ability to extend the program (extensibility), adaptability, serviceability.

3.3 Design Concepts: In this we are going to discuss and overview of important software design concepts that span both traditional and object-oriented software development.

3.3.1 Abstraction

When you consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more detailed description of the solution is provided.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* refers to a sequence of instructions that have a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.)

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door.** Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).

3.3.2 Architecture

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of properties has been given for an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building block.

With the specification of these properties, the architectural design can be represented using one or more of a number of different models.

- *Structural models* represent architecture as an organized collection of program components.
- *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of applications.
- *Dynamic models* address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.
- *Process models* focus on the design of business or technical process that the system must accommodate.
- *Functional models* can be used to represent the functional hierarchy of a system.

3.3.3 Patterns

A design pattern "conveys the essence of a proven design solution to a recurring problem within a certain context amidst computing concerns."

A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.

The intent of each design pattern is to provide a description that enables a designer to determine:

1. whether the pattern is applicable to the current work,
2. whether the pattern can be reused (hence, saving design time), and
3. Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

3.3.4 Separation of Concerns

Separation of concerns is a design concept [Dij82] that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A *concern* is a feature or behaviour that is specified as part of the requirements model for the software. By separating concerns into smaller and therefore more manageable pieces, a problem takes less effort and time to solve.

3.3.5 Modularity

Modularity is the single attribute of software that allows a program to be intellectually manageable. It is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called *modules that* are integrated to satisfy problem requirements. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

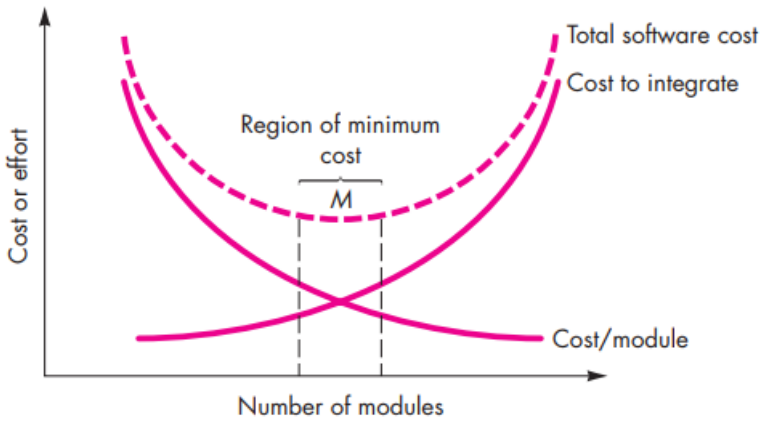


Fig 3.2 Modularity and software cost

Referring to Figure 3.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grow. These characteristics lead to a total cost or effort curve shown in the figure. The curves shown in Fig 3.2 do provide useful qualitative guidance when modularity is considered.

3.3.6 Information Hiding

It is about controlled interfaces. Modules should be specified and design so that information (algorithm and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining by a set of independent modules that communicate with one another only that information necessary to achieve software function.

The use of Information Hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to other location within the software.

3.3.7 Functional Independence

The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.

Independence is assessed using two qualitative criteria: cohesion and coupling.

Cohesion is an indication of the relative functional strength of a module. Cohesion is a natural extension of the information-hiding concept. A cohesive module performs a single task, requiring

little interaction with other components in other parts of a program. A cohesive module should always strive for high cohesion (i.e., single-mindedness).

Coupling is an indication of the relative interdependence among modules. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand.

3.3.8 Refinement

Refinement is a process of *elaboration.* It is a top-down design strategy. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Abstraction and refinement are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low-level details. Refinement helps you to reveal low-level details as design progresses. Both concepts allow you to create a complete design model as the design evolves.

3.3.9 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure. When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary y algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The result will be software that is easier to integrate, easier to test, and easier to maintain.

3.3.10 Design Classes

The next step is to define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that

supports the business solution. Five different types of design classes, each representing a different layer of the design architecture, can be developed.

3.4 Architectural Design

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

3.4.1 Software Architecture

Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system. Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering

3.4.1 What is Architecture?

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. The architecture is not the operational software. Rather, it is a representation that enables you to

1. analyse the effectiveness of the design in meeting its stated requirements,
2. consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. Reduce the risks associated with the construction of the software.

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers.

There is a distinct difference between the terms architecture and design. A *design* is an instance of architecture similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. Therefore, you cannot mix "architecture" and "design" with each other.

3.4.2 Why Is Architecture Important?

There are three key reasons that software architecture is important:

- ✓ Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- ✓ The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- ✓ Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together".

The architectural design model and the architectural patterns contained within it are transferable. That is, architecture genres, styles, and patterns can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

3.4.3 Architectural Descriptions

An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building.

The IEEE Computer Society has proposed IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], with the following objectives:

(1) to establish a conceptual framework and vocabulary for use during the design of software architecture,

(2) to provide detailed guidelines for representing an architectural description, and

(3) to encourage sound architectural design practices.

The IEEE standard defines an *architectural description* (AD) as "a collection of products to document an architecture." The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns." A *view* is created according to rules and conventions defined in a *viewpoint*—"a specification of the conventions for constructing and using a view".

3.4.4 Architectural Decisions

Each view developed as part of an architectural description addresses a specific stakeholder concern. To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern.

Therefore, architectural decisions themselves can be considered to be one view of the architecture. The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns.

As a system architect, you can use the template suggested in the sidebar to document each major decision. By doing this, you provide a rationale for your work and establish an historical record that can be useful when design modifications must be made.

3.5 Architectural Design

The design should define the external entities (other systems, devices, and people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modelled and all external software interfaces have been described, you can identify a set of architectural archetypes. An *archetype* is an abstraction (similar to a class) that represents one element of system behaviour. The set of archetypes provides a collection of abstractions that must be modelled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

3.5.1 Representing the System in Context

At the architectural design level, a software architect uses an *architectural context diagram* (ACD) to model the manner in which software interacts with entities external to its boundaries. The generic structure of the architectural context diagram is illustrated in Figure 3.5. Referring to the figure, systems that interoperate with the *target system* (the system for which an architectural design is to be developed) are represented as

- *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
- *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system.
- *Actors*—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Each of these external entities communicates with the target system through an interface.

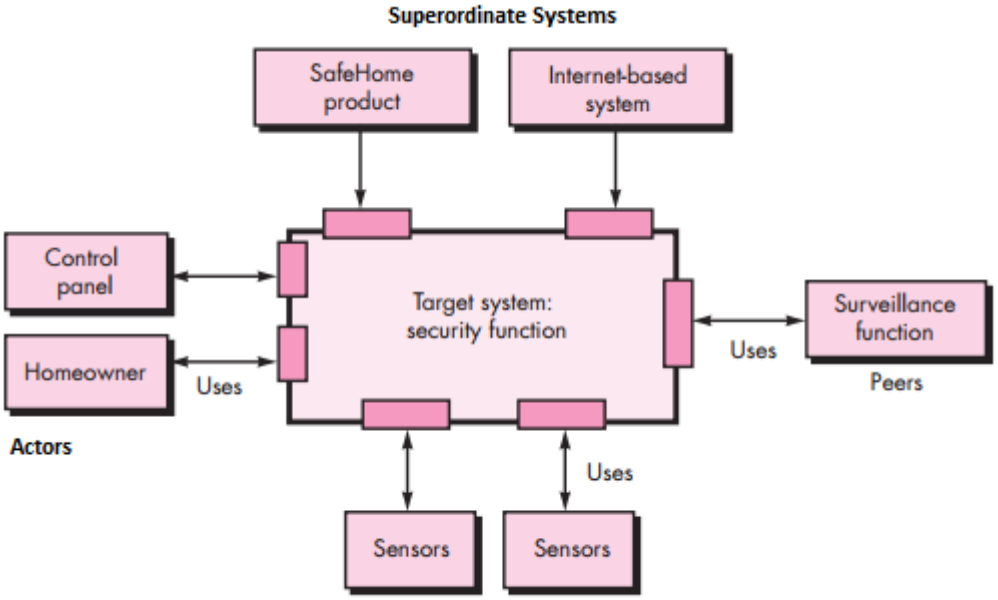


Fig 3.5 Architectural context diagram for the SafeHome security function

To illustrate the use of the ACD, consider the home security function of the *safe Home* product. The overall *Safe Home* product controller and the Internet-based system are both superordinate to the security function and are shown above the function in Figure 3.5. The surveillance function is a *peer system* and uses (is used by) the home security function in later versions of the product. The homeowner and control panels are actors that are both producers and consumers of information used/produced by the security software. Finally, sensors are used by the security software and are shown as subordinate to it.

As part of the architectural design, the details of each interface shown in Figure 3.5 would have to be specified. All data that flow into and out of the target system must be identified at this stage.

3.5.2 Defining Archetypes

An *archetype* is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small setof archetypes is required to design even relatively complex systems.

Archetypes can be derived by examining the analysis classes defined as part of the requirements model. Continuing the discussion of the *Safe Home* security function, you might define the following archetypes:

Node. Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of (1) various sensors and (2) a variety of alarm (output) indicators.

Detector. An abstraction that encompasses all sensing equipment that feeds information into the target system.

Indicator. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, and bell) for indicating that an alarm condition is occurring.

Controller. An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

3.5.3 Refining the Architecture into Components

As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management

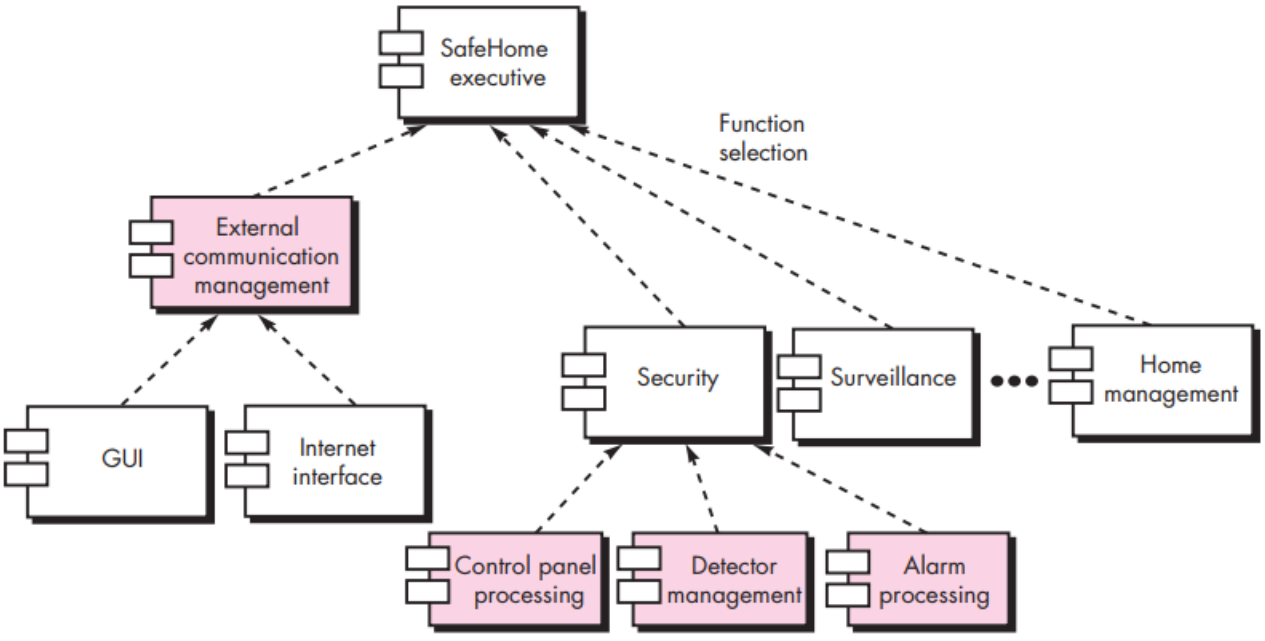


Fig 3.5.1 Overall architectural structure for SafeHome with top-level components

components are often integrated into the software architecture.

Continuing the *Safe Home* security function example, you might define the set of top-level components that address the following functionality:

External communication management—coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.

Control panel processing—manages all control panel functionality.

Detector management—coordinates access to all detectors attached to the system.

Alarm processing—verifies and acts on all alarm conditions.

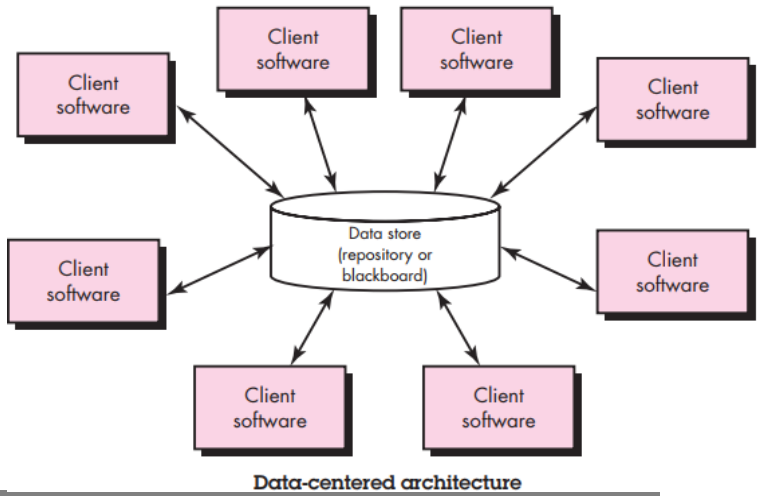
Each of these top-level components would have to be elaborated iteratively and then positioned within the overall *Safe Home* architecture. The overall architectural structure (represented as a UML component diagram) is illustrated in Figure 3.5.1. Transactions are acquired by *external communication management* as they move in from components that process the *Safe Home* GUI and the Internet interface. This information is managed by a *Safe Home* executive component that selects the appropriate product function (in this case security). The *control panel processing* component interacts with the homeowner to arm/disarm the security function. The *detector management* component polls sensors to detect an alarm condition, and the *alarm processing* component produces output when an alarm is detected.

3.6 Architectural Styles

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system. An architectural pattern, like an architectural style, imposes a transformation on the design of architecture.

3.6.1 A Brief Taxonomy of Architectural Styles

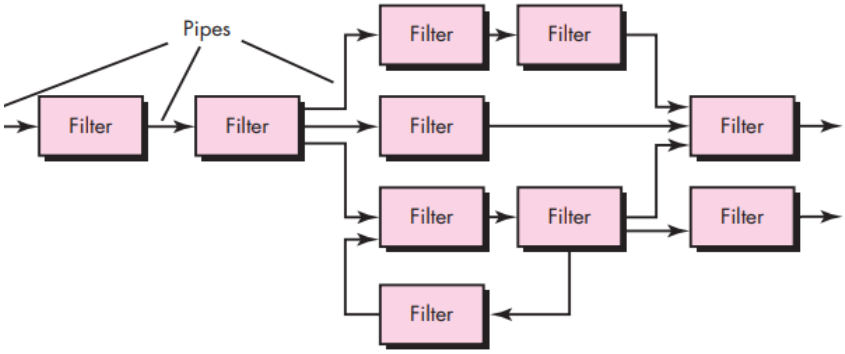
It can be categorized into one of a relatively small number of architectural styles as follows:



Data-centered architecture

Data-cantered architectures. A data store (e.g., a file or database) resides at the centre of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository. Data- cantered architectures promote *inerrability.* That is, existing components can be changed and new client components added to the architecture.

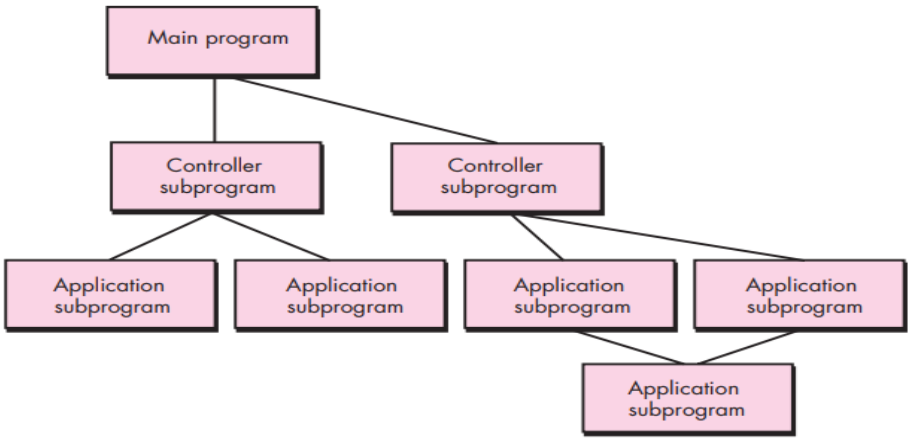
Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern has a set of components, called *filters*, connected by *pipes* that transmit data from one component to the next. Each filter works independently. If the data flow degenerates into a single line of transforms, it is termed batch sequential.



Data-flow architecture

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of sub styles exist within this category.

Main program/subprogram architectures. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program.

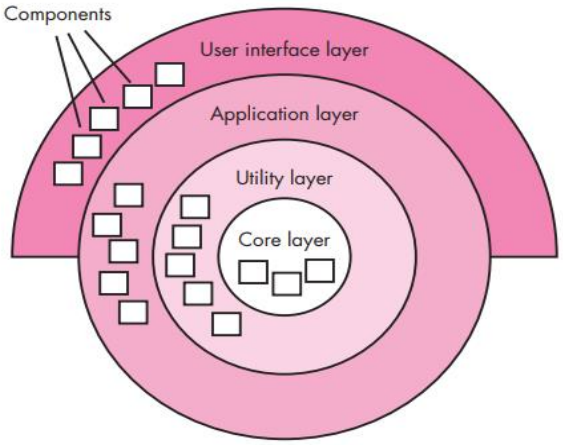


Main program/subprogram architecture

Remote procedure call architectures. The components of a main program/subprogram architecture are distributed across multiple computers on a network.

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components are accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated below. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



Layered architecture

3.7 Mapping Data Flow into Software Architecture

A mapping technique, called *structured design*, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a sixstep process:

(1) The type of information flow is established,

(2) Flow boundaries are indicated,

(3) The DFD is mapped into the program structure,

(4) Control hierarchy is defined,

(5) The resultant structure is refined using design measures and heuristics, and

(6) The architectural description is refined and elaborated.

As a brief example of data flow mapping, I present a step-by-step "transform" mapping for a small part of the *Safe Home* security function.8 In order to perform the mapping, the type of information flow must be determined. One type of information flow is called *transform flow* and exhibits a linear quality.

3.7.1 Transform Mapping

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. To map these data flow diagrams into software architecture, you would initiate the following design steps:

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 3.6.1 depicts a level 0 context model and Figure 3.6.2 shows refined data flow for the security function.

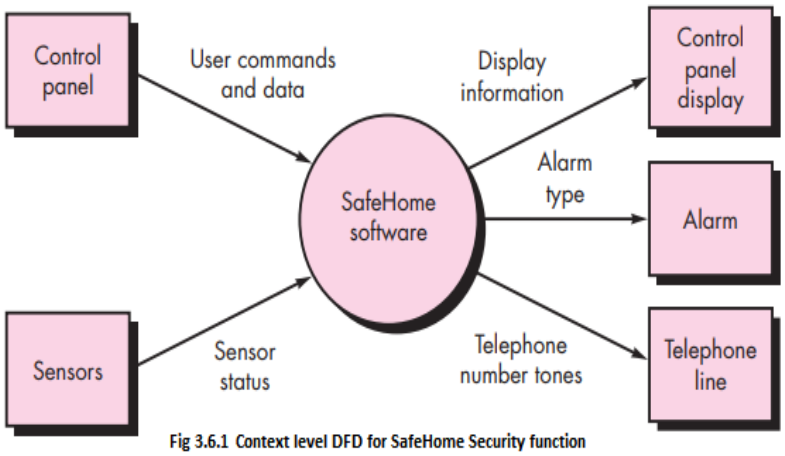


Fig 3.6.1 Context level DFD for SafeHome Security function

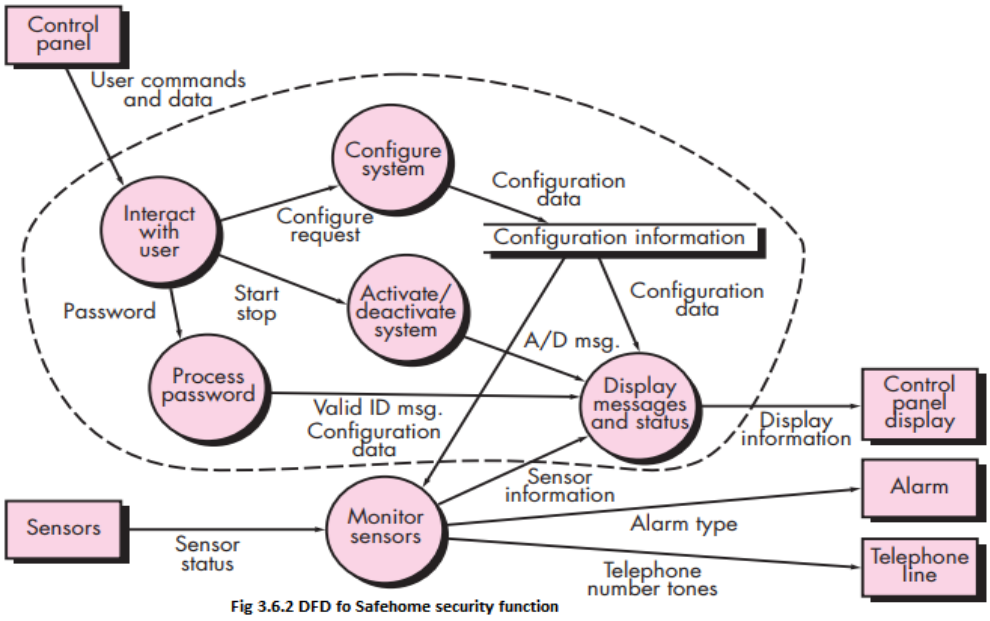


Fig 3.6.2 DFD fo Safehome security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for *monitor sensors* (Fig 3.6.3) is examined, and a level 3 data flow diagram is derived as shown in Fig 3.6.4. At level 3, each transform in the data flow diagram exhibits relatively high cohesion.

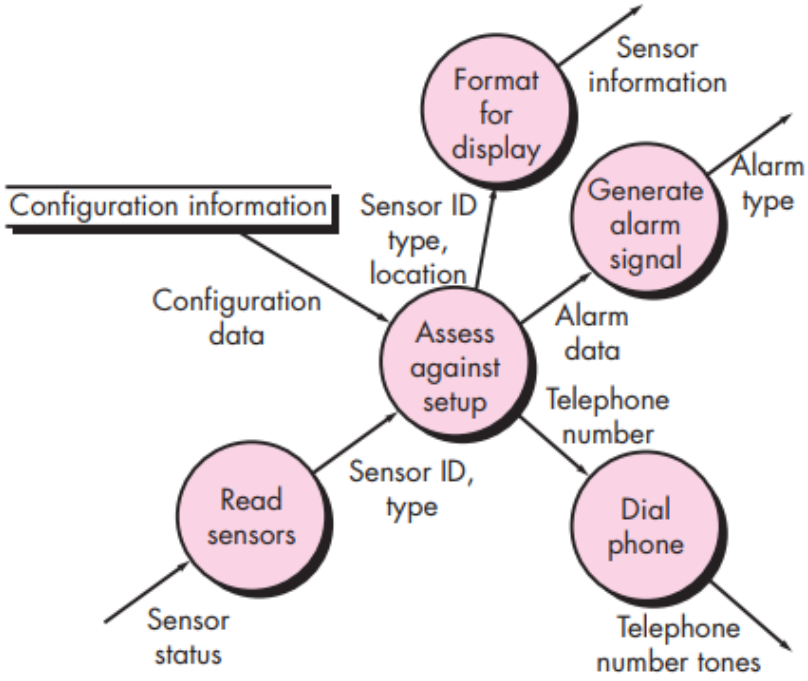


Fig 3.6.3 Level 2 DFD that refines the monitor sensors transform

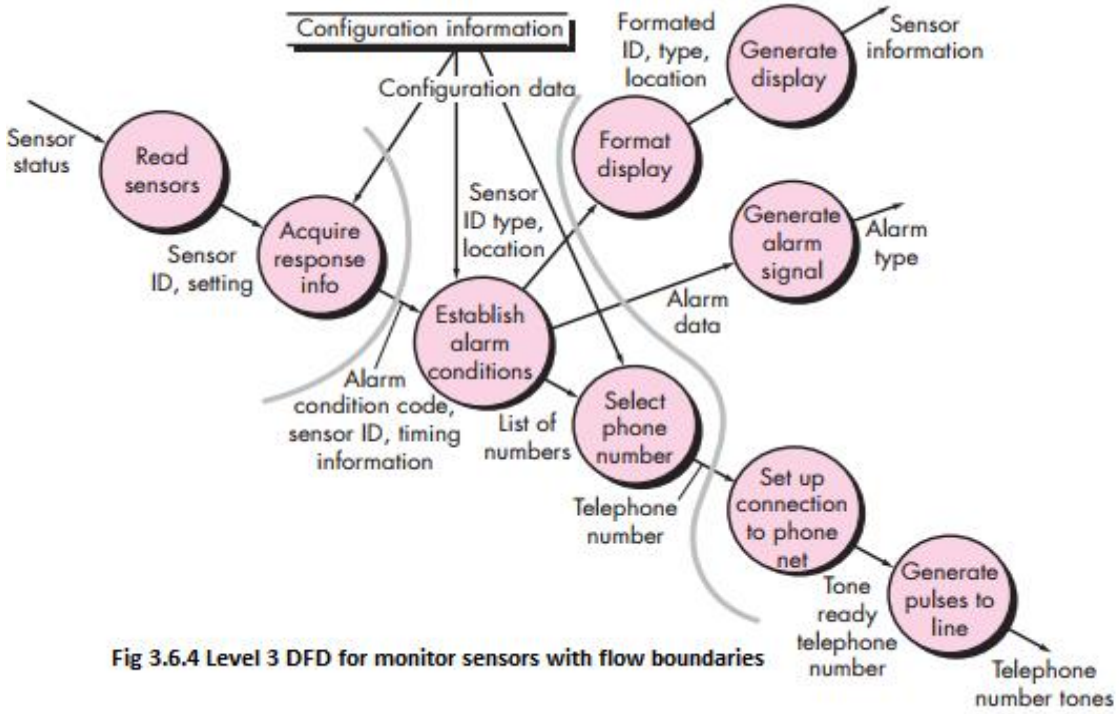


Fig 3.6.4 Level 3 DFD for monitor sensors with flow boundaries

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD (Fig 3.6.4), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Fig 3.6.4 The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. *Factoring* leads to a program structure in which top-level components perform decision making and low-level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work. This first-level factoring for the *monitor sensors* subsystem is illustrated in Fig 3.6.5.

A main controller (called *monitor sensors executive*) resides at the top of the program structure and coordinates the following subordinate control functions:

An incoming information processing controller, called *sensor input controller*, coordinates receipt of all incoming data.

A transform flow controller, called *alarm conditions controller*, supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

An outgoing information processing controller, called *alarm output controller*, coordinates production of output information.

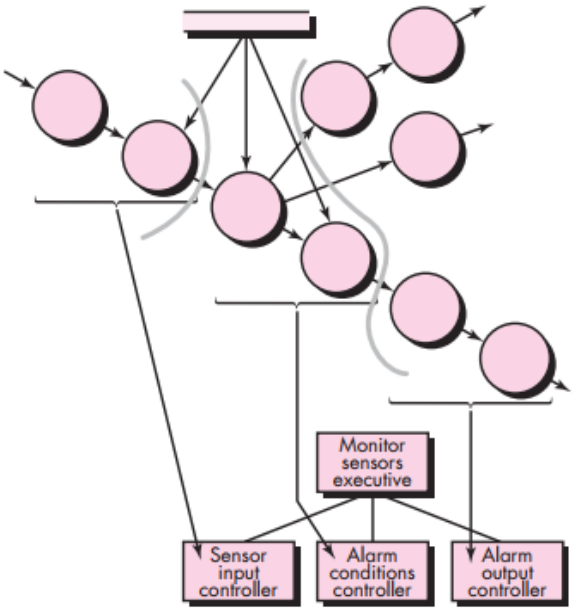


Fig 3.6.5 First-level factoring for monitor sensors

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to secondlevel factoring is illustrated in Fig 3.6.6

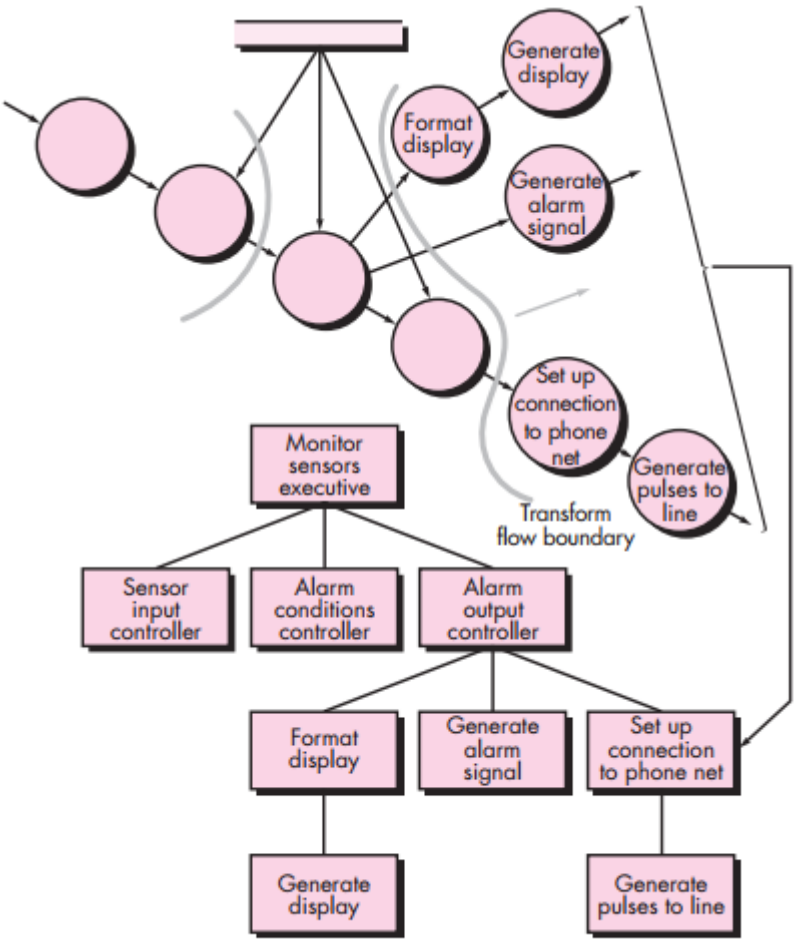


Fig 3.6.6 Second-level factoring for monitor sensors

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. First-iteration architecture can always be refined by applying concepts of functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality

3.7.2 Refining the Architectural Design

Refinement of software architecture during early stages of design is to be encouraged. An alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture. It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

3.8 Architectural Genres

The architectural genres describes the specific architectural approach to the structure that must be built. It implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on. In his evolving *Handbook of Software Architecture*,, Grady Booch suggests the following architectural genres for software-based systems:

Artificial intelligence—Systems that simulate or augment human cognition, locomotion, or other organic processes.

Commercial and nonprofit—Systems that are fundamental to the operation of a business enterprise.

Communications—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.

Content authoring—Systems that are used to create or manipulate textual or multimedia artifacts.

Devices—Systems that interact with the physical world to provide some point service for an individual.

Entertainment and sports—Systems that manage public events or that provide a large group entertainment experience.

Financial—Systems that provide the infrastructure for transferring and managing money and other securities.

Games—Systems that provide an entertainment experience for individuals or groups.

Government—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.

Industrial—Systems that simulate or control physical processes.

Legal—Systems that support the legal industry.

Medical—Systems that diagnose or heal or that contribute to medical research.

Military—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.

Operating systems—Systems that sit just above hardware to provide basic software services.

Platforms—Systems that sit just above operating systems to provide advanced services.

Scientific—Systems that are used for scientific research and applications.

Tools—Systems that are used to develop other systems.

Transportation—Systems that control water, ground, air, or space vehicles.

Utilities—Systems that interact with other software to provide some point services

3.9 User Interface Design

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

A software engineer designs the user interface by applying an iterative process that draws on predefined design principles. If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

3.10 The Golden Rules

For a Good User Interface design, a software engineer have to follow the three golden rules.

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

3.10.1 Place the user in control

A Software engineer have to consider the number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking code. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multitouch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard command (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file

management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

3.10.2 Reduce the user's memory load.

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. The design principles that enable an interface to reduce the user's memory load are:

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function.

3.10.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that

1. All visual information is organized according to design rules that are maintained throughout all screen displays,
2. Input mechanisms are constrained to a limited set that is used consistently throughout the application, and
3. Mechanisms for navigating from task to task are consistently defined and implemented.

For these reasons a set of design principles have to be considered to make the interface consistent. They are:

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what

alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

Questions and Answers

2Marks:

1. What are the properties that should be exhibited by a good software design?

   The goal of design is to produce a model or representation that exhibits firmness, commodity, and delight.

   - ✓ *Firmness:* A program should not have any bugs that inhibit its function.
   - ✓ *Commodity:* A program should be suitable for the purposes for which it was intended.
   - ✓ *Delight:* The experience of using the program should be a pleasurable one.

2. What are the components/elements required for a design model?

- ✓ Component level design
- ✓ Interface design
- ✓ Architectural design
- ✓ Data/Class design

3. What is cohesion?

Cohesion is a measure that defines the degree of intra-dependability within elements of a module. The greater the cohesion, the better is the program design.

4. What is coupling?

Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

5. Name the commonly used architectural styles.

- Data centered Architecture.
- Data flow Architecture.
- Call and return Architecture.
- Object-oriented Architecture.
- Layered Architecture

6. What is an Architectural design?

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

7. What are the steps involved in architectural design?
    1. Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system.
    2. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes.
    3. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

8. What is architectural style?
   An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

9. What is Architectural genres?
   The architectural genres describes the specific architectural approach to the structure that must be built. It implies a specific category within the overall software domain.

10. What is Transform mapping?
    The transform mapping is a set of design steps applied on the DFD in order to map the transformed flow characteristics into specific architectural style.

11. What is user interface design?
    User interface design creates an effective communication medium between a human and a computer.

12. Why user interface design is important?
    If software is difficult to use, if it forces mistakes, or if it frustrates the efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

13. What are the steps involved in designing the user interface?
    1. User interface design begins with the identification of user, task, and environmental requirements.
    2. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions.
    3. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items.

14. Define archetypes?
    An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems.

15. How IEEE Standard define architectural description?
    The IEEE standard defines an *architectural description* (AD) as a collection of products to document an architecture. The description itself is represented using multiple views, where each *view* is "a representation of a whole system from the perspective of a related set of concerns".

16. What are the objectives proposed by the IEEE Computer society for practising Architectural Description?
    1. To establish a conceptual framework and vocabulary for use during the design of software architecture,
    2. To provide detailed guidelines for representing an architectural description, and
    3. To encourage sound architectural design practices.

Answer in detail

1. Explain the software quality guidelines and attributes for the design process.
2. Explain the components of design model/ explain how the requirement model is translated into the design model.
3. Briefly explain the design concept.
4. Explain the software Architecture in detail.
5. Explain the Architecture Genres for software based systems.
6. Explain the Architecture styles in detail.
7. Explain how the software interoperates with one another.
8. Discuss the six steps involved in mapping the data flow into software.
9. Briefly explain the golden rules for performing user interface design.

Unit-IV

4.1 A STRATEGIC APPROACH TO SOFTWARE TESTING

Testing is a set of activities that can be planned in advance and conducted systematically it is called as **software testing**

a set of steps into which we can place specific test-case design techniques and testing methods.

4.1.1 Software testing strategies generic characteristics:

- To perform effective testing, you should conduct effective technical re-views by doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

4.1.2 Verification and validation (V&V).

Verification refers to the set of tasks that ensure that software correctly implements a specific function.

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Boehm states

Verification: **"Are we building the product right?"**

Validation: "Are we building the right product?"

4.2 A SOFTWARE TESTING STRATEGY FOR CONVENTIONAL SOFTWARE ARCHITECTURE

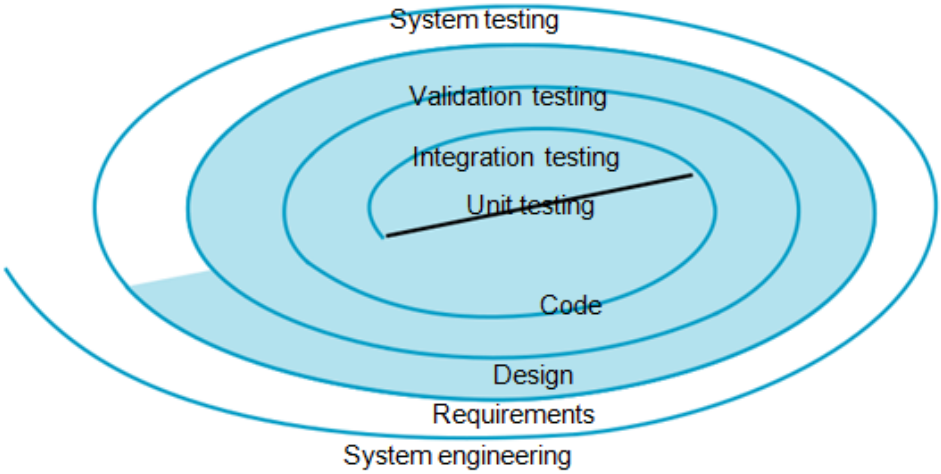


Fig: 1

Testing within the context of software engineering is actually a series of four steps that are implemented sequentially.

Tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name *unit testing.*

Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Integration testing addresses the issues associated with the dual problems of verification and program construction.

Test-case design techniques that focus on inputs and out-puts are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated

a set of *high-order tests* is conducted. Validation criteria (established during requirements analysis) must be evaluated.

Validation testing provides final assurance that software meets all functional, behavioural, and performance requirements.

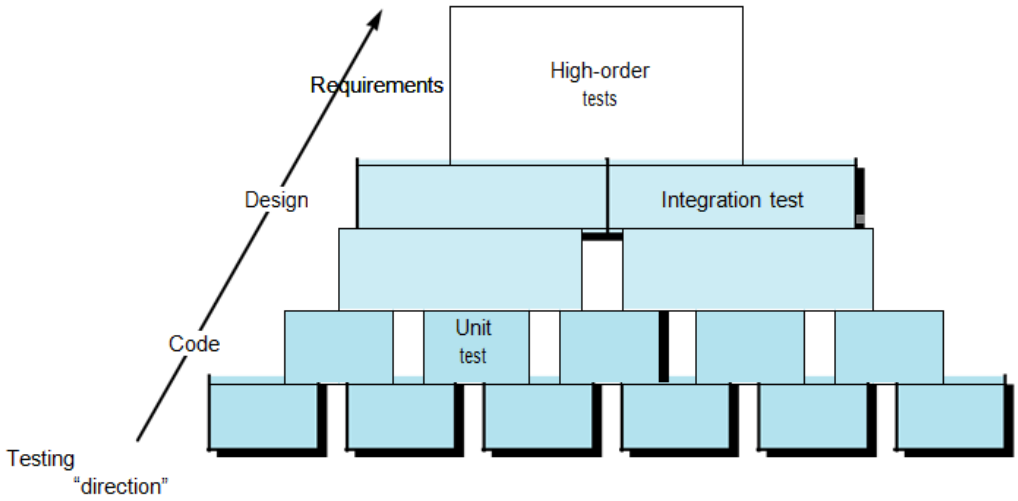


Fig :2

4.2 .1 TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

A testing strategy that is chosen by many software teams falls between the two extremes. It takes an incremental view of testing, beginning with the testing of individual program units, moving to tests designed to facilitate the integration of the units (sometimes on a daily basis), and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

4.2.1 .1 Unit Testing

Unit testing focuses verification effort on the smallest unit of software design— the software component or module.

Component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and the errors those tests uncover is limited by the constrained scope established for unit testing.

The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

4.2.1.1.2 Unit Test Considerations.

The module interface is tested to ensure that information properly flows into and out of the program unit under test.

Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.

All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing

Data flow across a component interface is tested before any other testing is initiated. If data do not enter and exit properly, all other tests are moot. In ad-dition, local data structures should be exercised and the local impact on global data should be ascertained (if possible) during unit testing.

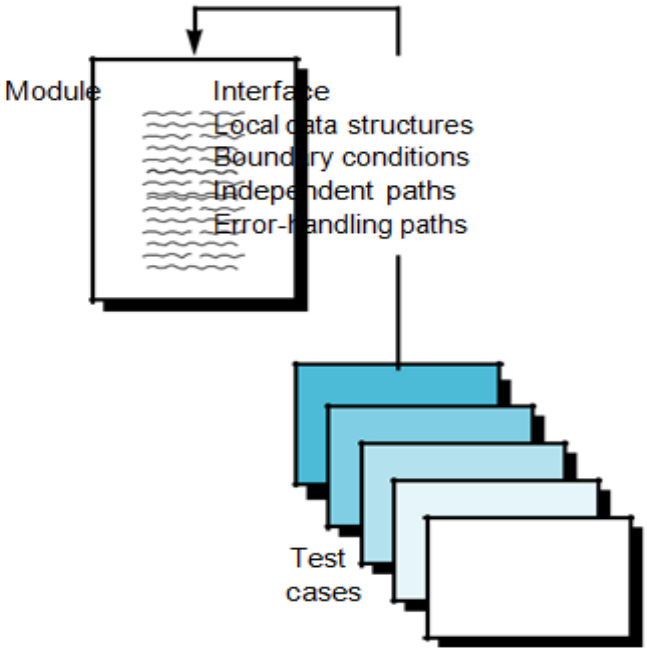


Fig : 3

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur its called as *antibugging*..

Among the potential errors that should be tested when error handling is evaluated are:

1. error description is unintelligible,
2. error noted does not correspond to error encountered,
3. error condition causes system intervention prior to error handling,
4. exception-condition processing is incorrect,
5. error description does not provide enough information to assist in the location of the cause of the error

Because a component is not a stand-alone program, driver and/or stub soft-ware must often be developed for each unit test.

n most applications a *driver* is nothing more than a "main program" that accepts test-case data, passes such data to the component (to be tested), and prints relevant results.

Stubs serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

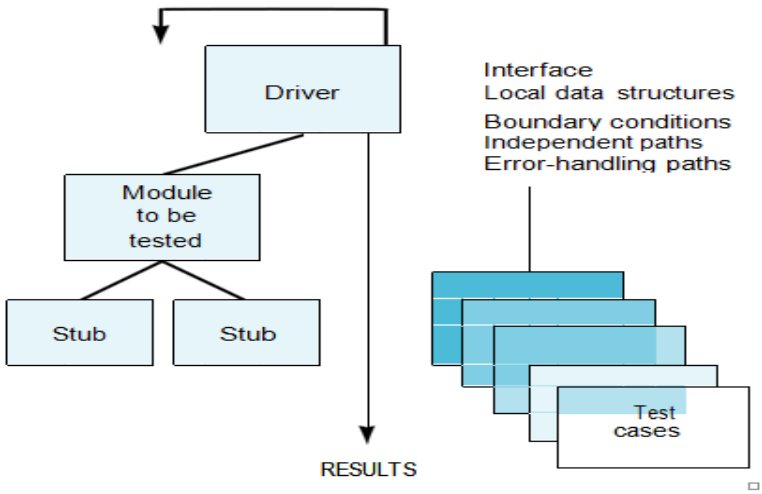


Fig : 4

4.2.1.3 Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

The objective is to take unit tested components and build a program structure that has been dictated by design.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "big bang" approach. All components are combined in advance and the entire program is tested as a whole.

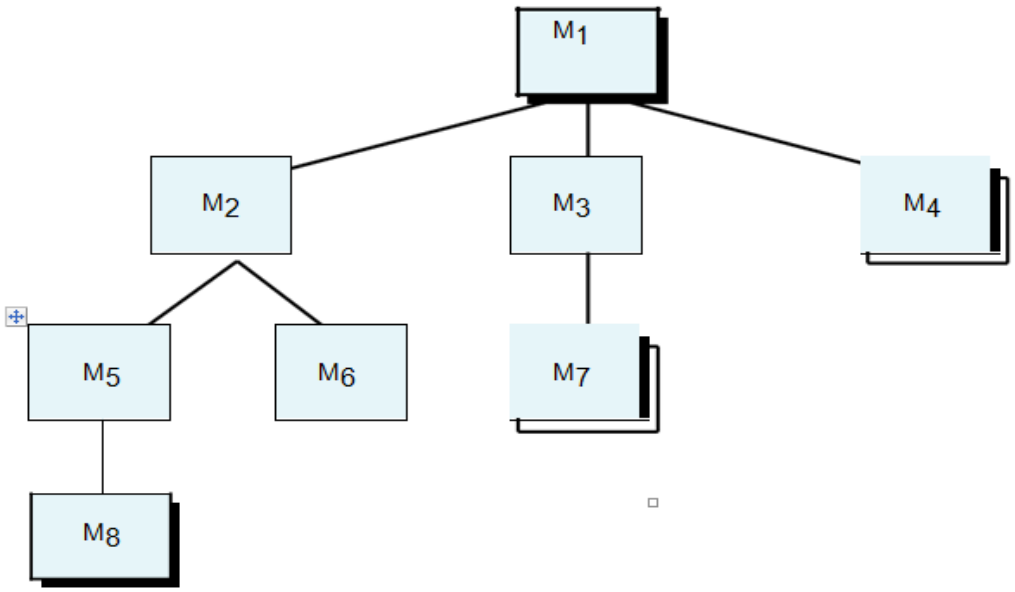


Fig: 5

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

4.2.1.3.1 Top-Down Integration.

Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

Referring to *depth-first integration* integrates all components on a major control path of the program structure. Selection of a major path is some-what arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built.

Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component
5. Regression testing (discussed later in this section) may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process.

In a "well-factored" program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first. If major control problems do exist, early recognition is essential.

If depth-first integration is selected, a complete function of the software may be implemented and demonstrated.

4.2.1.3.2 Bottom-Up Integration.

Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure).

A bottom-up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
2. A *driver* (a control program for testing) is written to coordinate test-case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward

4.2.1.4 Regression Testing.

R*egression testing* is the re execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behaviour or additional errors

The *regression test suite* (the subset of tests to be executed) contains three different classes of test cases:

1. A representative sample of tests that will exercise all software functions.
2. Additional tests that focus on software functions that are likely to be affected by the change.

3. Tests that focus on the software components that have been changed.

4.2.1.5 Smoke Testing.

Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

In essence, the smoke-testing approach encompasses the following activities:

1. Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.

2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show-stopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

4.2.2 TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span.

Although this fundamental objective remains unchanged for object-oriented soft-ware, the nature of object-oriented software changes both testing strategy and testing tactics

4.3.2.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects.

This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.

An encapsulated class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units.

Class testing for OO software is the equivalent of unit testing for conventional software. Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class testing for OO software is driven by the operations encapsulated by the class and the state behaviour of the class.

4.2.2.2 Integration Testing in the OO Context

There are two different strategies for integration testing of OO systems

1. *thread-based testing*, integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually. Regression testing is applied to ensure that no side effects occur.

2.*use-based testing*, begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) *server* classes. After the independent classes are tested, the next layer of classes, called *dependent classes*, that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed.

The use of drivers and stubs also changes when integration testing of OO systems is conducted. Drivers can be used to test operations at the lowest level and for the testing of whole groups of classes. A driver can also be used to replace the user interface so that tests of system functionality can be conducted prior to implementation of the interface. Stubs can be used in situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented.

4.3 VALIDATION TESTING

Validation succeeds when software functions in a manner that can be reasonably expected by the customer. At this point a battle-hardened software developer might protest: "Who or what is the arbiter of reasonable expectations?"

If a *Software Requirements Specification* has been developed, it describes all user-visible attributes of the software and contains a *Validation Criteria* section that forms the basis for a validation-testing approach

4.3.1Alpha and Beta Testing

- **Alpha test**

  Alpha test is conducted at the developer site by the end users

  the software is used in a natural setting with the developer site

  alpha test are conducted in a controlled environment

- **Beta test**

  The beta test is conducted at end user sites

  the beta test is a live application of the software in an environment that cannot be controlled by the developer.

4.4 SYSTEM TESTING

A classic system-testing problem is "finger pointing." This occurs when an error is uncovered, and the developers of different system elements blame each other for the problem.

1. design error-handling paths that test all information coming from other elements of the system,
2. conduct a series of tests that simulate bad data or other potential errors at the software interface,
3. record the results of tests to use as "evidence" if finger pointing does occur, and
4. Participate in planning and design of system tests to ensure that software is adequately tested.

4.4.1 RECOVERY TESTING

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed. If recovery is automatic (performed by the system itself), initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.

If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

4.4.2 SECURITY TESTING

Security testing attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration. "The system's security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack."

4.4.3 TESTING STRESS

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,

(1) special tests may be designed that generate 10 interrupts per second, when one or two is the average rate,
(2) input data rates may be increased by an order of magnitude to determine how input functions will respond,
(3) test cases that require maximum memory or other re-sources are executed,
(4) test cases that may cause thrashing in a virtual operating system are designed,
(5) test cases that may cause excessive hunting for disk-resident data are created. Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called *sensitivity testing.*. Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

4.4.4 PERFORMANCE TESTING

Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation. That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion.

External instrumentation can monitor execution intervals, log events (e.g., interrupts) as they occur, and sample machine states on a regular basis. By instrumenting a system, the tester can uncover situations that lead to degradation and possible system failure.

4.4.5 DEPLOYMENT TESTING

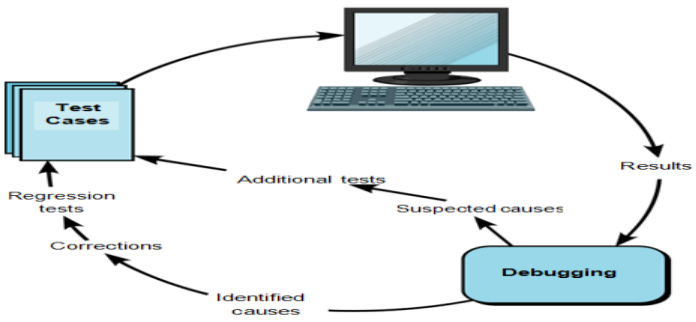
Deployment testing, sometimes called *configuration testing*, exercises the software in each environment in which it is to operate. In addition, deployment testing examines all installation procedures and specialized installation software (e.g., "installers") that will be used by customers, and all documentation that will be used to introduce the software to end users.

4.5 THE ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process, it is still very much an art. As a software engineer, you are often confronted with a "symptomatic" indication of a software problem as you evaluate the results of a test.

4.5.1 THE DEBUGGING PROCESS

Debugging is not testing but often occurs as a consequence of testing. Referring to the debugging process begins with the execution of a test case.



Results are assessed and a lack of correspondence between expected and actual performance is encountered..

The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will usually have one of two outcomes:

(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.

4.5.2 CHARACTERISTICS OF BUGS

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 at a site that is far removed. Highly coupled components exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by non errors (e.g., round-off inaccuracies).
4. The symptom may be caused by 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 re-al-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

4.6 DEBUGGING TACTICS.

The *brute force* category of debugging is probably the most common and least efficient method for isolating the cause of a software error.

Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

cause elimination is manifested by induction or deduction and introduces the concept of binary partitioning.

4.6.1 AUTOMATED DEBUGGING.

Each of these debugging approaches can be supplemented with debugging tools that can provide you with semi automated support as debugging strategies are attempted.

Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation."

A wide variety of debugging compilers, dynamic debugging aids ("tracers"), automatic test-case generators, and cross-reference mapping tools are available. However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

4.6.2 CORRECTING THE ERROR

Once a bug has been found, it must be corrected. But as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good.

1. *Is the cause of the bug reproduced in another part of the program?*
2. In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere. Explicit consideration of the logical pattern may result in the discovery of other errors.
3. *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures. If the correction is to be made in a highly coupled section of the program, special care must be taken when any change is made.
4. *What could we have done to prevent this bug in the first place?*

4.7 SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error.

Therefore, you should design and implement a computer-based system or a product with "testability" in mind.

At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort.

4.7.1 TESTABILITY.

"*Soft-ware testability* is simply how easily [a computer program] can be tested."

The following characteristics lead to testable software.

4.7.1.1 Operability.

"The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

4.7.1.2 Observability.

"What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

4.7.1.3 Controllability.

"The better we can control the software, the more the testing can be automated and optimized."

All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.

All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer.

Tests can be conveniently specified, automated, and reproduced.

4.7.1.4 Decomposability.

"By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting

4.7.1.5 Simplicity.

4.7.1.6 "The less there is to test, the more quickly we can test it." The program should exhibit *functional simplicity* (e.g., the feature set is the minimum necessary to meet requirements);

structural simplicity (e.g., architecture is modularized to limit the propagation of faults),

code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

4.7.1.6 Stability

. "The fewer the changes, the fewer the disruptions to testing." Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures.

4.7.1.7 Understand ability.

"The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers.

4.7.2 TEST CHARACTERISTICS.

attributes of a "good" test:

1. *A good test has a high probability of finding an error.* To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail.

2. *A good test is not redundant.* Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different).

3. *A good test should be "best of breed".* In a group of tests that have a similar intent, time and resource limitations may dictate the execution of only those tests that has the highest likelihood of uncovering a whole class of errors.

4. *A good test should be neither too simple nor too complex.* Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

4.8 WHITE-BOX TESTING

White-box testing, sometimes called *glass-box testing* or *structural testing*, is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

(4) exercise internal data structures to ensure their validity.

4.8.1 BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe

The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

Test cases derived to exercise the basis set are guar-anteed to execute every statement in the program at least one time during testing.

4.8.2 FLOW GRAPH NOTATION

a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be

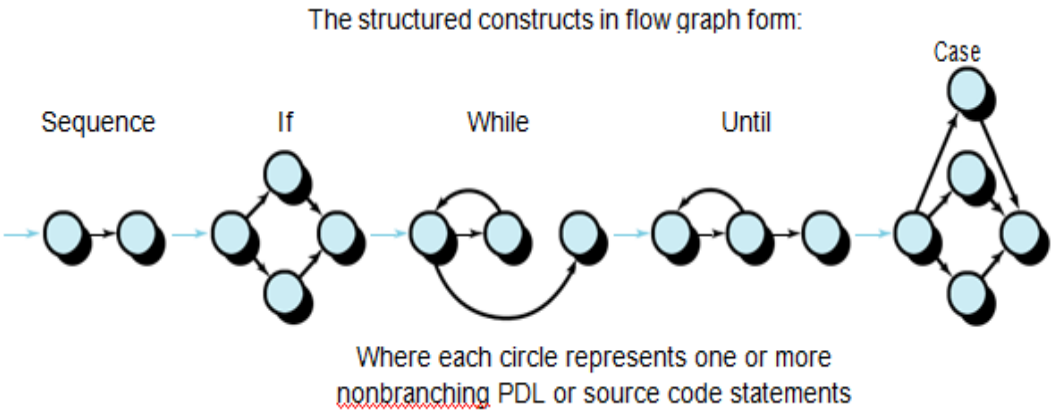


Fig:7

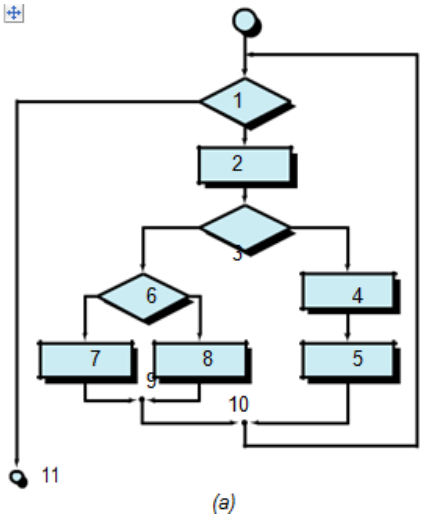


Fig:8(a)

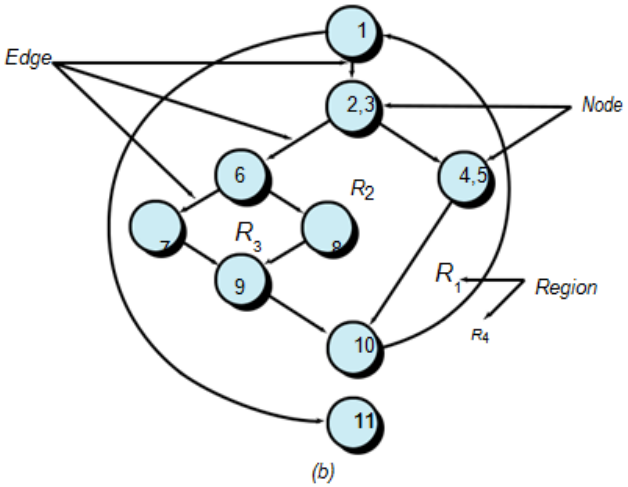


Fig: 8(b)

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be

To illustrate the use of a flow graph, consider the procedural design representation in

Here, a flowchart is used to depict program control structure. Figure maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).,

Each circle, called a *flow graph node*, represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node.

The arrows on the flow graph, called *edges* or *links*, represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent. Areas bounded by edges and nodes are called regions

4.8.2.1 INDEPENDENT PROGRAM PATHS

An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in is

Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1 through 4 constitute a *basis set* for the flow graph in Figure 23.2b. That is, if you can design tests to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will

have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways:

1.The number of regions of the flow graph corresponds to the cyclomatic complexity.
2.Cyclomatic complexity *V*(*G*) for a flow graph *G* is defined as

***V*(*G*)=*E*-N+2**

where *E* is the number of flow graph edges and

N is the number of flow graph nodes.

3.Cyclomatic complexity *V*(*G*) for a flow graph *G* is also defined as

***V*(*G*)=P+1**

where *P* is the number of predicate nodes contained in the flow graph *G*.

Referring once more to the flow graph in Figure, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.
2. *V*(*G*)= 11 edges - 9 nodes 1+2 5=4.
3. *V*(*G*)= 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Fig 8(b) is 4.

More important, the value for *V*(*G*) provides you with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

4.9 CONTROL STRUCTURE TESTING

The basis path testing technique described is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

Condition testing is a test-case design method that exercises the logical conditions contained in a program module. *Data flow testing* selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops

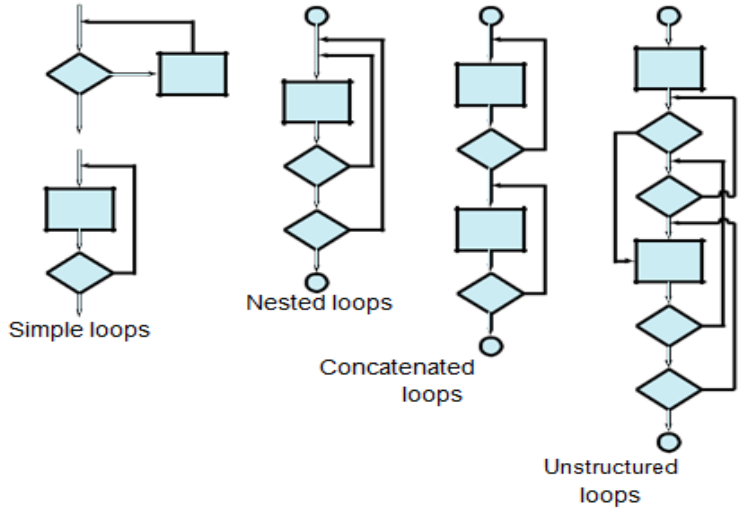


Fig:9

SIMPLE LOOPS.

The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.

1.Skip the loop entirely.

2.Only one pass through the loop.

3.Two passes through the loop.

4.m passes through the loop where *m* , *n.*

5.*n* 2 1, *n*, *n* 1 1 passes through the loop.

NESTED LOOPS.

If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests.

suggests an approach that will help to reduce the number of tests:

1.Start at the innermost loop. Set all other loops to minimum values.

2.Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.

3.Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4.Continue until all loops have been tested.

CONCATENATED LOOPS.

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not in-dependent, the approach applied to nested loops is recommended.

UNSTRUCTURED LOOPS.

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

4.9.1 BLACK-BOX TESTING

Black-box testing, also called *behavioural testing* or *functional testing*, focuses on the functional requirements of the software. That is, black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods.

Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions,

(2) interface errors,

(3) errors in data structures or external database access,

(4) behaviour or performance errors,

(5) initialization and termination errors.

Tests are designed to answer the following questions:

1.How is functional validity tested?

2.How are system behaviour and performance tested?

3.What classes of input will make good test cases?

4.Is the system particularly sensitive to certain input values?

5.How are the boundaries of a data class isolated?

6.What data rates and data volume can the system tolerate?

7.What effect will specific combinations of data have on system operation?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Testing Methods

The first step in black-box testing is to understand the objects[5] that are modelled in software and the relationships that connect these objects. Once this has been accomplished, the next step is to define a series of tests that verify "all objects

4.9.2 EQUIVALENCE PARTITIONING

Equivalence partitioning is a black-box testing method that divides the input do-main of a program into classes of data from which test cases can be derived.

Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition.

equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
4. If an input condition is Boolean, one valid and one invalid class are defined.

By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed. Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once

4.9.3 BOUNDARY VALUE ANALYSIS

A greater number of errors occurs at the boundaries of the input domain rather than in the "centre." It is for this reason that *boundary value analysis* (BVA) has been developed as a testing technique.

Boundary value analysis leads to a selection of test cases that exercise bounding values.

Boundary value analysis is a test-case design technique that complements equivalence partitioning. Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

4.9.4 ORTHOGONAL ARRAY TESTING

There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test the input domain. How-ever, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing becomes impractical or impossible.

Orthogonal array testing can be applied to problems in which the input do-main is relatively small but too large to accommodate exhaustive testing. The orthogonal array testing method is particularly useful in finding *region faults*—an error category associated with faulty logic within a software component.

Detect and isolate all single mode faults. A single mode fault is a consistent problem with any level of any single parameter. For example, if all test cases of factor P1 5 1 cause an error condition, it is a single mode failure.

Detect all double mode faults. If there exists a consistent problem when specific levels of two parameters occur together, it is called a *double mode fault.* Indeed, a double mode fault is an indication of pair wise incompatibility or harmful interactions between two test parameters.

Multimode faults. Orthogonal arrays can assure the detection of only single and double mode faults. However, many multi-mode faults are also detected by these tests.

2 MARKS

1.Define Software testing?

Software testing is a critical element of software quality assurance and represents the ultimate review specification design and coding

2.**What are the two levels of testing**?

component testing-individual components are tested .tests are derived from developer experience

system testing-the group of components are integrated to create a system or sub-system is done

3.**What is block box testing**?

the black box testing also known as behavioural testing. this method fully focus on the functional requirements of the software. tests are derived that fully exercise all functional requirements

4.**What is equivalence partitioning**?

equivalence partitioning is a block box technique that divides the input domain into classes of data from this data test cases can be derived equivalence class represents a set of valid or invalid states for input conditions

5.**What is boundary value analysis**?

a boundary value analysis is a testing technique in which the elements at the edge of domain are selected and tested it is a test case design technique that complements equivalence partitioning techniques

6.**What is cyclomatic complexity**?

Cyclomatic complexity is software metric that gives the quantitative measure of logical complexity of the program.

7.**Define debugging**.

Debugging is defined as the process of removal of defect it occurs as a consequence of successful testing

8.**What is meant by regression testing**?

Regression testing is used to check foe defects propagated to other modules by changes made to existing program. thus regression testing is used to reduce the side effects of the changes

9.**What is meant by unit testing**?

the unit testing focuses verification effort on the smallest unit of software design the software component or module

10.**What are the various testing strategies for conventional software**?

unit testing

integration testing

validation testing

system testing

5 marks

1.Explain the various types of software testing.

2.Explain in detail white box testing.

3.What is the necessity of unit testing? write all unit test considerations.

4.Explain in detail system testing and basis path testing?

5.Explain in detail about system testing and debugging.

Unit -5

PROJECT MANAGEMENT

5.1 PROJECT MANAGEMENT

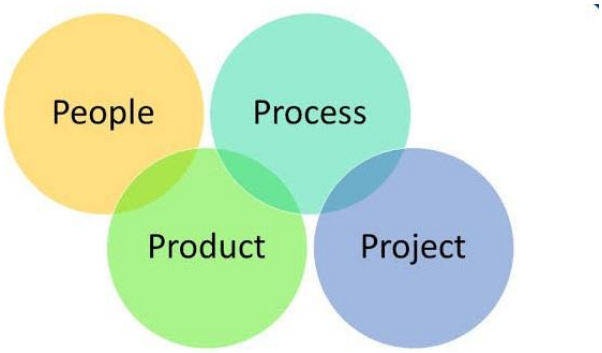
Project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management.

Management spectrum

The management spectrum describes the management of the software project. The management of a software project starts from requirement analysis and finishes based on the nature of the product, it may or may not end because almost all software products faces changes and requires support.

The management spectrum focuses on the four P's

1. People
2. Product
3. Process
4. Project



4 P's of Management Spectrum

The People

People of a project includes from manager to developer, from customer to end user. It is so important to have highly skilled and motivated developers that the Software Engineering Institute has developed a People Management Capability Maturity Model (PMCMM).

The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others.

The Product

The Product before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identified.

Without this information, it is impossible to define reasonable (and accurate) estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule that provides a meaningful indication of progress.

Once the product objectives and scope are understood, alternative solutions are considered. The alternatives enable managers and practitioners to select a "best" approach.

The Process

Software process provides the framework from which a comprehensive plan for software development can be established. A small number of framework activities are applicable to all software projects, regardless of their size or complexity.

A number of different tasks set-tasks, milestones, work products and quality assurance points enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.

Finally, umbrella activities such as software quality assurance, software configuration management, and measurement overlay the process model. Umbrella activities are independent of any one framework activity and occur throughout the process.

The Project

The project is the complete software project that includes requirement analysis, development, delivery, maintenance and updates. The project manager of a project or sub-project is responsible for managing the people, product and process.

The responsibilities or activities of a software project manager would be a long list but that has to be followed to avoid project failure.

A software project could be extremely complex and as per the industry data the failure rate is high.

5.2. THE PEOPLE

The most important ingredient that was successful on this project was having small people.

The most important thing we do for a project is selecting the staff.

The success of the software organization is much associate with an ability to recruit suitable people for suitable job.

The stakeholders

The software process is populated by stakeholders who can be categorized into one of five constituencies:

1. Senior managers- Who defines the business issues that often have a significant on the project.
2. Project (technical) managers- Who must plan, motivate, organize, and control the practitioners who do software work.
3. Practitioners- Who delivers the technical skills that are necessary to engineer a product or application.
4. Customers- Who specifies the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.
5. End users- Who interact with the software once it is released for production use.

The software team

Democratic decentralized (DD)

This software engineering team has no permanent leader. Rather, "the task co-ordinates are appointed for short duration and then replaced by others who may coordinate different tasks". Decisions on problems and approach are made by group consensus. Communication among team members is horizontal.

Controlled decentralized (CD)

This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problem solving remains a group activity, but implementation of solution is partitioned among subgroups by

the team leaders. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

Controlled Centralized (CC)

Top-level problem solving and internal team coordinators are managed by a team leader. Communication between the leader and team members is vertical.

The description of seven project factors that should be considered when planning the structure of software engineering teams:

- The difficulty of the problem to be solved.
- The size of the resultant program(s) in lines of code or function points.
- The time that the term will stay together (team lifetime).
- The degree to which the problem can be modularized.
- The required quality and reliability of the system to be built.
- The rigidity of the delivery date.
- The degree of sociability (communication) required for the project.

Team leaders

1. Motivation- The ability to encourage (by "push or pull") technical people to produce to their best ability.
2. Organization- The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
   a. Ideas or innovation- The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application. Another view of the characteristics that define an effective project manager emphasizes four key traits:

Problem solving: An effective software project manager can diagnose the technical and organizational issues that are most relevant, systematically structure a solution, apply lessons learned from past projects to new situation, and remain flexible enough to change direction if initial attempts at problems solution are fruitless.

Managerial identity: A good project manager must take charge of the project. They must have the confidence to assume control when necessary and the assurance to allow good technical people to follow their instincts.

Achievement: A competent manager must reward initiative and accomplishment to optimize the productivity of a project team. They must demonstrate through their own actions that controlled risk taking will not be punished.

Influence and team building: An effective project manager must be able to read people, understand verbal and nonverbal signals and react to the needs of the people. The manager must remain under control in high-stress situation.

Coordination and Communication Issues

The scale of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members. Uncertainty is common resulting in a continuous stream of changes that ratchets the project team. Interoperability has become a key characteristic of many system. New software must communicate with existing software and conform to predefined constraints imposed by the system or product.

A collection of project coordination techniques that are categorized in the following manner:

Formal, impersonal approaches – include software engineering documents and deliverables (including source code), technical memos, project milestones, schedules, and project control tools, change requests and related documentation, error tracking reports, and repository data.

Formal, interpersonal procedures – focus on quality assurance activities applied to software engineering work products. These include status review meetings and design and code inspections.

Informal, interpersonal procedures – include group meetings for information dissemination and problem solving and "collocation of requirements and development staff".

Electronic communication – It encompasses electronic mail, electronic bulletin boards, and by extension, video-based conferencing systems.

Interpersonal networking – It includes informal discussions with team members and those outside the project who may have experience or insight that can assist team members.

Agile Teams

Agile software development has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams.

The small, highly motivated project team, also called an agile team, adopts many of the characteristics of successful software project teams.

The agile philosophy stresses individual (team member) competency coupled with group collaboration as critical success factors for the team.

"If the people on the project are good enough, they can use almost any process and accomplish their assignment. If they are not good enough no process will repair their

inadequacy. However, lack of uses and executive support can kill a project--- "politics trump people". Inadequate support can keep even good people from accomplishing the job".

Agile terms are self-organizing; a self-organizing team does not necessarily maintain a single team structure but instead uses elements of Constantine's random, open, and synchronous paradigms.

Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools) constrained only by business requirements and organizational standards.

5.3 THE PRODUCT

A detailed analysis of software requirements would provide necessary information for estimation but analysis often takes weeks or months to complete. Worse requirements may be fluid, changing regularly as the project proceeds.

Therefore, we must examine the product and the problem it is intended to solve at the beginning of the project.

At a minimum, the scope of the product must be established and bounded.

Software Scope

The first software project management activity is the determination of software scope. Scope is defined by answering the following questions:

Context – How does the software to be built fit into the larger system, product, or business context, and what constraints are improved as a result of the context.

Information objectives – What customer visible data objects are produced as output from the software and what date objects are required for input.

Function and performance – What function does the software perform to transform input date into output. Are any special performances characteristics to be addressed?

Software project scope must be unambiguous and understandable at the management and technical levels. The statement of the scope must be bounded.

Problem Decomposition

Problem decomposition, sometimes called partitioning or problem elaboration is an activity that's sits at the core of software requirements analysis.

During the scoping activity no attempt is made to fully decompose the problem.

Decomposition is applied in major two areas:

1) The functionally and content (information) that must be delivered.
2) The process that will be used to deliver it.

A complex problem is partitioned into smaller problems that are more manageable. This is the strategy that applies as project planning begins. Software functions, described in the statement of scope, are evaluated and refined to provide more detail prior to the beginning of estimation.

Major content or data are decomposed into their constituent parts, providing a reasonable understanding of the information to be produced by the software.

As the statement of scopes evolves, the first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing:

1) Spell checking,
2) Sentence grammar checking,
3) Reference checking for large documents,
4) The implementation of a style sheet feature that imposed consistency across a document,
5) Section and chapter reference validation for large documents.

5.4 THE PROCESS

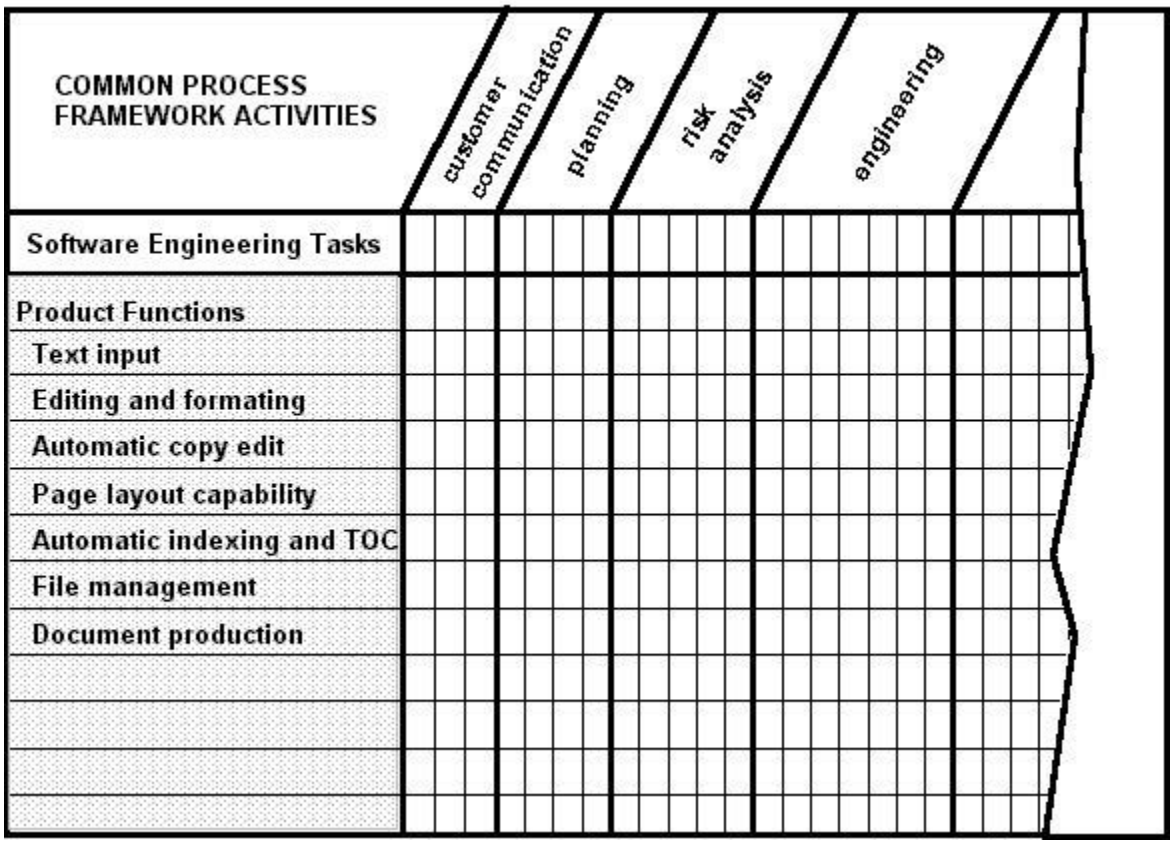
The framework activities that characterize the software process are applicable to all software projects.

The problem is to select the process model that is appropriate for the software to be engineered by project team.

The team must decide which process model is most appropriate for

1) The customers who have requested the product and the people who will do the work,
2) The characteristics of the product itself,
3) The project environment in which the software team works.

When a process model has been selected, the team then defines a preliminary project plan based on the set of process framework activities.



Melding the product and the process

Project planning begins with the melding of the product and the process. Each function to be engineered by your team must pass through the set of framework activities that have been defined for your software organization.

The generic framework activities – communication, planning, modeling, construction, construction, and deployment.

Process decomposition

A software team should have a significant degree of flexibility in choosing the software process model that is best for the project and the software engineering tasks that populate the process model once it is chosen.

Once the process model has been chosen, the process framework is adapted to it. In every case, the generic process framework discussed earlier can be used. It will work for linear models, for iterative and incremental models, for evolutionary models, and even for concurrent or component assembly models.

The process framework is invariant and serves as the basis for all work performed by a software organization.

Process decomposition commences when the project manager asks, "How do we accomplish this framework activity?"

For example, a small relatively simple project might require the following work tasks for the communication activity:

1) Develop list of clarification issues.
2) Meet with stakeholders to address clarification issues.
3) Jointly develop a statement of scope.
4) Review the statement of scope with all concerned.
5) Modify the statement of scope as required.

Consider a more complex project which has a broader scope and more significant business impact. Such a project might require the following work tasks for the communication:

1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a "working document" and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, functional, and behavioral features of the software. Alternatively, develop use casas that describe the software from the user's point of view.
7. Review each mini-spec or use case for correctness, consistency, and lack of ambiguity.
8. Assemble the mini-specs into a scoping document.
9. Review the scoping document or collection of use cases with all concerned.
10. Modify the scoping document or use cases as required.

Both projects perform the framework activity that we call communication, but the first project team performs half as many software engineering work tasks as the second.

5.5 THE PROJECT

In order to manage a successful software project, we have to understand what can go wrong so that problems can be avoided. For this we need the following criteria:

1. Software people do not understand their customer needs.

2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

How does a manager act to avoid the problems just noted?

To answer this question we need a five-part commonsense approach to software projects:

Start on the right foot. This is accomplished by working hard to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project.

To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible.

Track progress. For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity.

Make smart decisions. In essence, the decisions of the project manager and the software team should be to "keep it simple".

Conduct a postmortem analysis. Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback team members and customers.

5.6 ESTIMATION

Software project management begins with a set of activities that are collectively called project planning. Before the project can begin, the software team should estimate the work to be done, the resources that will be required, and the time will elapse from start to finish.

Planning requires us to make an initial commitment, even though it's likely that this "commitment" will be proven wrong. Whenever estimates are made, we look into the future and accept some degree of uncertainty as a matter of course.

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists.

Project complexity has a strong effect on the uncertainty inherent in planning.

Project size is another important factor that can affect the accuracy and efficacy of estimates.

The degree of structural uncertainty also has an effect on estimation risk.

5.6.1 The Project Planning Process

The objective of the software project planning is to provide a framework that enables the manager to reasonable estimates of resources, cost and schedule.

In addition, estimates should attempt to define best – case and worst – case scenarios so that project outcomes can be bounded.

The plan must be adapted and updated as the project proceeds.

Task set for project planning

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks.
4. Define required resources.
    a. Determine required human resources.
    b. Define reusable software resources.
    c. Identify Environmental resources.
5. Estimate cost and efforts.
    a. Decompose the problem.
    b. Develop two or more estimates using size, function points, process tasks or cases.
    c. Reconcile the estimates.
6. Develop a project schedule
    a. Establish a meaningful task set.
    b. Define a task network.
    c. Use scheduling tools to develop a timeline charts.

d. Define schedule tracking mechanisms.

Software scope and feasibility

Software scope describes the functions and features that are to be delivered to end users. The data that are input and output; the "content" that is presented to users as a consequence of using the software; and performance, constraints, interfaces and reliability that bound the system.

Scope is defined using one of two techniques:

1. A narrative description of software scope is developed after communication with all stakeholders.
2. The set of use cases 3 is developed by end users.

5.6.2 RESOURCES

The next planning task is estimation of resources required to accomplish the software development efforts.

The three major categories of software engineering resources-people, reusable software components, and the development environment.

Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development.

Organizational position (e.g., manager, senior software engineer) and specify (e.g., telecommunication, database, client-server) are specified.

For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations.

Reusable Software Resources

Component-based software engineering (CBSE) emphasizes reusability i.e., the creation and reuse of software building blocks. Such building blocks, often called components.

Off-the-shelf components. Existing software that can be acquired from a third party or from a past project. Components are purchased from the third party, are ready for use on the current project, and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have full experience in the application area represented by these components.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components.

New components. Software components must be built by the software team specifically for the needs of the current project.

Environmental Resources

The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software.

Hardware provides a platform that supports the tools (software) required to produce the work produces that are an outcome of good software engineering practice.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams.

5.7 SOFTWARE PROJECT ESTIMATION

Software cost and effort estimation will never be an exact science. Too many variables-human, technical, environmental, political- can affect the ultimate cost of software and effort applied to develop it.

Software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk.

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

1. Delay estimation until late in the project
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition technique to generate project and efforts estimation.
4. Use one or more empirical models for software cost and effort estimation.

The first option, is not practical. Cost estimates must be provided up-front.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other.

5.8 DECOMPOSITION TECHNIQUES

Decomposition techniques take a divide-and-conquer approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion.

The decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process.

But before an estimate can be made, we must understand the scope of the software to be built and generate an estimate of its "size".

Software Sizing

Size refers to aquantifiable outcome of the software project. If a direct approach is taken, size can be measured in lines of code(LOC). If an indirect approach is chosen, size is represented as function points(FP).

Four different approaches to the sizing problem:

Function logic sizing

Function Point sizing

Standard component sizing

` change sizing

Function logic sizing

This appoarch uses the approximate reasoning techniques that are the cornerstone of fuzzy logic

To apply this approach, the Planner must identify the type of application, establish its magnitude within the original range.

Function point sizing

The planner develops estimates of the information domain characteristics.

Standard component sizing

Software is composed of a number of different "Standard components" that are generic to particular application area.

Change sizing

This approach is used when a project encompasses the use of exiting software that must be modified in some way as part of a project. The Planner estimates the number and type of modification that must be accomplished.

Problem-Based Estimation

Problem-Based Estimation is start with a bounded scope of statement. Decompose the software into problems function that can be estimated individually.

Compute an LOC or FP values for each function. Derive cost of effort estimate by applying LOC or FP vaues to our baseline productivity metrics. Combines function estimates to produce an overall estimates for entire project

In general, the LOC/pm and FP/pm (pm=person month) metrics should be computed by project domain.

Improtant factors are team size, application area and complexity

LOC and Fp estimation differ in the level of detail required for decomposition with each value.

External inputs , external outputs, external inquires, internal logical files , external interface files.

Then the expected size value S is computed as follows:

S=(Sopt + 4Sm +Spess)/6

Historical LOC or FP is then Compared to S in order to cross check it.

Process Based Estimation

Identify the set of functions that the software needs to perform as obtained from the project scope. Identify the series of the frame work activities that need to be performed for each function.

Estimate the effort (in person month)that will be required to accomplish each software process activity for each function.

Apply average labor rates (i.e., Cost/ Unit effort) to the effort estimated for each process activity.

Compute the total cost and effort for each function and each framework activity. Compare the resulting values to those obtained by way of the LOC and Fp estimates

- If the both set of estimate agree, then your numbers are higly reliable.
- Otherwise , conduct further investigation and analysis

Estimation with Use Cases

Developing an estimation approach with use cases in problematic for the following reasons

- Use cases are described using many formats and styles there is no standard form.
- Use cases represent an external view of the software and it can be written at a many different level of abstraction
- Use cases do not address the complexity of the function and features that are described.
- Use case can describe complex behavior that involve many functions and features.

  Before use cases can be used for eastimation, the level within the structural hierarchy is established the average length (in pages) of each use case is determined , the type of software is defined an a rough architecture for the system is considered.

  To illustrate how this computation might be made , consider the following relationship:

  LOC estimate = N*LOC avg + [(Sa/Sh- 1) + (Pa/Ph - 1)] * LOC adjust
  Where

N actual number of use cases

Loc Avg is historical average LOC per use case for this type of sub system.

LOC adjust is represent an adjustment based on n percent of LOC avg

Where n defined locally and reperents tha difference between this project and "average" projects

Sa actual scenarios per use case

Sh average scenarios per use case fot his type of subsystem

Pa actual pages per usecase

Ph average pages per usecase for this type of subsystem

Reconciling Estimates

The results gathered from the various estimation techniques must be reconciled to produce a single estimate of effort, project duration and cost

If widely divergent estimates occur, investigate the following causes.

The Scope of the Project is not adequately understood of has been misinterpreted by the planner.

Productivity data used for problem-based estimation tech is in appropriate for the application, obsolete or has been misapplied.

EMPIRICAL ESTIMATION MODELS

Estimation model for computer software use emporocally derived formulas to predict efforts as a function of LOC (line of code) and FP (function point).

- Resultant values computed for LOC or Fp are entered into an estimation model
- The empirical data for these models are derived from limited sample of projects

Consequently the models should be calibrated to reflect local software development conditions.

5.9.1 The COCOMO II Model

Stands for constructive cost model. Introduced by Barry Boehm in 1981 in his book "Software Engineering Economics".

Became one of the well known and widely used estimation model in the industry. It has evolved into more comprehensive estimation model called COCOMO II. COCOMO II is actually a hierarchy of three estimation models.

COCOMO II is actually hierarchy of estimation models that address the following areas:

Appplication Composition model: used during the early stages of software engineering and basic software engineering when the following are important:

- Prototyping of user interfaces.
- Consideration of software and system interaction.
- Assesment of performance – Evaluation of technology maturity.

Early design stage model: used once requirement have been stabilized and basic software architecture has been established.

Post-architecture stage model: Used during the construction of the software.

5.9.2 COCOMO Cost Drivers

- **Personnel Factors**
    - Application Experience
    - Programming Language experience.
    - Virtual machine experience.
    - Personnel capability
    - Personnel experience.
    - Personnel experience.
    - Personnel continuity.
    - Platform experience.
    - Language and tool experience.
- **Product factors**
    - Required software reliability
    - Database size – software Product complexity.
    - Required reusability
    - Documentation match to life cycle needs
    - Product reliability and complexity
- **Platform factors**
    - Execution time constraint
    - Main strorage constraint

- Computer turn around time
- Virtual machine volatility
- Platform volatility
- Platform difficulty

- **Project Factors**
  - Use of software tools
  - Use of modern programming practices
  - Required development schedule
  - Classified security application
  - Multi – site development
  - Requirement volatility.

5.10 ESTIMATION FOR OBJECT – ORIENTED PRJECTS

The techniques that has been designed explicitly for object – oriented software with following approach.

1. Develop estimates using effort decomposition , FP Analysis and any other method that is applicable for conventional applications.
2. Using the requirement model , develop use cases and determine a count , recognize that the number of use cases may change as the project progresses.
3. From the requirements model, determine the number of key classes.
4. Categorize the type of interface for the application and develop a multiplier for support classes.

5. Multiply the total number of classe (key+ support) by the average number of work units per case.

5.11 SPECILALIZED ESTIMATION TECHNIQUES

Estimation for Agile Development

Estimation for agile development used a decomposistion approach that encompasses the following steps:

1. Each user scenario (the equivalent of a mini use case created at the very start of a project by end users or other stakeholders) is considered separately for estimation purposes.
2. The scenario is decomposed into the set of software engineering tasks that will be required to develop it.
3. i. The effort required for each task is estimated separately. Note: Estimation can be based on historical data, an empirical model, or "experience".

   ii. Alternative, the "volume" of the scenario can be estimated in LOC, FP, or some other volume-oriented measure (e.g., use-case count).
4. i. Estimates for each task are summed to create an estimate for the scenario.

   ii. Alternatively, the volume estimate for the scenario is translated into effort using historical data.
5. The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment.

5.11.1. Estimation for WebApp Projects

- WebApp projects often adopt the agile process model. A modified function point measure, coupled with the steps outlined in Section 26.9.1, can be used to develop an estimate for the WebApp. The following approach when adapting function points for WebApp estimation:
- Inputs are each input screen or form (for example, CGI of Java), each maintenance screen, and if you use a tab notebook metaphor anywhere, each tab.
- Outputs are each static Web page, each dynamic Web page script (for example, ASP, ISAPI, or other DHTML script), and each report (whether Web based or administrative in nature).
- Tables are each logical table in the database plus, if you are using XML to store data in a file, each XML object (or collection of XML attributes).

5.12 SCHEDULING

It is an important part of project planning activity. It involves deciding which tasks would be taken up when.

The majority of projects are 'completed' late, if at all. A project schedule is required to ensure that required project commitments are met.

A schedule is required to track progress toward achieving these commitments.

Why software is delivered late

- An unrealistic deadline
- Changing but unpredicted customer requirements

- Underestimation of efforts needed
- Risks not considered at the project start
- Unforeseen technical difficulties
- Unforeseen human difficulties
- Miscommunication among project staff
- Failure to recognize that project is falling behind Schedule.

5.12.1 Project Scheduling

On large projects, hundreds of small tasks must occur to accomplish a larger goal

Project manager's objectives.

Define all projects tasks.

Build an activity network that depicts their interdependencies

Identify the tasks that are critical within the activity network

Build a timeline depicting the planned and actual progress of each task-Track task progress to ensure that delay is recognized "one day at a time"

To do this, the schedule should allow progress to be monitored and the project to be controlled

Software project scheduling distributes estimated effort across the planned project duration by allocating the effort to specific tasks

Scheduling for projects can be viewed from two different perspectives

In the **first view**, an end-date for release of a computer based system has already been established and fixed.

The software organization is constrained to distribute effort within the prescribed time frame.

In the **second view**, assume that rough chronological bounds have been discussed but that the end-date is set by the software engineering organization.

Effort is distributed to make best use of resources and an end-date is defined after careful analysis of the software.

The first view is encountered far more often that the second.

Basic Principles

Compartmentalization

The project must be compartmentalized into a number of manageable activities, actions, and tasks; both the product and the process are decomposed.

Interdependency

The interdependency of each compartmentalized activity, action, or task must be determined – Some tasks must occur in sequence while others can occur in parallel. Some actions or activities cannot commence until the work product produced by another is available Time allocation.

Each task to be scheduled must be allocated some number of work units. In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies.

Start and stop dates are also established based on whether work will be conducted on a full-time or part-time basis.

Effort validation

Every project has a defined number of people on the tam. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time.

Defined responsibilities

Every task that is scheduled should be assigned to a specific team member.

Defined outcomes

Every task that is scheduled should have a defined outcome for software projects such as a work product or part of a work product. Work products are often combined in deliverables.

Defined milestones

Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

5.13 DEFINING A TASK SET FOR THE SOFTWARE PROJECT

A task set is the work breakdown structure for the project.

No single task set is appropriate for all projects and process models

It varies depending on the project type and the degree of rigor (based on influential factors) with which the team plans to work.

The task set should provide enough discipline to achieve high software quality.

But it must not burden the project team with unnecessary work.

Types of Software Projects

Concept development projects – Explore some new business concept or application of some new teachnology.

New application development – Undertaken as a consequence of a specific customer request.

Application enhancement – Occur when existing software undergoes major modifications to function, performance, or interfaces that are observable by the end user.

Application maintenance – Correct, adapt, or extend existing software in ways that may not be immediately obvious to the end user.

Reengineering projects – Undertaken with the intent of rebuilding an existing (legacy system in whole or in part.

Purpose of a Task Network

It is also called an activity network. It is a graphic representation of the task flow for a project. It depicts task length, sequence, concurrency, and dependency.

Points out inter-task dependencies to help the manager ensure continuous progress toward project completion.

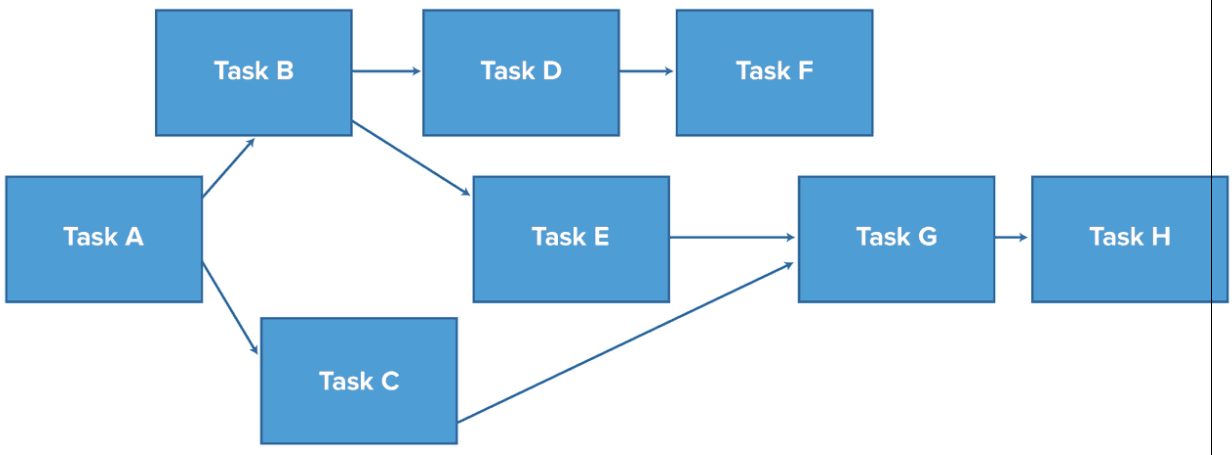
The critical path

A single path leading for start to finish in a task network.

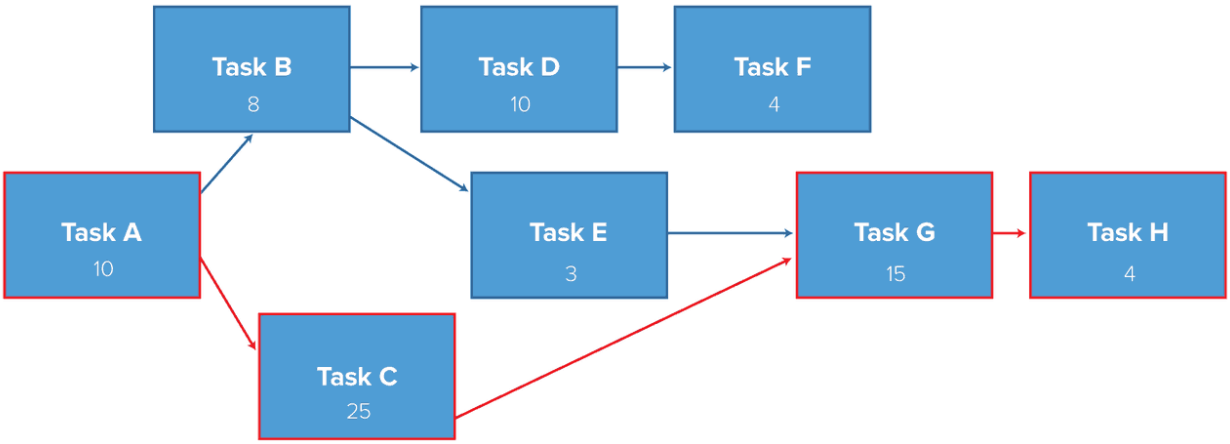
It contains the sequence of tasks that must be completed on schedule if the project as a whole is to be completed on schedule.

It also determines the minimum duration of the project.

Example of Task Network



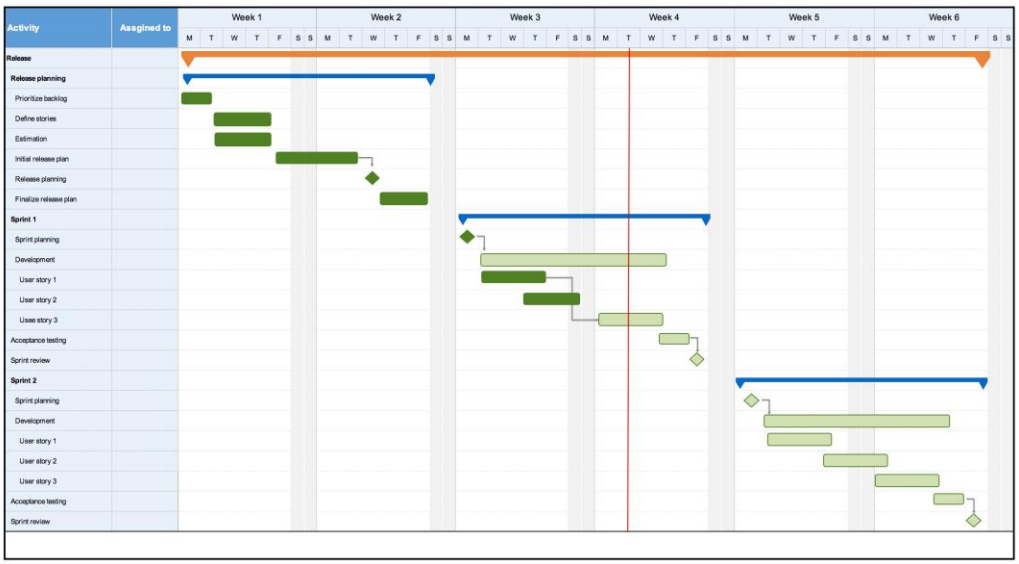
Example of task network with critical path



Critical Path: A-C-G-H

Timeline chart

It is also called a Gantt chart; All project tasks are listed in the far left column.



The next few columns may list the following for each task: projected start date, projected stop date, projected duration, actual start date, actual stop date, actual duration, task interdependencies (i.e., predecessors).

To the far right are columns representing dates on a calendar. The length of a horizontal bar on the calendar indicates the duration of the task. When multiple bars occur at the same time interval on the calendar, this implies task concurrency.

A diamond in the calendar area of a specific task indicates that the task is a milestone; a milestone has a time duration of zero.

Methods for tracking the schedule

Qualitative approaches

Conduct periodic project status meetings in which each team member reports progress and problems – Evaluate the results of all reviews conducted throughout the software engineering process.

Determine whether formal project milestones (i.e., diamonds) have been accomplished by the scheduled date.

Compare actual start date to planned start date for each project task listed in the timeline chart.

Meet informally with the software engineering team to obtain their subjective assessment of progress to date and problems on the horizon

Quantitative approach

Use earned value analysis to assess progress quantitatively

Software Quality

Software quality is an effective software process applied in a way which creates a useful product and the product provides measurable value for those who produce and use it.

5.14 SOFTWARE QUALITY FACTORS

Following are the software quality factors:

1. McCall's Quality factors
2. ISO 9126 Quality factors

5.14.1 McCall's Quality Factors

McCall's software quality factors focus on following aspect of a software product.

1. **Operational characteristic of software product**

Correctness – A program satisfies specification and fulfills the customer requirements.

Reliability – A program is expected to perform the intended function with needed precision.

Efficiency – Amount of computing resources and the code required by a program to perform its functions.

Integrity – The efforts are taken to handle access authorization

Usability – The efforts are needed to learn, operate, prepare input and interpret the output of a program.

2. **Ability to undergo change of the product transition**

   Portability – Transfer the program from one system (hardware or software) environment to another.

   Reusability – Extent to that a program or a part of a program is reused in other applications.

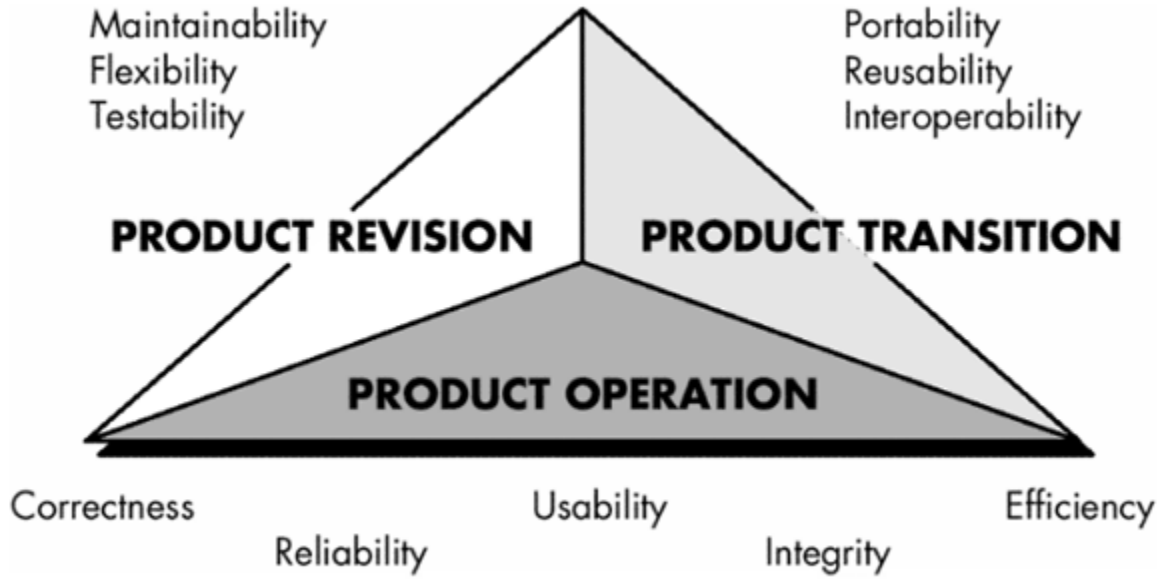
   Interoperability – The efforts are needed to couple on system to another.

3. **Adaptability to new environment or product revision**

   Maintainability – The efforts are needed to locate and fix an error in the program.

   Flexibility – The efforts are needed to modify an operational program.

   Testability – The effort needed to test a program to check that it performs its intended function.



5.14.2 ISO 9126 Quality factors

It is developed in an attempt to recognize the quality attributes.

The standard identifies the following quality attributes:

1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

Software Reliability

The probability of failure free program in a specified environment for a specified time is known as software reliability.

The software application does not produce the desired output according to the requirements then the result in failure.

Measures of software reliability and availability

A measure of reliability is Mean Time Between Failure (MTBF).

$$MTBF=MTTF + MTTR$$

Where, the MTTF and MTTR are Mean Time To Failure and Mean Time To Repair.

An alternate measure of reliability is Failures In Time (FIT).

A software availability is the probability that a program is running according to the requirements at a given point in time.

Availability = [MTTF/(MTTF + MTTR)] * 100%

The availability measure is an indirect measure of the maintainability of the software and it is more sensitive to MTTR.

Distributed Software Engineering

In distributed system, various computers are connected in a network.

The connected computers communicate with each other by passing the message in a network.

Distributed system issues

Distributed system is more complex system than the system running on a single processor.

Complexity occurs because various part of the system are managed separately as in the network.

Following are design issues considered while designing distributed systems:

Resource sharing – Sharing hardware and software resources.

The openness – System is designed in way that equipment and software from different vendors are used.

Concurrency – Different users are concurrently accessing the resources from different locations.

Scalability – The distributed operating system should be scalable to accommodate the increase service load.

Fault tolerance – The system should continue to function properly after the fault has occurred.

Software Quality Assurance

SQA should be utilized to the full extend with traceability of errors, cost efficient. There are the principles behind a highly efficient SQA procedure. Every business should use SQA and use it to the best as they can.

Businesses that develop and application and setting it for global standard will have a high reputation and the salability is even better. SQA definitely has benefits that will launch the company towards the global standard it needs.

5.15 ELEMENTS OF SOFTWARE QUALITY ASSURANCE

Three categories of software are acquired from external software vendors.

Security management

With the increase in cyber-crime and new government regulations regarding privacy, every software organization should institute policies that protect data at all levels, establish firewall protection for WebApps, and ensure that software has not been tampered with internally.

Safety

Because software is almost an essential component of human rated systems (e.g., automotive or aircraft applications), the impact of hidden defects can be catastrophic.

Risk management

Although the analysis and mitigation of risk is the concern of software engineers.

5.15.1 Attributes, and Metrics

Requirements quality

The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow. SQA must ensure that the software team has properly reviewed the requirements model to achieve a high level of quality.

Design quality

Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that it exhibits high quality and that the design itself conforms to requirements. SQA looks for attributes of the design that are indicators of quality.

Code quality

Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate

maintainability. SQA should isolate those attributes that allow a reasonable analysis of the quality of code.

Quality control effectiveness

A software team should apply limited resources in a way that has the highest likelihood of achieving a high-quality result. SQA analyzes the allocation of resources for reviews and testing to assess whether they are being allocated in the most effective manner.

5.16 STATISTICAL SOFTWARE QUALITY ASSURANCE

Statistical quality assurance reflects a growing trend throughout industry to become more quantitative about quality. For software, statistical quality assurance implies the following steps:

1. Information about software errors and defects his collected and categorized.
2. An attempt is made to trace each error and defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer).
3. Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the vital few).
4. Once the vital few causes have been identified, move to correct the problems that have caused the errors and defects.

5.16.1 Six Sigma for Software Engineering

Six Sigma is the most widely used strategy for statistical quality assurance in industry today. The Six Sigma strategy is a rigorous and disciplined methodology that uses data and

statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects in manufacturing and service-related processes

The Six Sigma methodology defines three core steps:

- Define customer requirements and deliverables and project goals via well-defined methods of customer communication.
- Measure the existing process and its output to determine current quality performance (collect defect metrics).
- Analyze defect metrics and determine the vital few causes.

If and existing software process is in place, but improvement is required, Six Sigma suggest two additional steps:

- Improve the process by eliminating the root causes of defects.
- Control the process to ensure that future work does not reintroduce the causes of defects.

These core and additional steps are sometimes referred to as the DMAIC (define, measure, analyze, improve, and control) method.

If an organization is developing a software process (rather than improving and existing process), the core steps are augmented as follows:

- Design the process to (1) avoid the root causes of defects and (2) to meet customer requirements.
- Verify that the process model will, in fact, avoid defects and meet customer requirements.

5.16.2 The ISO 9001:2000 Quality Standards

Establish the elements of a quality management system.

- Develop, implement, and improve the system.
- Define a policy that emphasizes the importance of the system.

Document the quality system

- Describe the process.
- Produce an operational manual.
- Develop methods for controlling (updating) documents.
- Establish methods for record keeping.

Support quality control and assurance.

- Promote the importance of quality among all stakeholders.
- Focus on customer satisfaction

Define a quality plan that addresses objectives, responsibilities, and authority.

Define communication mechanisms for the quality management system.

Establish review mechanisms for the quality management system.

- Identify review methods and feedback mechanisms.

Define follow-up procedures.

- Identify quality resources including personnel, training, and infrastructure elements.

- Establish control mechanisms.
- For planning
- For customer requirements
- For technical activities (e.g., analysis, design, testing)
- For project monitoring and management

Define methods for remediation.

- Assess quality data and metrics.

Define approach for continuous process and quality improvement.

5.17 FORMAL TECHNICAL REVIEW

A FTRis a software quality control activity performed by software engineers and others. The objectives are:

1. To uncover errors in function, logic or implementation for any representation of the software.
2. To verify that the software under review meets its requirements.
3. To ensure that the software has been represented according to predefined standards.
4. To achieve software that is developed in a uniform manner and
5. To make projects more manageable.

Review meeting in FTR

The Review meeting in a FTR should abide to the following constraints

1. Review meeting members should be between three and five.

2. Every person should prepare for the meeting and should not require more than two hours of work for each person.
3. The duration of the review meeting should be less than two hours.

Formal Technical Review

- The focus of FTR is on a work product that is requirement specification, a detailed component design, a source code listing for a component.
- The individual who has developed the work product i.e, the producer informs the project leader that the work product is complete and that a review is required.
- The project leader contacts a review leader, who evaluates the product for readiness, generate copy of product material and distributes them to two or three review members for advance preparation.
- Each reviewer is expected to spend between one and two hours reviewing the product, making notes.
- The review leader also reviews the product and establish and agenda for the review meeting.
- The review meeting is attended by review leader, all reviewers and the producer.
- One of the reviewer act as a recorder, who notes down all important points discussed in the meeting.
- The meeting (FTR) is started by introducing the agenda of meeting and then the producer introduces his product. Then the producer "walkthrough"

the product, the reviewers raise issues which they have prepared in advance.

- If errors are found the recorder note down.

Review reporting and record keeping

- During the FTR, a reviewer (recorder) records all issues that have been raised.
- A review summary report answers three questions
  1. What was review?
  2. Who reviewed it?
  3. What were the findings and conclusions?
- Review summary report is a single page form with possible attachments.
- The review issues list serves two purposes.
  1. To identify problem areas in the product
  2. To serve as an action item checklist that guides the producer as corrections are made

Review Guidelines

- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and rebuttal.
- Enunciate problem areas, but don't attempt to solve every problem noted.
- Take return notes.
- Limit the number of participants and insist upon advance preparation.
- Develop a checklist for each product i.e., likely to be reviewed.

- Allocate resources and schedule time for FTRS.
- Conduct meaningful training for all reviewer.
- Review your early reviews.

PART – A (2 Marks)

1. **Define Project management.**

   Project management comprises of a number of activities, which contains planning of project, deciding scope of software product, estimation of cost in various terms, scheduling of tasks and events, and resource management.

2. **Define Democratic decentralized (DD).**

   This software engineering team has no permanent leader. Rather, "task coordinators are appointed for short durations and then replaced by others who may coordinate different tasks". Decisions on problems and approach are made y group consensus. Communication among team members is horizontal.

3. **Define Controlled decentralized (CD).**

   This software engineering team has a defined leader who coordinates specific tasks and secondary leaders that have responsibility for subtasks. Problems solving remains a group activity, but implementation of solutions is partitioned among subgroups by the team leader. Communication among subgroups and individuals is horizontal. Vertical communication along the control hierarchy also occurs.

4. **Define Agile.**

   Agile software development has been suggested as an antidote to many of the problems that have plagued software project work. To review, the agile philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams.

5. **What is Software scope?**

   Software scope describes the functions and features that are to be delivered to end user; the data that are input and output; the "content" that is presented to users as a

consequence of using the software; and the performance, constraints, interfaces, and reliability that bound the system.

6. **Define Software quality.**

   Software quality is an effective software process applied in a way which creates a useful product and the product provides measurable value for those who produce and use it.

7. **Define software reliability.**

   The probability of failure free program in a specified environment for a specified time is known as software reliability.

8. **Define Distributed Software Engineering.**

   In distributed system, various computers are connected in a network.

   The connected computers communicate with each other by passing the message in a network.

9. **Define Software Quality Assurance.**

   SQA should be utilized to the full extend with traceability of errors, cost efficient. There are the principles behind a highly efficient SQA procedure. Every business should use SQA and use it to the best as they can.

10. **Define Six Sigma.**

    The Six Sigma strategy is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating defects in manufacturing and service-related processes.