S.Y.B.Sc. (I.T)

SEMESTER - IV

PAPER - I

SOFTWARE ENGINEERING

Dr. Suhas Pednekar
Vice Chancellor
University of Mumbai, Mumbai

Prof. Ravindra D. Kulkarni
Pro Vice-Chancellor,
University of Mumbai

Prof. Prakash Mahanwar
Director,
IDOL, University of Mumbai

Programme Co-ordinator : **Shri Mandar Bhanushe**
Head, Faculty of Science and Technology IDOL,
Univeristy of Mumbai – 400098.

Course Co-ordinator : **Gouri S.Sawant**
Assistant Professor B.Sc.I.T, IDOL,
University of Mumbai- 400098.

Course Writers : **Ms Aarti Sahitya**
Assistant Professor,
K. J. Somaiya Institute of Engineering &
Information Technology, Sion , Mumbai 400022.

: **Ms. Mithila Chavan**
Assistant Professor,
Vidyalankar School of Information technology
Wadala (E), Mumbai 400031.

: **Mr Rajendra Patole**
Assistant Teacher,
Vidyalankar School of Information technology
Wadala (E), Mumbai 400031.

: **Ms Jyotsna Anthal**
Assistant Professor,
Thakur College of Science and Commerce,
Kandivali (E), Mumbai 400101.

: **Ms Prachi Surve**
Assistant Professor of Mathematics
Ramniranjan Jhunjhunwala College,
Ghatkopar West, Mumbai 400086

April 2021, Print - 1

Published by : **Director,**
Institute of Distance and Open Learning,
University of Mumbai,
Vidyanagari,Mumbai - 400 098.

DTP Composed by : **7SKILLS**
Dombivli West, Thane - 421202

Printed by :

CONTENTS

S.Y.B.SC. (I.T)

SEMESTER - 4

Software Engineering

SYLLABUS

Unit-I	**An Introduction** : To Software, Software Engineering, Software Process, Software Engineering Methods; CASE Tools, Attributes of good software. **Socio-technical system :** Essential characteristics of socio technical systems, Emergent System Properties, Systems Engineering, Components of system such as organization, people and computers, Dealing Legacy Systems. **Critical system** : Types of critical system, A simple safety critical system, Dependability of a system, Availability and Reliability, Safety and Security of Software systems
Unit-II	**Software processes :** Fundamental activities of software process, Different software process models, Process Iteration and Activities, The Rational Unified Process, CASE in detail. **Project Management** : Software Project Management, Management activities, Project Planning, Project Scheduling, Risk Management. **Software Requirements:** Functional and Non-functional requirements, User Requirements, System Requirements, Interface Specification, Documentation of the software requirements
Unit-III	**Requirements Engineering Processes** : Feasibility study, Requirements elicitation and anlaysis, Requirements Validations, Requirements Management. **System Models** : Models and its types, Context Models, Behavioural Models, Data Models, Object Models, Structured Methods. **Architectural Design** : Architectural Design Decisions, System Organisation, Modular Decomposition Styles, Control Styles, Reference Architectures
Unit-IV	**Application Architectures** : Data Processing Systems, Transaction Processing Systems, Event Processing Systems, Language Processing Systems **Object Oriented Design** : Objects and Object Classes, An object Oriented Design

	Process, Design Evolution **User Interface Design** : Need of UI design, Design issues, The UI design Process, User analysis, User Interface Prototyping, Interface Evaluation **Rapid Software Development :** Agile Methods, Extreme Programming, Rapid Application Development, Software Prototyping
Unit-V	**Component based Software Engineering :** Components and Component models, The CBSE Process, Component Composition. **Verification and Validation :** Planning Verification and Validation, Software Inspections, Automated Static Analysis, Verification and Formal Methods. **Software Testing :** System Testing, Component Testing, Test Case Design, Test Automation. **Software Cost Estimation :** Software Productivity, Estimation Techniques, Algorithmic Cost Modeling, Project Duration and Staffing
Unit-VI	**Quality Management :** Process and Product Quality, Quality assurance and Standards, Quality Planning, Quality Control, Software Measurement and Metrics **Process Improvement :** Process and product quality, Process Classification, Process Measurement, Process Analysis and Modeling, Process Change, The CMMI Process Improvement Framework. **Security Engineering :** Security Concepts, Security Risk Management, Design for Security, System Survivability. **Service Oriented Software Engineering** : Services as reusable components, Service Engineering, Software Development with Services

Books:

Software Engineering, "Ian Somerville", 8th edition, Pearson Education.

Software Engineering, Pankaj Jalote, Narosa Publication

Reference:

Software Design, "D.Budgen", 2nd edition, Pearson education.

Software engineering, A practitioner's approach, Roger Pressman, Tata McGraw-Hill

Software Engineering by KL James, PHI(2009) EEE edition

Software Engineering principles and practice by WS Jawadekar Tata McGraw-Hill

UNIT 1

1

AN INTRODUCTION

Unit Structure:

1.0 Objective

At the end of this unit, the student will be able to

- Understand the software process
- Define the software and software engineering
- Illustrate the concept of Case tools
- Signify what are Soft Engineering Methods
- Explain the attributes of Good Software

1.1 Introduction to Software and Software Engineering

1. The term software engineering is composed of two words, software engineering

2. Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

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

4. So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

4. IEEE defines software engineering as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

5. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

6. Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes.

7. Software engineering helps to reduce this programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

8. The principle of abstraction implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose.

9 Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem.

10 The other approach to tackle problem complexity is DEPT OF CSE & IT VSSUT, Burla decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help.

11 The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution.

12 A good decomposition of a problem should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

13. The need of software engineering arises because of higher rate of change in user requirements and environments on which the software is working

13.1 Large Software- It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

13.2 Scalability- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

13.3 Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

13.4 Dynamic Nature-The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

13.5 Quality Management- Better process of software development provides better and quality software product.

1.2 Software Process

1. It is the structured set of activities required to develop a software system with specification, design, validation and evolution.

2. A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective

3. Basic steps of software process are as follow

3.1 Systems Engineering- Software as part of larger system, determine requirements for all system elements, allocate requirements to software

3.2 Software Requirement Analysis- Develop understanding of problem domain, user needs, function, performance, interfaces, software design, multi-step process to determine architecture, interfaces, data structures, functional detail produces high-level form that can be checked for quality conformance before coding.

3.3 Coding- Produce machine readable and executable form, match HW, OS and design needs.

3.4 Testing- Confirm that components, subsystems and complete products meet requirements specifications and quality, find and fix defects.

3.5 Maintenance- Incrementally evolve software to fix defects, add features, adapt to new condition. Often 80% of effort spent here. Figure below show general steps of Software Process

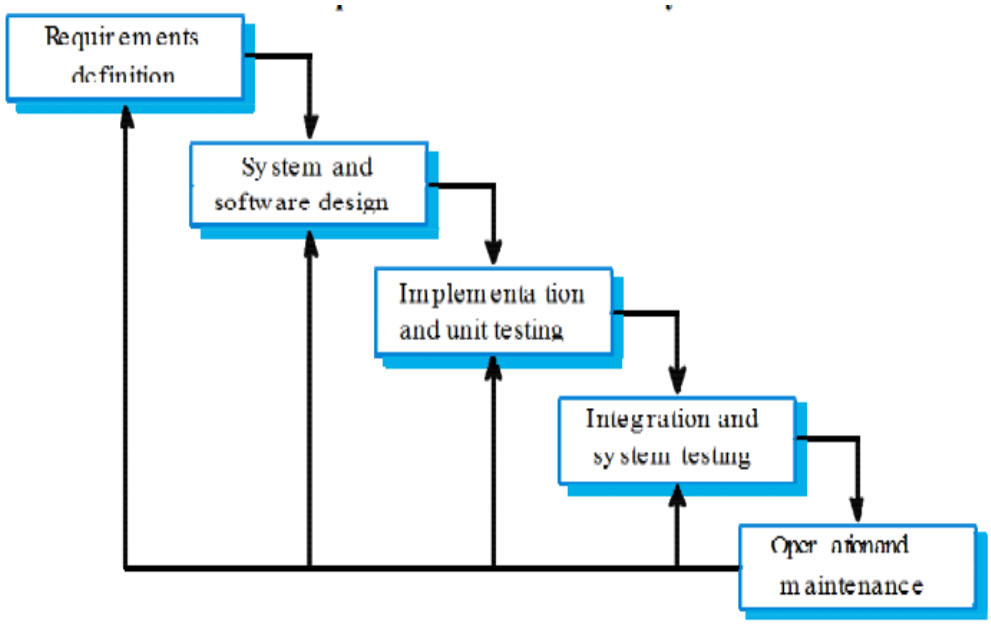


Fig 1 Steps of Software Process

4 A software process is the set of activities and associated outcome that produce a software product. Software engineers mostly carry out these activities. There are four key process activities, which are common to all software processes. These activities are as follows

1. Software specifications: The functionality of the software and constraints on its operation must be defined.
2. Software development: The software to meet the requirement must be produced.
3. Software validation: The software must be validated to ensure that it does what the customer wants.
4. Software evolution: The software must evolve to meet changing client needs.

5 Software Process model- A software process model is a specified definition of a software process, which is presented from a particular perspective. Models, by their nature, are a simplification, so a software process model is an abstraction of the actual process, which is being described. Process models may contain activities, which are part of the software process, software product, and the roles of people involved in software engineering. Some examples of the types of software process models that may be produced are:

5.1 A workflow model**:** This shows the series of activities in the process along with their inputs, outputs and dependencies. The activities in this model perform human actions.

5.2 A dataflow or activity model**:** This represents the process as a set of activities, each of which carries out some data transformations. It shows how the input to the process, such as a specification is converted to an output such as a design. The activities here may be at a lower level than activities in a workflow model. They may perform transformations carried out by people or by computers.

5.3 A role/action model**:** This means the roles of the people involved in the software process and the activities for which they are responsible.

6. There are several various general models or paradigms of software development:

6.1 The waterfall approach**:** This takes the above activities and produces them as separate process phases such as requirements specification, software design, implementation, testing, and so on. After each stage is defined, it is "signed off" and development goes onto the following stage.

6.2 Evolutionary development**:** This method interleaves the activities of specification, development, and validation. An initial system is rapidly developed from a very abstract specification.

6.3 Formal transformation**:** This method is based on producing a formal mathematical system specification and transforming this specification, using mathematical methods to a program. These transformations are 'correctness preserving.' This means that you can be sure that the developed programs meet its specification.

6.4 System assembly from reusable components**:** This method assumes the parts of the system already exist. The system development process target on integrating these parts rather than developing them from scratch.

7. Example of Software Process- Suppose a Hotel wants to build a automatic bill generation report for every customer visited to hotel and stores the previous history of user who were already visited to Hotel, then according to software process steps, a Hotel follows

7.1 The system should check the amount is correct or as per the items taken by customer, without giving error , the system should generate the proper bill of particular customer so that is part of first step as software specification.

7.2 By taking the feedback from every customer visited to the hotel in order to check the system as per requirements or not so this is the second step of Software process as software development.

7.3 The system should be consistent upon giving any test cases that is third step of software process that is software validation.

7.4 By taking more requirements from customer, the hotel owner always evolves the system with new modifications that is fourth step of software process that is software evolution.

1.3 Software Engineering Methods

The Software engineering methods divided in to following methods as

1.1 Computer Based System

1 A set or arrangement of elements that are organized to accomplish some predefined goal by processing information.

2. The goal may be to support some business function or to develop a product that can be sold to generate business revenue. To accomplish the goal, a computer-based system makes use of a variety of system element.

    2.1 Software. Computer programs, data structures, and related documentation that serve to effect the logical method, procedure, or control that is required

    2.2 Hardware. Electronic devices that provide computing capability, the interconnectivity devices (e.g., network switches, telecommunications devices) that enable the flow of data, and electromechanical devices (e.g., sensors, motors, pumps) that provide external world function

    2.3 People. Users and operators of hardware and software.

    2.4 Database. A large, organized collection of information that is accessed via software.

    2.5 Documentation. Descriptive information (e.g., hardcopy manuals, on-line help files, Web sites) that portrays the use and/or operation of the system.

    2.6 Procedures. The steps that define the specific use of each system element or the procedural context in which the system resides.

3 The elements combine in a variety of ways to transform information. For example, a marketing department transforms raw sales data into a profile of the typical purchaser of a product; a robot transforms a command file containing specific instructions into a set of control signals that cause some specific physical action

4 One complicating characteristic of computer-based systems is that the elements constituting one system may also represent one macro element of a still larger system. The macro element is a computer-based system that is one part of a larger computer-based system. As an example, we consider a "factory automation system" that is essentially a hierarchy of systems.

5 At the lowest level of the hierarchy we have a numerical control machine, robots, and data entry devices. Each is a computerbased system in its own right. The elements of the numerical control machine include electronic and electromechanical hardware (e.g., processor and memory, motors, sensors),

software (for communications, machine control, interpolation), people (the machine operator), a database (the stored NC program), documentation, and procedures. A similar decomposition could be applied to the robot and data entry device. Each is a computer-based system.

6 At the next level in the hierarchy, a manufacturing cell is defined. The manufacturing cell is a computer-based system that may have elements of its own (e.g., computers, mechanical fixtures) and also integrates the macro elements that we have called numerical control machine, robot, and data entry device.

1.2 The System Engineering Hierarchy

1 System engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy illustrated in figure shown below

2 The system engineering process usually begins with a "world view." That is, the entire business or product domain is examined to ensure that the proper business or technology context can be established.

3 The world view is refined to focus more fully on specific domain of interest. Within a specific domain, the need for targeted system elements (e.g., data, software, hardware, people) is analyzed.

4 At the top of the hierarchy, a very broad context is established and, at the bottom, detailed technical activities, performed by the relevant engineering discipline (e.g., hardware or software engineering), are conducted.

5 System Modelling-

5.1 System engineering is a modelling process. Whether the focus is on the world view or the detailed view, the engineer creates models that

1 Define the processes that serve the needs of the view under consideration.

2 Represent the behavior that serve the needs of the view under consideration.

3 Explicitly define both exogeneous and endogenous input to the model.

4 Represent all linkages that will enable the engineer to better understand the view.

5 To construct a system model, the engineer should consider a number of restraining factors

5.1 Assumptions that reduce the number of possible permutations and variations thus enabling a model to reflect the problem in a reasonable manner.

5.2 Simplifications- that enable the model to be created in a timely manner.

5.3 Limitations- that helps to bound the system.

5.4 Constraints-that will guide the manner in which the model is created and the approach taken when the model is implemented.

5.5 Preferences- that indicate the preferred architecture for all data, functions and technology.

5.2 System Simulation

1. Many computer-based systems interact with the real world in a reactive fashion. That is, real-world events are monitored by the hardware and software that form the computer-based system, and based on these events, the system imposes control on the machines, processes, and even people who cause the events to occur.

2. If the system crashed due to incorrect function, inappropriate behavior, or poor performance, we picked up the pieces and started over again.

3. Many systems in the reactive category control machines and/or processes (e.g., commercial aircraft or petroleum refineries) that must operate with an extremely high degree of reliability.

4. Today, software tools for system modelling and simulation are being used to help to eliminate surprises when reactive, computer-based systems are built. These tools are applied during the system engineering process, while the role of hardware and software, databases and people is being specified. Modelling and simulation tools enable a system engineer to "test drive" a specification of the system.

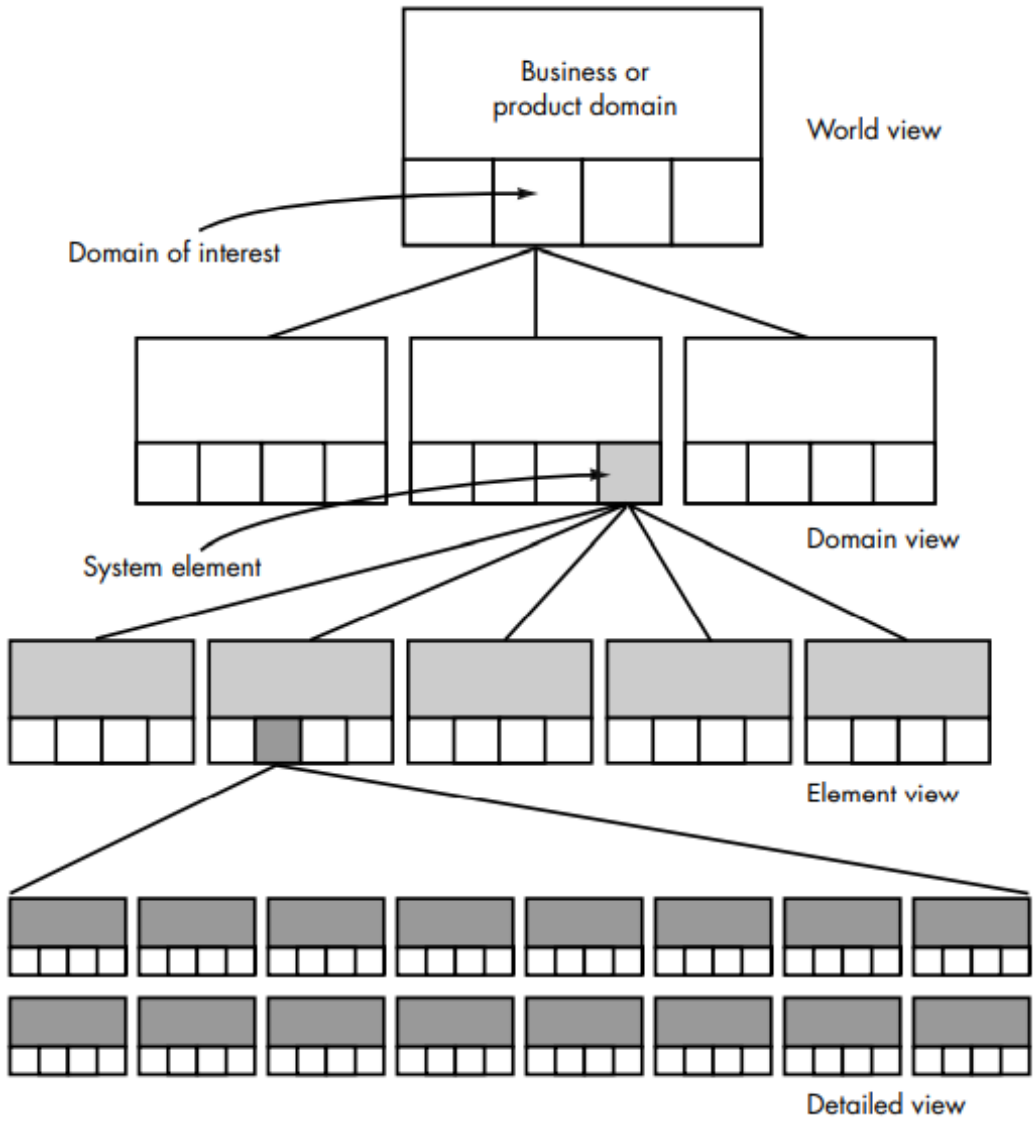


Fig 2 The System Engineering Hierarchy

1.3 Business Process Engineering: An Overview

1 The goal of business process engineering is to define architectures that will enable a business to use information effectively.

2 Today, each IT organization must become, in effect, its own systems integrator and architect. It must design, implement, and support its own unique configuration of unique computing resources, distributed logically and geographically throughout the enterprise and connected by an appropriate enterprisewide networking scheme.

3 Moreover, this configuration can be expected to change continuously, but unevenly, across the enterprise, due to changes in business requirements and in computing technology.

4. Three different architectures must be analyzed and designed within the context of business objectives and goals

   1 Data Architecture- Provides a framework for the information needs of a business or a business function.

   2 Application Architecture- encompasses those elements of a system that trans-form objects within the data architecture for some business purpose.

   3 Technology- It provides the foundation for the data and application architectures.

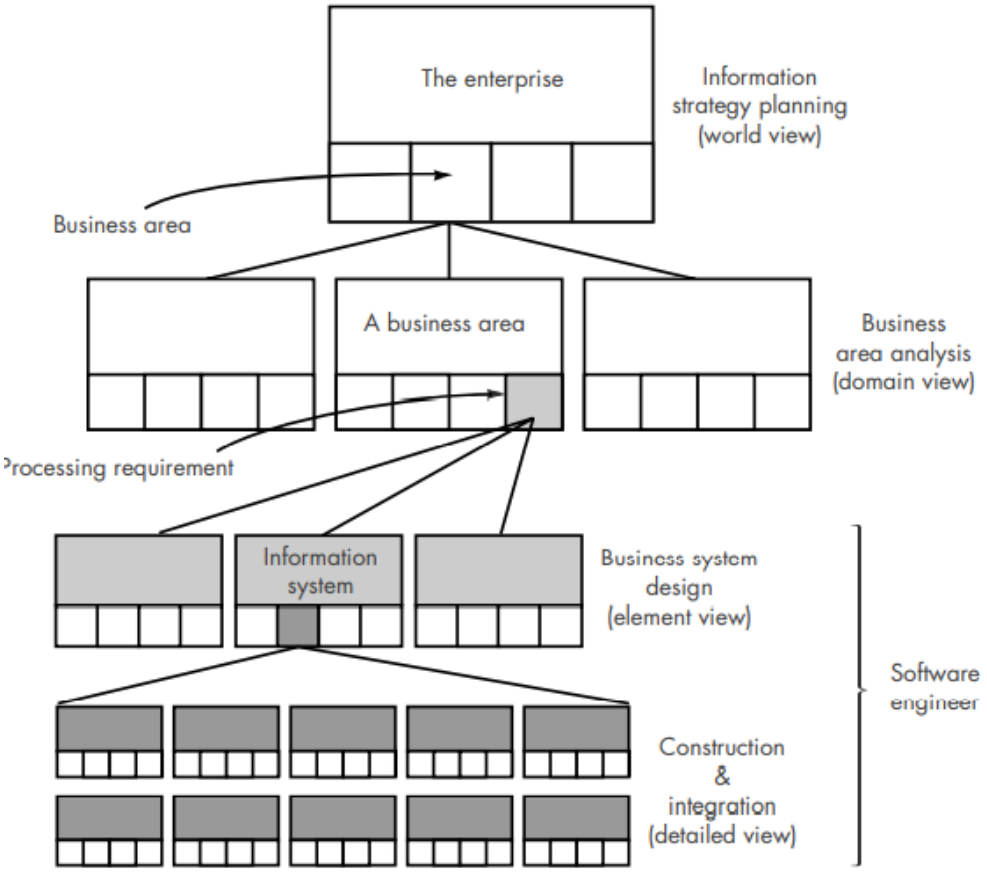


Fig 3 The Business process engineering hierarchy

1.4 Product Engineering: An Overview

1. The goal of product engineering is to translate the customer's desire for a set of defined capabilities in to a working product. To achieve this goal, product engineering like business process engineering must derive architecture and infrastructure.

2. Once the requirements are known, the job of requirements engineering is to allocate function and behavior to each of the four component engineering commences.

3. Once allocation has occurred, system component engineering commences. System component engineering is actually a set of concurrent activities that address each of the system components separately: software engineering, hardware engineering, human engineering, and database engineering

4. The analysis step models allocated requirements into representations of data, function, and behavior. Design maps the analysis model into data, architectural, interface, and software component-level designs.

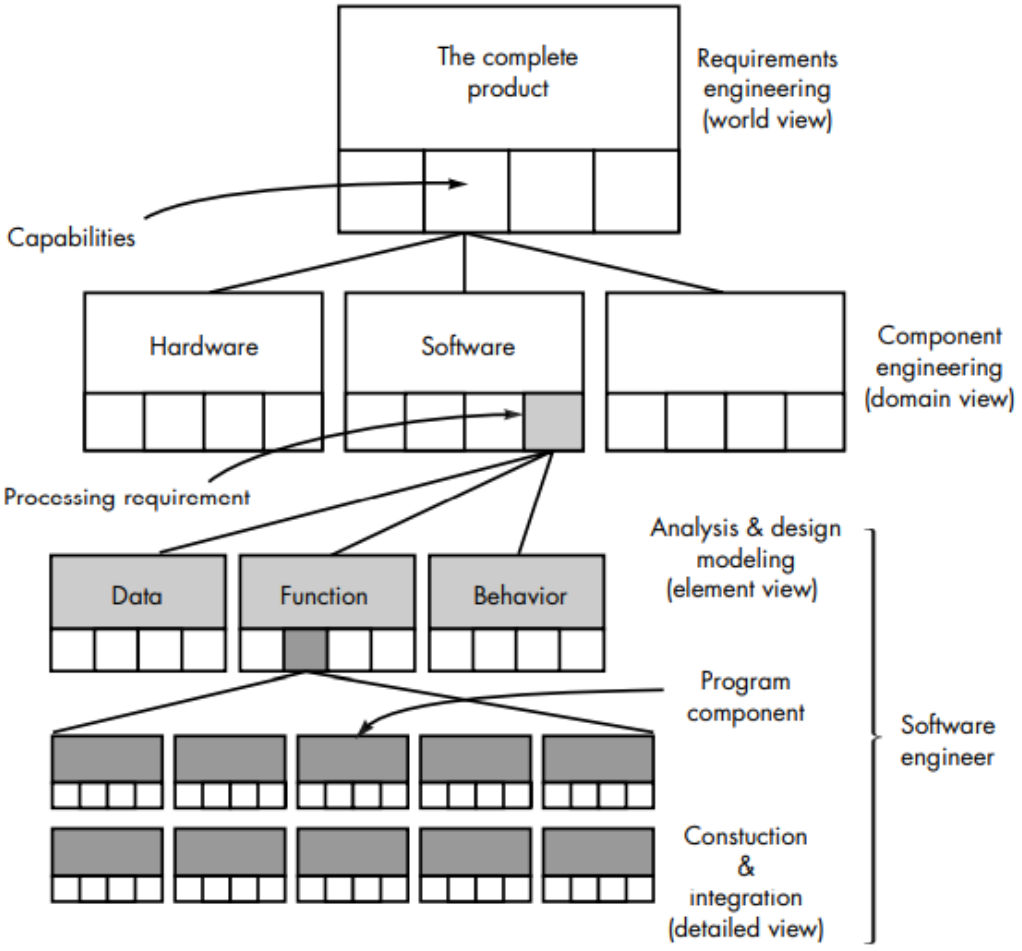


Fig 4 The product Engineering Hierarchy

1.5 Requirement Engineering

1. Before we start to develop our software, it becomes quite essential for us to understand and document the exact requirement of the customer. Experienced members of the development team carry out this job. They are called as system analysts.

2. The analyst starts requirements gathering and analysis activity by collecting all information from the customer which could be used to develop the requirements of the system. He then analyzes the collected information to obtain a clear and thorough understanding of the product to be developed, with a view to remove all ambiguities and inconsistencies from the initial customer perception of the problem. The following basic questions pertaining to the project should be clearly understood by the analyst in order to obtain a good grasp of the problem:

    1 What is the problem?

    2 Why is it important to solve the problem?

    3 What are the possible solutions to the problem?

    4 What exactly are the data input to the system and what exactly are the data output by the system?

    5 What are the likely complexities that might arise while solving the problem?

    6 If there are external software or hardware with which developed software has to interface, then what exactly would the data interchange formats with the external system be

3 When the analyst detects any inconsistencies, anomalies or incompleteness in the gathered requirements, he resolves them by carrying out further discussions with the end users and the customers.

4 Parts of a SRS document

    4.1 Functional Requirement-The functional requirements part discusses the functionalities required from the system. The system is considered to perform a set of high-level functions {fi }. The functional view of the system is shown in fig below. . Each function f i of the system can be considered as a transformation of a set of input data (i1) to the corresponding set of output data (oi). The user can get some meaningful piece of work done using a high-level function.

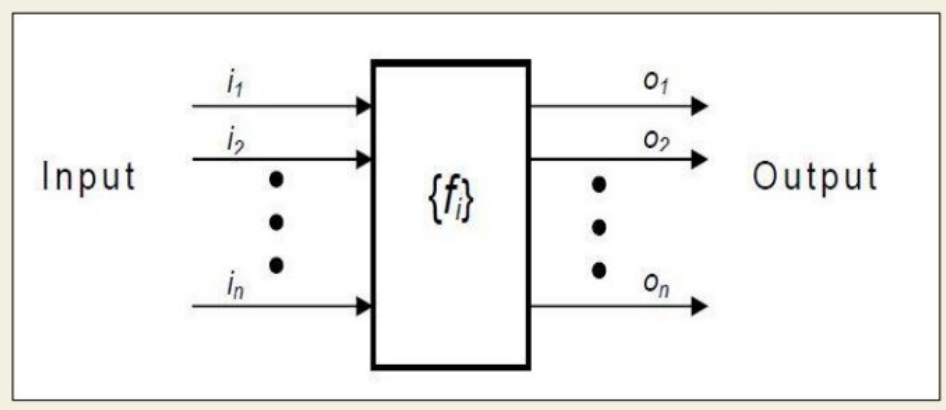


Fig 5 View of a system performing a set of functions

4.2 Non-functional Requirements- Non-functional requirements deal with the characteristics of the system which cannot be expressed as functions - such as the maintainability of the system, portability of the system, usability of the system, etc.

4.3 Goal of Implementation- The goals of implementation part documents some general suggestions regarding development. These suggestions guide trade-off among design goals. The goals of implementation section might document issues such as revisions to the system functionalities that may be required in the future, new devices to be supported in the future, reusability issues, etc. These are the items which the developers might keep in their mind during development so that the developed system may meet some aspects that are not required immediately.

5 Identifying functional requirements from a problem description

5.1 The high-level functional requirements often need to be identified either from an informal problem description document or from a conceptual understanding of the problem. Each highlevel requirement characterizes a way of system usage by some user to perform some meaningful piece of work. There can be many types of users of a system and their requirements from the system may be very different. So, it is often useful to identify the different types of users who might use the system and then try to identify the requirements from each user's perspective.

5.2 Example: - Consider the case of the library system, where – F1: Search Book function, Input: an author's name, Output: details of the author's books and the location of these books in the library.

6 Properties of a Good SRS

The important properties of a good SRS document are the following

1 Concise – The SRS document should be concise and at the same time umbiguous, consistent, and complete.

2 Structured- It should be well Structured. A well structured document is easy to understand and modify.

3. Black-Box view- It should only specify what the system do and refrain from stating how to do these.

4 Conceptual Integrity- It should show conceptual integrity so that the reader can easily understand it.

5 Response to undesired events- It should characterize acceptable responses to undesired events. These are called system response to exceptional conditions.

6. Verifiable- All requirements of the system as documented in the SRS document should be verifiable.

1.6 System Modelling

1 Every computer-based system can be modelled as an information transform using an input-processing-output template.

2 Hatley and Pirbhai [HAT87] have extended this view to include two additional system features—user interface processing and maintenance and self-test processing.

3 Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

4 The system engineer allocates system elements to each of five processing regions within the template: (1) user interface, (2) input, (3) system function and control, (4) output, and (5) maintenance and self-test as shown in figure below

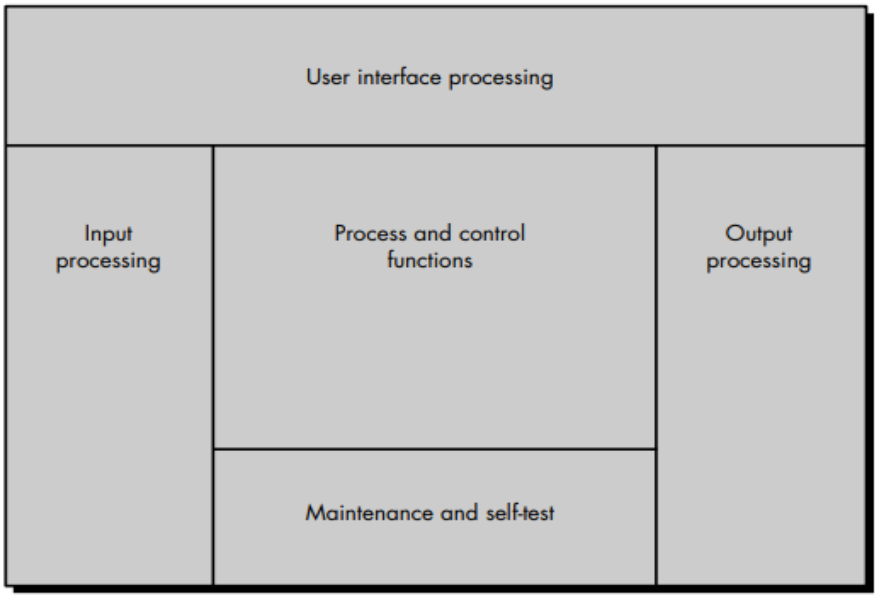


Fig 6 System Model Template

1.4 Case Tools

1 Computer aided software engineering (CASE) is the implementation of computer facilitated tools and methods in software development. CASE is used to ensure a high-quality and defect-free software.

2 CASE ensures a check-pointed and disciplined approach and helps designers, developers, testers, managers and others to see the project milestones during development.

3 CASE can also help as a warehouse for documents related to projects, like business plans, requirements and design specifications.

4 One of the major advantages of using CASE is the delivery of the final product, which is more likely to meet real-world requirements as it ensures that customers remain part of the process.

5 CASE illustrates a wide set of labor-saving tools that are used in software development. It generates a framework for organizing projects and to be helpful in enhancing productivity.

6 There was more interest in the concept of CASE tools years ago, but less so today, as the tools have morphed into different functions, often in reaction to software developer needs.

7 The concept of CASE also received a heavy dose of criticism after its release.

8 Types of Case Tools

8.1 Diagramming tools- It helps in diagrammatic and graphical representations of the data and system processes. It represents system elements, control flow and data flow among different software components and system structure in a pictorial form. For example, Flow Chart Maker tool for making state-of-the-art flowcharts.

8.2 Computer Display and Report Generators- It helps in understanding the data requirements and the relationships involved.

8.3 Analysis Tools-It focuses on inconsistent, incorrect specifications involved in the diagram and data flow. It helps in collecting requirements, automatically check for any irregularity, imprecision in the diagrams, data redundancies or erroneous omissions.

8.4 Central Repository- It provides the single point of storage for data diagrams, reports and documents related to project management.

8.5 Document Generators- It helps in generating user and technical documentation as per standards. It creates documents for technical users and end users. For example, Doxygen, DrExplain, Adobe RoboHelp for documentation.

8.6 Code Generators- It aids in the auto generation of code, including definitions, with the help of the designs, documents and diagrams.

9 Advantages

1 As special emphasis is placed on redesign as well as testing, the servicing cost of a product over its expected lifetime is considerably reduced.

2 The overall quality of the product is improved as an organized approach is undertaken during the process of development.

3 Chances to meet real-world requirements are more likely and easier with a computer-aided software engineering approach.

4 CASE indirectly provides an organization with a competitive advantage by helping ensure the development of high-quality products.

10 Disadvantages

1 Cost-Using case tool is a very costly. Mostly firms engaged in software development on a small scale do not invest in CASE tools because they think that the benefit of CASE are justifiable only in the development of large systems.

2 Learning Curve- In most cases, programmers productivity may fall in the initial phase of implementation , because user need time to learn the technology. Many consultants offer training and on-site services that can be important to accelerate the learning curve and to the development and use of the CASE tools.

3 Tool Mix- It is important to build an appropriate selection tool mix to urge cost advantage CASE integration and data integration across all platforms is extremely important.

1.5 Attributes of Good Software

1. A software product can be judged by what it offers and how well it can be used. This Software must satisfy on the following grounds

   1.1 Operational

   1.2 Transitional

   1.3 Maintenance

2. Operational consist of following things such as Budget, usability, efficiency, correctness, functionality, dependability, security, safety.

3 Transitional aspect is important when the software is moved from one platform to another. It consist of following items such as Portability, Interoperability, Reusability, Adaptability.

4 Maintenance- This aspect briefs how well a software has the capabilities to maintain itself in the ever changing environment. It consist of following things such as Modularity, Maintainability, Flexibility, Scalability

Questions

Q1 Define Software Engineering as IEEE standard.

Q2. Explain the conventional methods of Software Engineering

Q3. Justify with your answer. Explain what are the attributes of a good software

Q4. Illustrate with example the software process mechanism?

UNIT 1

2

SOCIO-TECHNICAL SYSTEM

Unit Structure

2.0 Objectives

At the end of this unit, the student will be able to

- To explain what a socio-technical system is and the distinction between this and a computer based system
- To introduce the concept of emergent system properties such as reliability and security
- To explain system engineering and system procurement processes
- To explain why the organizational context of a system affects its design and use
- To discuss legacy systems and why these are critical to many businesses.

2.1 Essential Characteristics of Socio Technical Systems

1. The term system is one that is universally used. We talk about computer systems, operating systems, payment systems, the educational systems, the systems of government and so on.

2. These are all obviously quite different uses of the word system although they share the characteristic that, somehow, the system is more than simply the sum of its parts.

3. A useful working definition of these types of systems is: A system is a purposeful collection of interrelated components that work together to achieve some objective.

4 This general definition embraces a vast range of systems. For example, a very simple system such as a pen may only include three or four hardware components. By contrast, an air traffic control system includes thousands of hardware and software components plus human users who make decisions based on information from the computer system.

5. Systems that include software fall in to two categories

   5.1 Technical computer-based systems are systems that include hardware and software components but not procedures and processes. Examples of technical systems include televisions, mobile phones and most personal computer software.

   5.2 Socio-technical systems include one or more technical systems but, crucially, also include knowledge of how the system should be used to achieve some broader objective. This means that these systems have defined operational processes, include people (the operators) as inherent parts of the system, are governed by organisational policies and rules and may be affected by external constraints such as national taws and regulatory policies.

6 Essential Characteristics of Socio-Technical Systems are as follows

   6.1 They have emergent properties that are properties of the system as a whole rather than associated with individual parts of the system. Emergent properties depend on both the system components and the relationships between them. As this is so complex, the emergent properties can only be evaluated once the system has been assembled.

   6.2 They are often nondeterministic. This means that, when presented with a specific input, they may not always produce the same output. The system's behaviour depends on the human operators, and people do not always react in the same way. Furthermore, use of the system may create new relationships between the system components and hence change its emergent behaviour.

6.3 The extent to which the system supports organisational objectives does not just depend on the system itself. It also depends on the stability of these objectives, the relationships and conflicts between organisational objectives and how people in the organisation interpret these objectives. New management may reinterpret the organisational objective that a system is designed to support, and a successful' system may then become a failure.

7 Software engineers should have some knowledge of socio-technical systems and systems engineering (White, et al., 1993; Thayer, 2(02) because of the importance of software in these systems. For example, there were fewer than 10 megabytes of software in the US Apollo space program that put a man on the moon in 1969, but there are about 100 megabytes of software in the control systems of the Columbus space station.

8 A characteristic of all systems is that the properties and the behaviour of the system components are inextricably intermingled. The successful functioning of each system component depends on the functioning of some other components. Thus, software can only operate if the processor is operational.

9 Systems are usually hierarchical and so include other systems. For example, a police command and control system may include a geographical information system to provide details of the location of incidents. These other systems are called sub-systems.

10 A characteristic of sub-systems is that they can operate as independent systems in their own right. Therefore, the same geographical information system may be used in different systems.

11 Because software is inherently flexible, unexpected systems problems are often left to software engineers to solve. Say a radar installation has been sited so that ghosting of the radar image occurs. It is impractical to move the radar to a site with less interference, so the systems engineers have to find another way of removing this ghosting.

12 This situation, where software engineers are left with the problem of enhancing software capabilities without increasing hardware cost, is very common. A good example of this was the failure of the Denver airport baggage system (Swartz, 1996), where the controlling software was expected to deal with several limitations in the equipment used.

13 Software engineering is therefore critical for the successful development of complex, computer-based socio-technical systems. As a software engineer, you should not simply be concerned with the software itself but you should also have a broader awareness of how that software interacts with other hardware and software systems and how it is supposed to be used.

14 This knowledge helps us understand the limits of software, to design better software and to participate as equal members of a systems engineering group.

2.2 Emergent System Properties

1. The complex relationships between the components in a system mean that the system is more than simply the sum of its parts. It has properties that are properties of the system as a whole.

2. These emergent properties (Checkland, 1981) cannot be attributed to any specific part of the system. Rather, they emerge only once the system components have been integrated. Some of these properties can be derived directly from the: comparable properties of sub-systems.

3 However, more often, they result from complex sub-system interrelationships that cannot, in practice, be derived from the properties of the individual system components. Examples of some emergent properties are shown in table 1 given below.

4 There are two types of emergent properties

4.1 Functional emergent properties appear when all the parts of a system work together to achieve some objective. For example, a bicycle has the functional property of being a transportation device once it has been assembled from its components.

4.2 a) Non-functional emergent properties relate to the behaviour of the system in its operational environment. Examples of non-functional properties are reliability, performance, safety and securi1ty. These are often critical for computer-based systems, as failure to achieve some minimal defined level in these properties may make the system unusable. b) Some users may not need some system function, so the system may be acceptable without them. However, a system that is unreliable or too slow is likely to be rejected by all its users.

5 To illustrate the complexity of emergent properties, consider the property of system reliability. Reliability is a complex concept that must always be considered at the system level rather than at the individual component level. The component in a system are interdependent, so failures in one component can be propagated through the system and affect the operation of other component failures propogate through the system.

6. It is often difficult to anticipate how the consequences of component failures propagate through the system. Consequently, you cannot make good estimates of overall system reliability from data about the reliability of system components.

7 There are three related influences on the overall reliability of a system

7.1 Hardware reliability What is the probability of a hardware component failing and how long does it take to repair that component?

7.2 Software reliability How likely is it that a software component will produce an incorrect output? Software failure is usually distinct from hardware failure in that software does not wear out. Failures are usually transient so the system carries on working after an incorrect result has been produced

7.3 Operator reliability How likely is it that the operator of a system will make an error?

8 The software can then behave unpredictably. Operator error is most likely in conditions of stress, such as when system failures are occurring. These operator errors may further stress the hard ware, causing more failures, and so on.

9 Thus, the initial, recoverable failure can rapidly develop into a serious problem requiring a complete system shutdown.

10 Like reliability, other emergent properties such as performance or usability are hard to assess but can be measured after the system is operational. Properties such as safety and security, however, pose different problems.

11 A secure system is one that does not allow unauthorised access to its data but it is clearly impossible to predict all possible modes of access and explicitly forbid them. Therefore, it may only be possible to assess these properties by default. That is, you only know that a system is insecure when someone breaks into it.

12 Table 1 Examples of Emergent Properties

Sr.No	Property	Description
1	Volume	The volume of a system varies depending on how the component assemblies are arranged and connected
2	Reliability	System reliability depends on component reliability but unexpected interactions can cause new types of failure and therefore affect the reliability of the system
3	Security	The security of the system is a complex property that cannot be easily measured. Attacks may be devised that were not anticipated by the system designers and so may defeat built-in safeguard.
4	Repairability	This property reflects how easy it is to fix a problem with the system once it has been discovered. It depends on being able to diagnose the problem. access the components that are faulty Mel modify or replace these components.
5	Usability	This property reflects how easy it is to use the system. It depends on the technical system components, its operators and its operating environment.

2.3 Systems Engineering

1. Systems engineering is the activity of specifying, designing, implementing, validating, deploying and maintaining socio-technical systems.

2. Systems engineers are not just concerned with software but also with hardware and the system's interactions with users and its environment. They must think about the services that the system provides, the constraints under which the system must be built and operated and the ways in which the system is used to fulfil its purpose.

3. software engineers need an understanding of system engineering because problems of software engineering are often a result of system engineering decisions.

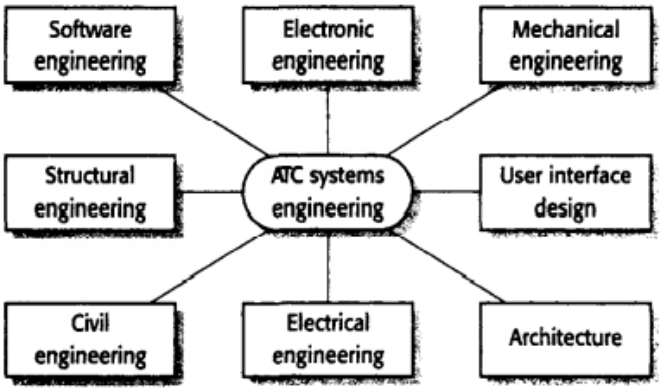


Fig 1 Disciplines involved in Systems Engineering

4. There are important distinctions between the system engineering process and the software development process

   4.1 Limited scope for rework during system development Once some system engineering decisions, such as the siting of base stations in a mobile phone system, have been made, they are very expensive to change. Reworking the system design to solve these problems is rarely possible. One reason software has become so important in systems is that it allows changes to be made during system development, in response to new requirements.

   4.2 Interdisciplinary involvement Many engineering disciplines may be involved in system engineering. There is a lot of scope for misunderstanding because different engineers use different terminology and conventions.

5 Systems engineering is an interdisciplinary activity involving teams drawn from various backgrounds. System engineering teams are needed because of the wide knowledge required to consider all the implications of system design decisions. Fig 1 shows some of the disciplines that may be involved in the system engineering team for an air traffic control (ATC) system that uses radars and other sensors to determine aircraft position.

6 For many systems, there are almost infinite possibilities for trade-offs between different types of sub-systems. Different disciplines negotiate to decide how functionality should be provided. Often there is no correct' decision on how a system should be decomposed.

7 Rather, you may have several possible alternatives, but you may not be able to choose the best technical solution. Say one alternative in an air traffic control system is to build new radars rather than refit existing installations.

8 If the civil engineers involved in this process do not have much other work, they may favour this alternative because it allows them to keep their jobs. They may then rationalise this choice with technical arguments.

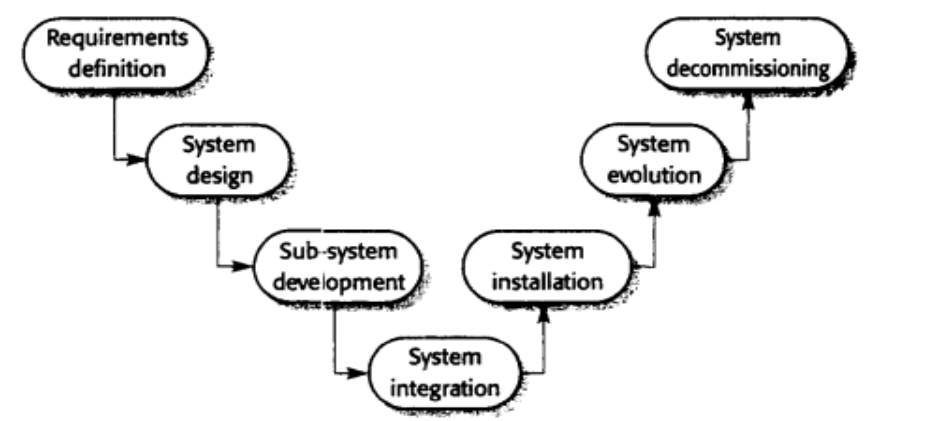


Fig 2 System Engineering Process

9 The above figure explains the phases of the systems engineering process. This process was an important influence on the 'waterfall' model of the software process.

10 Types of System Engineering

10.1 System requirement definition

1. System requirements definitions specify what the system should do (its functions) and its essential and desirable system properties. As with software requirements analysis.

2. As with software requirements analysis creating system requirements definitions involves consultations with system customers and end-users. This requirements definition phase usually concentrates on deriving three types of requirement

    2.1 Abstract functional requirements The basic functions that the system must provide are defined at an abstract level. More detailed functional requirements specification takes place at the sub-system level. For example, in an air traffic control system, an abstract functional requirement would specify that a flight-plan database should be used to store the flight plans of all aircraft entering the controlled airspace.

However, you would not normally specify the details of the database unless they affected the requirements of other sub-systems.

2.2 System properties These are non-functional emergent system properties such as availability, performance and safety. These non functional system properties affect the requirements for all sub-systems.

2.3 Characteristics that the system must not exhibit It is sometimes as important to specify what the system must not do as it is to specify what the system should do. For example, if you are specifying an air traffic control system, you might specify that the system should not present the controller with too much information.

3 An Important part of the requirements definition phase is to establish a set of overall objectives that the system should meet. These should not necessarily be expressed in terms of the system', functionality but should define why the system is being procured for a particular environment.

4 For eg specifying a system for an office building to provide for fire protection and for intruder detection. A statement of objectives based on the system functionality might be to provide a fire and intruder alarm system for the building that will provide internal and external warning office or unauthorised intrusion.

5 This objective states explicitly that there needs to be an alarm system that provides warnings of undesired events. Such a statement might be appropriate if you were replacing an existing alarm system. By contrast, a broader statement of objectives might be to ensure that the normal functioning of the work carried out in the building is not seriously disrupted by events such as fire and unauthorized intrusion.

6 A fundamental difficulty in establishing system requirements is that the problems that complex systems are usually built to help tackle are usually 'wicked problems. A wicked problem is a problem that is so complex and where there are so many related entities that there is no definitive problem specification.

10.2 System Design

1 System design is concerned with how the system functionality is to be provided by the components of the system. The activities involved in this process are

1.1 Partition requirements You analyse the requirements and organise them into related groups. There are usually several possible partitioning options, and you may suggest a number of alternatives at this stage of the process.

1.2 Identify sub-systems You should identify sub-systems that can individually or collectively meet the requirements. Groups of requirements are usually related to sub-systems, so this activity and requirements partitioning may be amalgamated. However, sub-system identification may also be influenced by other organisational or environmental factors.

1.3 Assign requirements to sub-systems You assign the requirements to subsystems. In principle, this should be straightforward if the requirements partitioning is used to drive the sub-system identification. In practice, there is never a clean match between requirements partitions and identified sub-systems. Limitations of externally purchased sub-systems may mean that you have to change the requirements to accommodate these constraints.

1.4 Specify sub-system functionality You should specify the specific functions provided by each sub-system. This may be seen as part of the system design phase or, if the sub-system is a software system, part of the requirements specification activity for that system. You should also try to identify relationships between sub-systems at this stage.

1.5 Define sub-system interfaces You define the interfaces that are provided and required by each sub-system. Once these interfaces have been agreed upon, it becomes possible to develop these sub-systems in parallel.

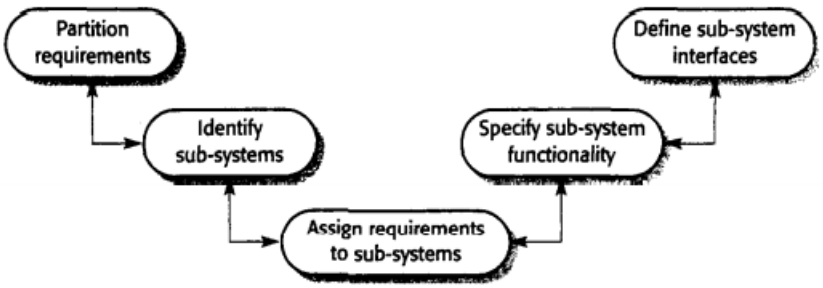


Fig 2 The System design process

2 The spiral process reflects the reality that requirements affect design decisions and vice versa and so it makes sense to interleave these processes. Starting in the centre, each round of the spiral may add detail to the requirements and the design. Some rounds may focus on requirements, some on design. Sometimes, new knowledge collected during the requirements and design process means that the problem statement itself has to be changed.

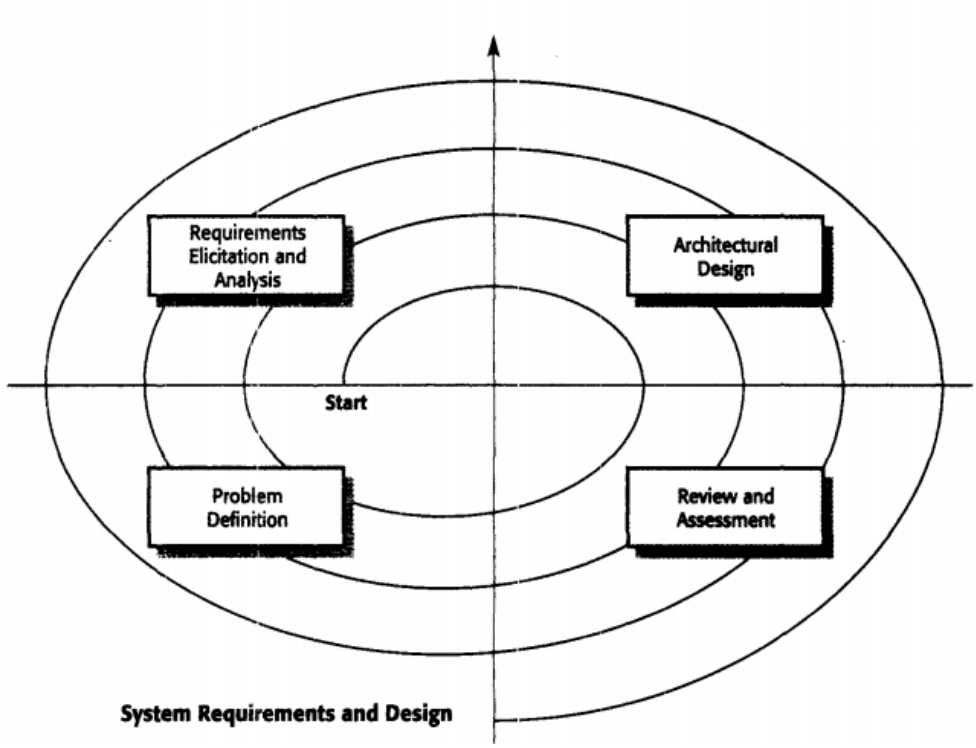


Fig 4 A spiral model of requirements and design

10.3 System Modelling

1. During the system requirements and design activity, systems may be modelled as a set of components and relationships between these components. The system architecture may be presented as a block diagram showing the major sub-systems and the interconnections between these sub-systems.

2 When drawing a block diagram, you should represent each sub-system using a rectangle, and we should show relationships between the sub-systems using arrows that link these rectangles. The relationships indicated may include data flow, is used by' relationship or some other type of dependency relationship.

3 For eg figure shown below depict the decomposition of an intruder alarm system in to its principal components.

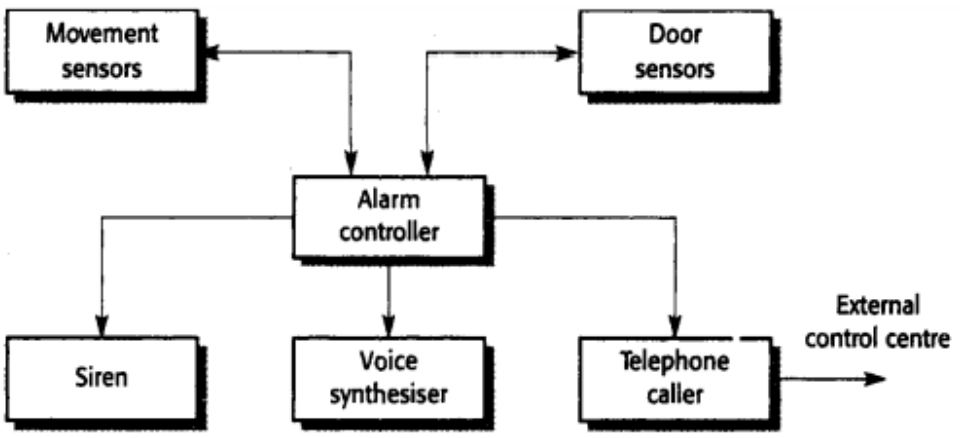


Fig 5 A simple Burglar Alarm System

4 Every component of above figure is explained in table 2 given below

Sr. No	Subsystem	Description
1	Movement Sensors	Detects movement in the rooms monitored by the system
2	Door Sensors	Detects door opening in the external doors of the building
3	Alarm Controller	Controls the operation of the system
4	Siren	Emits an audible warning when an intruder is suspected
5	Voice Synthesizer	Synthesises a voice message giving the location of the suspected intruder
6	Telephone Caller	Makes eternal calls to notify security, the police etc

5 Figure shown below the architecture of a much larger system for air traffic control. Several major sub-systems shown are themselves large systems. The arrowed lines that link these systems show information flow between these sub-systems.

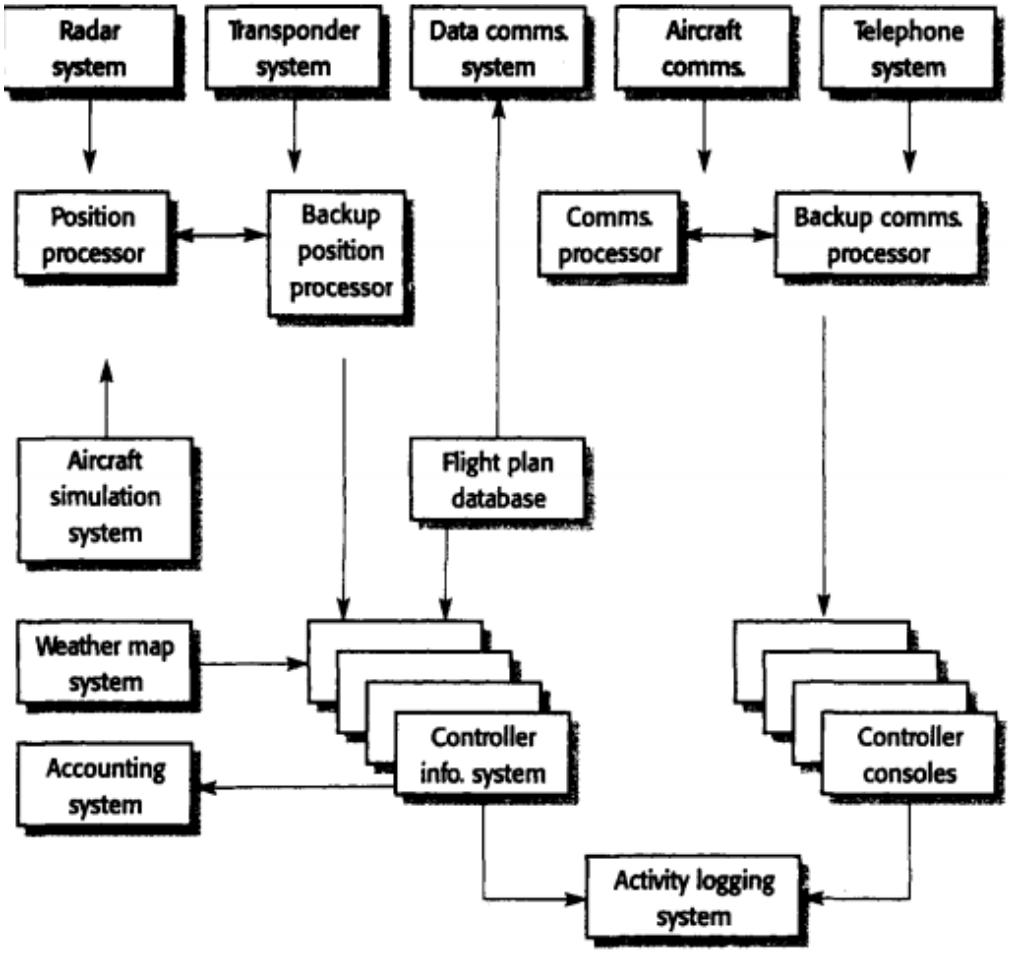


Fig 6 An architectural model of an air traffic control system

10.4 Sub-System development

1 During sub-system development, the sub-systems identified during system design are implemented. This may involve starting another system engineering process for sub-system is software, a software process involving requirements, design, implement.

2 Occasionally, all sub-systems are developed from scratch during the development process. Normally, however, some of the sub-systems are commercial, off-the-shelf (COTS) systems that are bought for integration into the system.

3 Sub-systems are usually developed in parallel. When problems are encountered that cut across sub-system boundaries, a system modification request must be made. Where systems involve extensive hardware

engineering, making modifications after manufacturing has started is usually very expensive.

10.5 System Integration

1 During the systems integration process, you take the independently developed sub systems and put them together to make up a complete system. Integration can be done using a 'big bang' approach, where all the sub-systems are integrated at the same time.

2 However, for technical and managerial purposes, an incremental integration process where sub-systems are integrated one at a time is the best approach, for two reasons as follows

2.1 It is usually impossible to schedule the development of all sub-systems so that they are all finished at the same time.

2.2 a) Incremental integration reduces the cost of error location. If many sub-systems are simultaneously integrated, an error that arises during testing may be in any of these sub systems. b) When a single sub-system is integrated with an already working system, errors that occur are probably in the newly integrated sub-system or in the interactions between the existing subsystems and the new sub-system.

3 Once the components have been integrated, an extensive programme of system testing takes place.

4 Sub-system faults that are a consequence of invalid assumptions about other subsystems are often revealed during system integration. This may lead to disputes between the various contractors responsible for the different sub-systems.

5 As more and more systems are built by integrating COTS hardware and software components, system integration is becoming increasingly important.

10.6 System Evolution

1 Large, complex systems have a very long lifetime. During their life, they are changed to correct errors in the original system requirements and to implement new requirements that have emerged. The organisation that uses the system may reorganise itself and hence use the system in a different way.

The external environment of the system may change, forcing changes to the system.

2 System evolution is like software evolution is inherently costly for several reasons

1 Proposed changes have to be analysed very carefully from a business and a technical perspective. Changes have to contribute to the goals of the system and should not simply be technically motivated.

2 Because sub-systems are never completely independent, changes to one subsystem may adversely affect the performance or behaviour of other subsystems. Consequent changes to these sub-systems may therefore be needed.

3 The reasons for original design decisions are often unrecorded. Those responsible for the system evolution have to work out why particular design decisions were made.

5 As systems age, their structure typically becomes corrupted by change so the costs of making further changes increases.

2.4 Components of Systems such as Organization, People and Computers

1. Socio-technical systems are enterprise systems that are intended to help deliver some organisational or business goal. This might be to increase sales, reduce material used in manufacturing, collect taxes, maintain a safe airspace, etc.

2 Because they are embedded in an organisational environment, the procurement, development and use of these system is influenced by the organisation's policies and procedures and by its working culture.

3 The users of the system are people who are influenced by the way the organisation is managed and by their interactions with other people inside and outside of the organisation.

4 Therefore, when we are trying to understand the requirements for a socio-technical system we need to understand its organisational environment. If we don't understand the systems may not meet business needs, and users and their managers may reject the system.

5 Human and organisational factors from the system's environment that affect the system design include as follows

5.1 Process changes Does the system require changes to the work processes in the environment? If so, training will certainly be required. If changes are significant, or if they involve people loosing their jobs, there is a danger that the users will resist the introduction of the system.

5.2 Job changes Does the system de-!;kill the users in an environment or cause them to change the way they work? If so, they may actively resist the introduction of the system into the organisation. Designs that involve managers having to change their way of working to fit the computer system are often resented. The managers may feel that their status in the organisation is being reduced by the system.

5.3 Organisational changes Does the system change the political power structure in an organisation? For example, if an organisation is dependent on a complex system, 1hose who know how to operate the system have a great deal of political power.

6 Organizational Processes

1. The development process is not the only process involved in systems engineering. It interacts with the system procurement process and with the process of using and operating the system. This depicts in figure given below

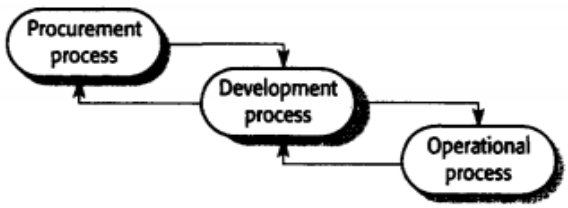


Fig7 Procurement, development and operational processes

2 The procurement process is normally embedded within the organisation that will buy and use the system (the client organisation). The process of system procurement is concerned with making decisions about the best way for an organisation to acquire a system and deciding on the best suppliers of that system.

3 Large complex systems usually consist of a mixture of off-the-shelf and specially built components. One reason why more and more software is included in systems is that it allows more use of existing hardware components, with the software acting as a 'glue' to make these hardware components work together effectively.

4 Figure given below depicts the procurement process for both existing systems and systems teams that have to be specially designed. Some important points about the process shown in the diagram are as follows

5.1 Off the shelf components do not usually much requirements exactly, unless the requirements have been written with these components in mind. Therefore, choosing a system means that we have to find the closest match between the system requirements and the facilities offered by off the shelf systems.

5.2 When a system is to be built specially, the specification of requirements acts as the basis of a contract for the system procurement. It is therefore a legal, as well as a technical document.

5.3 After a contractor to build a system has been selected, there is a contract negotiation period where we may have to negotiate further changes to the requirements and also discussed the issues such as the cost of changes to the system.

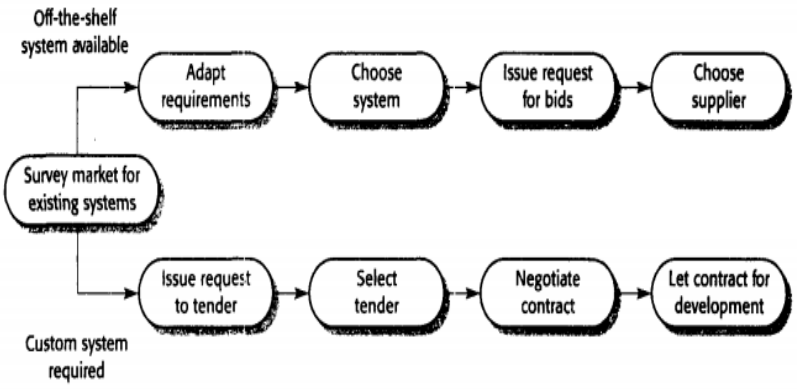


Fig 8 The System procurement process

6 The supplier who is usually called the principal contractor may contract out the development of different sub-systems to a number of sub-contractors. For large systems, such as air traffic control systems, a group of suppliers may form a consortium to bid for the contract.

7 The procurer deals with the contractor rather than the sub-contractors so that there is a single procurer/supplier interface. The sub-contractors design and build parts of the system to a specification that is produced by the principal contractor.

8 Operational processes are the processes that are involved in using the system for its defined purpose. For example, operators of an air traffic control system follow specific processes when aircraft enter and leave airspace, when they have to change height or speed, when an emergency occurs and so on.

9 The key benefit of having people in a system is that people have a unique capability of being able to respond effectively to unexpected situations even when they have never had direct experience of these situations.

10 Designers should design operational processes to be flexible and adaptable. The operational processes should not be too constraining, they should not require operations to be done in a particular order, and the system software should not rely on a specific process being followed.

11 An issue that may only emerge after the system goes into operation is the problem of operating the new system alongside existing systems. There may be physical problems of incompatibility, or it may be difficult to transfer data from one system to another.

2.5 Dealing Legacy Systems

1. The time and effort required to develop a com-lex system, large computer-based usually have a long lifetime. For example military systems are often designed for a 20 year lifetime and much of the world's air traffic control still relies on software and operational processes that were originally developed in the 1960's and 1970's.

2 Their development continues throughout their life with changes to accommodate new requirements, new operating platforms, and so forth.

3 Legacy systems are socio-technical computer-based systems that have been developed in the past, often using older or obsolete technology. These systems include not only hardware and software but also legacy processes and procedures-old ways of doing things that are difficult to change because they rely on legacy software.

4 Legacy systems are often business-critical systems. They are maintained because it is too risky to replace them. For example, for most banks the customer accounting system was one of their earliest systems. Organisational policies and procedures may rely on this system. If the bank were to scrap and replace the customer accounting software (which may run on expensive mainframe hardware) then there would be a serious business risk if the replacement system didn't work properly.

5 Figure shown below depicts the logical parts of a legacy system and their relationships

5.1 System hardware-In many cases" legacy systems have been written for mainframe hardware that is no longer available, that is expensive to maintain and that may not be compatible with current organisational IT purchasing policies.

5.2 Support software-The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.

5.3 Application software- The application system that provides the business services is usually composed of a number of separate programs that have been developed at different times. Sometimes the term legacy system means this application software system rather than the entire system.

5.4 Application data -These are the data that are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent and may be duplicated in several files.

5.5 Business processes -These are processes that are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting all order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.

5.6 Business policies and rules -These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

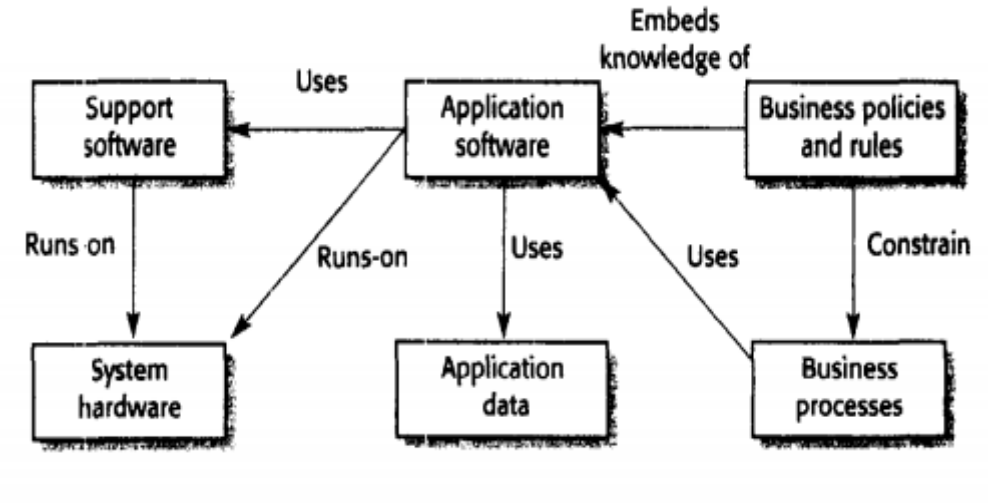


Fig 9 Legacy System Components

6 An alternative way of looking at these components of a legacy system is as a series of layers as shown in figure below. Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then we should be able to make changes within a layer without affecting either of the adjacent layers.

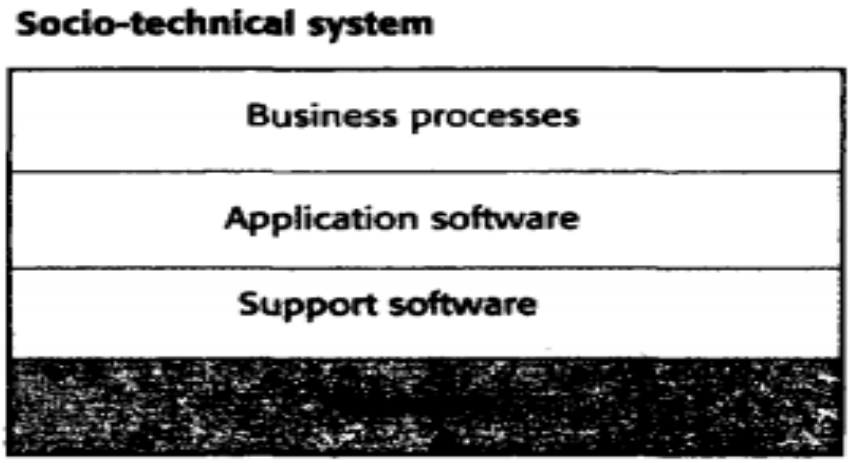


Fig 10 Layered model of a Legacy System

Questions

Q1. Explain what is socio-technical system?

Q2 Describe the emergent properties of socio-technical system>

Q3 Illustrate the System engineering process along with it types?

Q4 Describe about the need of Organizational process in socio-technical system?

Q5 Describe in short about Legacy Systems?

❖ ❖ ❖

UNIT 1

3

CRITICAL SYSTEMS

Unit Structure:

3.0 Objectives

At the end of this unit, the student will be able to

- Understand that in a critical system, system failure can have severe human or economic consequences.
- Illustrate four dimensions of system dependability- Availability, reliability, safety and security.
- Understand the importance of dependability with respect to software engineering.

3.1 Types of Critical System

1 Software failures are relatively common. In most cases, these failures cause inconvenience but no serious, long-term damage. However in some systems failure can result in significant economic losses, physical damage or threats to human life. Such type of systems are called critical systems.

2 Critical systems are technical or socio-technical systems that people or businesses depend on. If these systems fail to deliver their services as expected then serious problems and significant losses may result.

3. There are three main types of critical system

   3.1 Safety-critical systems -A system whose failure may result in injury, loss of life or serious environmental damage. An example of a safety-critical system is a control system for a chemical manufacturing plant.

   3.2 Mission-critical systems -A system whose failure may result in the failure of some goal-directed activity. An example of a mission-critical system is a navigational system for a spacecraft.

   3.3 Business-critical systems- A system whose failure may result in very high costs for the business using that system. An example of a business-critical system is the customer accounting system in a bank.

4 The most important emergent property of a critical system is its dependability. The term dependability was proposed by Laprie(1995) to cover the related systems attributes of availability, reliability, safety and security.

5. There are several reasons why dependability is the most important emergent property for critical systems.

   5.1 Systems that are unreliable, unsafe or insecure are often rejected by their users. If users don't trust a system, they will refuse to use it.

   5.2 System failure costs may be enormous. For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system.

   5.3 Untrustworthy systems may cause information loss. Data is very expensive to collect and maintain, it may sometimes be worth more than the computer system on which it is processed.

6 Consequently, critical systems are usually developed using well-tried techniques rather than newer techniques that have not been subject to extensive practical experience.

7 Expensive software engineering techniques that are not cost-effective for noncritical systems may sometimes be used for critical systems development.

8 One reason why these formal methods are used is that it helps reduce the amount of testing required. For critical systems, the costs of verification and validation are usually very high-more than 50% of the total system development costs.

9 Although a small number of control systems may be completely automatic, most critical systems are socio-technical systems where people monitor and control the operation of computer-based systems.

10 There are three system components where critical systems failures may occur

10.1 System hardware may fail because of mistakes in its design, because components fail as a result of manufacturing errors or because the components have reached the end of their natural life.

10.2 System software may fail because of mistakes in its specification, design or implementation.

10.3 Human operators of the system may fail to operate the system correctly. As hardware and software have become more reliable, failures in operation are now probably the largest single cause of system failures.

11 These failures can be interrelated. A failed hardware component may mean system operator have to cope with an unexpected situation and additional workload.

12 As a result, it is particularly important that designers of critical systems take a holistic systems perspective rather than focus on a single aspect of the system. If the hardware, software and operational processes are designed separately without taking the potential weaknesses of other parts of the system take in to account, then it is more likely that errors at interfaces between the various parts of the system.

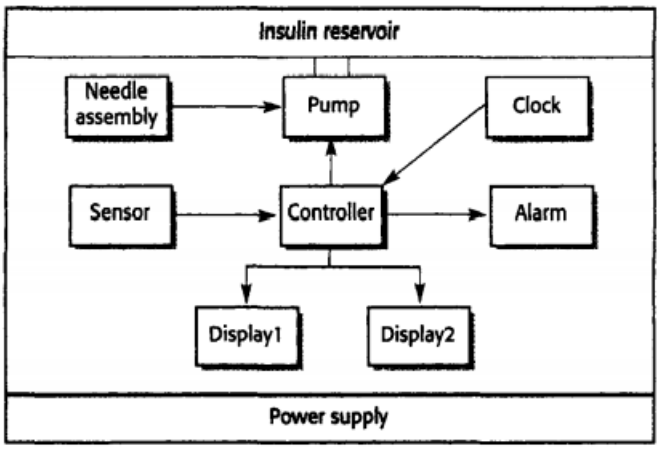


Fig 1 Insulin Pump Structure

3.2 A Simple Safety Critical Systems

1. There are many types of critical computer-based systems, ranging from control systems for devices and machinery to information and e-commerce systems. Understanding the critical system can be very difficult as we need to understand the features and constraints of the application domain where they operate.

2 To understand the safety critical system, a simple example is taken over here is medical system that simulates the operation of the pancreas(internal organ). Diabetes is a relatively common condition where the human pancreas is unable to produce sufficient quantities of a hormone called insulin. Insulin metabolises glucose in the blood. The conventional treatment of diabetes involves regular injections of genetically engineered insulin. Diabetics measure their blood sugar levels using an external meter and then calculate the dose of insulin that they should inject.

3 The problem with this treatment is that the level of insulin in the blood does not just depend on the blood glucose level but is a function of the time when the insulin injection was taken. This can lead to very low levels of blood glucose (if there is too much insulin) or very high levels of blood sugar (if there is too little insulin). Low blood sugar is, in the short term, a more serious condition, as it can result in temporary brain malfunctioning and, ultimately, unconsciousness and death. In the long term, continual high levels of blood sugar can lead to eye damage, kidney damage, and heart problems.

4 Current advances in developing miniaturised sensors have meant that it is now possible to develop automated insulin delivery systems. These systems monitor blood sugar levels and deliver an appropriate dose of insulin when required. Insulin delivery system already exist for the treatment of hospitals patients.

5 A software controlled insulin delivery system might work by using a micro-sensor embedded in the patient to measure some blood parameter that is proportional to the sugar level. This is sent to the pump controller. This controller computes the sugar level and the amount of insulin that is needed. It then sends signals to a pump to deliver the insulin via a permanently attached needle.

6 Figure shown below depict the data flow model for transformation of insulin to a human body. There are two high-level dependability requirements for this insulin pump system

6.1 The system shall be available to deliver insulin when required.

6.2 The system shall perform reliably and deliver the correct amount of insulin to counteract the current level of blood sugar.

7. This might be possible that if system fails then human body gets excessive amount of insulin which threaten the life of the user.

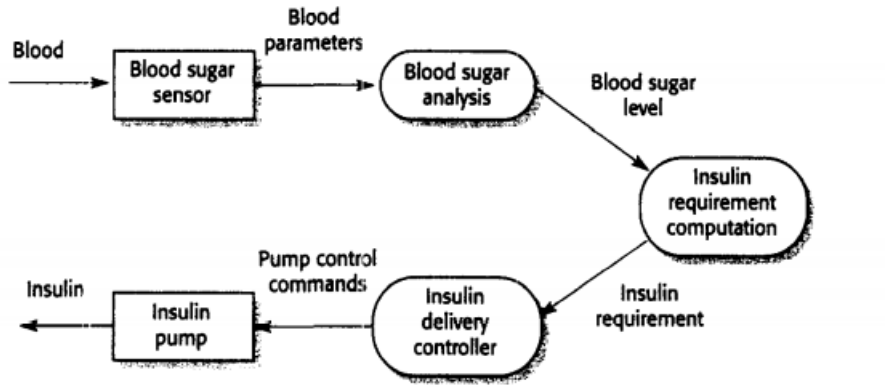


Fig 2 Data-flow model of the insulin pump

3.3 Dependability of A System

1. All of us are familiar with the problem of computer system failure. Without any reason computer systems sometimes crash and fail to deliver the services that have been requested.

2 Program running on these computers may not operate as expected and occasionally, may corrupt the data that is managed by the system.

3 We have learned to live with these failures and few of us completely trust the personnel computers that we normally use.

4 The dependability of a computer system is a property of the system that equates to its trustworthiness. Trustworthiness essentially means the degree of user confidence that the system will operate as they expect and that the system will not fail in normal use.

5 This property cannot be expressed numerically, but we use relative terms such as not dependable, very dependable and ultra dependable to reflect the degrees of trust that we have with system.

6 There are four principal dimensions to dependability as shown in figure below

6.1 Availability- It is the probability, over a given period of time, that the system will correctly deliver services as expected by the user.

6.2 Reliability- It is the probability over a given period of time, that the system will cause damage to people or its environment.

6.3 Safety – It is a judgement of how likely it is that the system will cause damage to a people or its environment.

6.4 Security- It is a judgment of how likely it is that the system can resist accidental or deliberate intrusions.

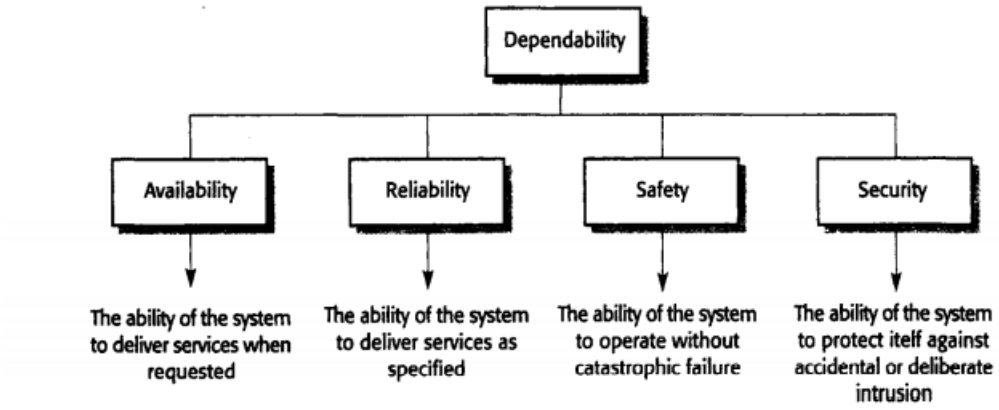


Fig 3 Dimensions of Dependability

7 These are complex properties that can be decomposed in to a number of other, simpler properties. For example security includes integrity and confidentiality. Reliability includes correctness (ensuring the system services are as specified), precision (ensuring information is delivered at an appropriate level of detail) and timeliness (ensuring that information is delivered when it is required).

8 The dependability properties of availability, security, reliability and safety are all interrelated. Safe system operation usually depends on the system

being available and operating reliability. A system may become unreliable because its data has been corrupted by an intruder.

9 Denial-of-service attacks on a system are intended to compromise its availability. If a system that has been proved to be safe is infected with a virus, safe operation can no longer be assumed. It is because of these close links that the notion of system dependability as an encompassing property was introduced.

10 Properties of Dependability

10.1 Repairability- System failures are inevitable, but the disruption caused by failure can be minimised if the system can be repaired quickly. In order for that to happen, it must be possible to diagnose the problem, access the component that has failed and make changes to fix that component. Repairability in software is enhanced when the organisation using the system has access to the source code and has the skills to make changes to it.

10.2 Maintainability- As systems are used, new requirements emerge. It is important to maintain the usefulness of a system by changing it to accommodate these new requirements. Maintainable software is software that can be adapted economically to cope with new requirements and where there is a low probability that making changes will introduce new errors into the system.

10.3 Survivability- A very important attribute for Internet-based systems is survivability, which is closely related to security and availability (Ellison, et al., 1999). Survivability is the ability of a system to continue to deliver service whilst it is under attack and, potentially, while part of the system is disabled. Work on survivability focuses on identifying key system components and ensuring that they can deliver a minimal service. Three strategies are used to enhance survivability-namely, resistance to attack, attack recognition and recovery from the damage caused by an attack.

10.4 Error Tolerance- This property can be considered as part of usability and reflects the extent to which the system has been designed so that user input error are avoided and tolerated. When user errors occur, the system should, as far as possible, detect these errors and either fix them automatically or request the user to re-input their data.

11 All project will not have all these dimensions dependability properties. For eg insulin pump system consist of properties like availability, reliability and safety whereas security is not required.

12 Dependable software includes extra, often redundant, code to perform the necessary checking for exceptional system states and to recover from system faults. This reduces system performance and increases the amount of store required by the software. It also adds significantly to the costs of system development.

13 Because of additional design, implementation and validation costs, increasing the dependability of a system can significantly increase development costs.

14 Figure given below shows relationship between costs and incremental improvements in dependability. The higher the dependability that you need, the more that you have to spend on testing to check that we have reached that level. the exponential nature of this cost/dependability curve, it is not possible to demonstrate that a system is 100% dependable, as the costs of dependability assurance would then be infinite.

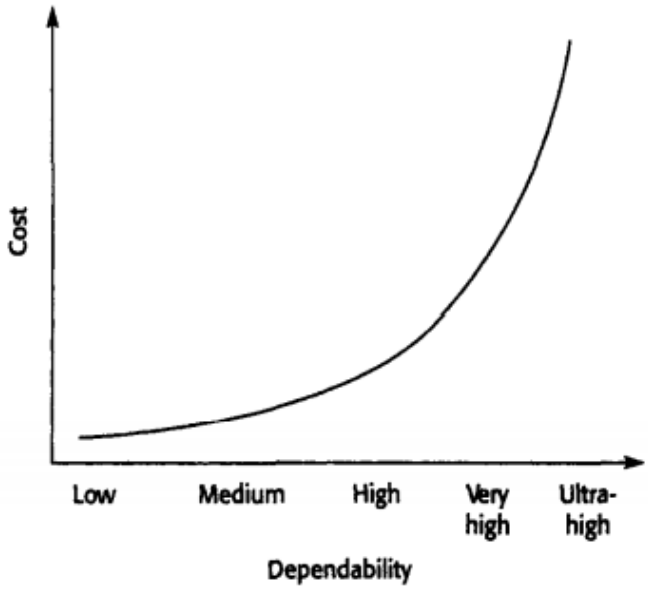


Fig 4 Cost/ Dependability Curve

3.4 Availability And Reliability

1. System availability and reliability are closely related properties that can both be expressed as numerical probability. The reliability of a system is the probability that the system's services will be correctly delivered as specified. The availability of a system is the probability that the: system will be up and running to deliver these services to users when they request them.

2 For example, some systems can have a high availability requirement but a much lower reliability requirement. If users expect continuous service then the availability requirements are high. However, if the consequences of a failure are minimal and the system can recover quickly from these failures then the same system can have low reliability requirements.

3 An example of a system where availability is more critical than reliability is a telephone exchange switch. Users expect a dial tone when they pick up a phone so the system has high availability requirements. However, if a system fault causes a connection to fail, this is often recoverable.

4 Availability does not simply d1epend on the system itself but also on the time needed to repair the faults that make the system unavailable. Therefore, if system A fails once per year, and system B fails once per month, then A is clearly more reliable then B.

5 System reliability and availability may be defined more precisely as follows

   5.1 Reliability-The probability of failure-free operation over a specified time in a given environment for a specific purpose.

   5.2 Availability- The probability that a system at a point in time, will be operational and able to deliver the requested services.

6 The definition of reliability states that the environment in which the system is used and the purpose that it is used for must be taken into account. For example, let's say that we measure the reliability of a word processor in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system.

7 Human perceptions and patterns of use are also significant. For example, say a car has a fault in its windscreen wiper system that results in intermittent failures of the wipers to operate correctly in heavy rain. The reliability of that system as perceived by a driver depends on where they live and use the car.

8 A strict definition of reliability relates the system implementation to its specification. That is, the system is behaving reliably if its behaviour is consistent with that defined in the specification. Reliability and availability are compromised by system failures. These may be a failure to provide a service, a failure to deliver a service as specified, or the delivery of a service in such a way that is unsafe or insecure.

9 Human errors do not inevitably lead to system failures. The faults introduced may be in parts of the system that are never used. Faults do not necessarily result in system errors, as the faulty state may be transient and may be corrected before erroneous behaviour occurs.

10 Table shown below differentiate among the terms fault, error and failure

Sr.No	Term	Description
1	System Failure	An event that Occurs at some point in time when the system does not deliver a service as expected by its users
2	System Error	An erroneous system state that can lead to system behaviour.
3	System Fault	A characteristic of a software system that can lead to a system error. For example, failure to initialise a variable could lead to that variable having the wrong value when it is used.
4	Human Error or Mistake	Human behaviour that results in the introduction of faults in to a system

11 Three approaches that are used to improve the reliability of a system

11.1 Fault Avoidance-Development techniques are used that either minimise the possibility of mistakes and/or that trap mistakes before they result in the introduction of system faults. Examples of such techniques include avoiding error-prone programming language constructs such as pointers and the use of static analysis to detect program anomalies.

11.2 Fault detection and removal -The use of verification and validation techniques that increase the chances that faults will be detected and removed before the system is used. Systematic system testing and debugging is an example of a fault-detection technique.

11.3 Fault tolerance Techniques- that ensure that faults in a system do not result in system errors or that ensure that system errors do not result in system failures. The incorporation of self-checking facilities in a system and the use of redundant system modules are examples of fault tolerance techniques.

12 Software faults cause software failures when the faulty code is executed with a set of inputs that expose the software fault. The code works properly for most inputs. Figure given below depicts a software system as a mapping of an input to an output set.

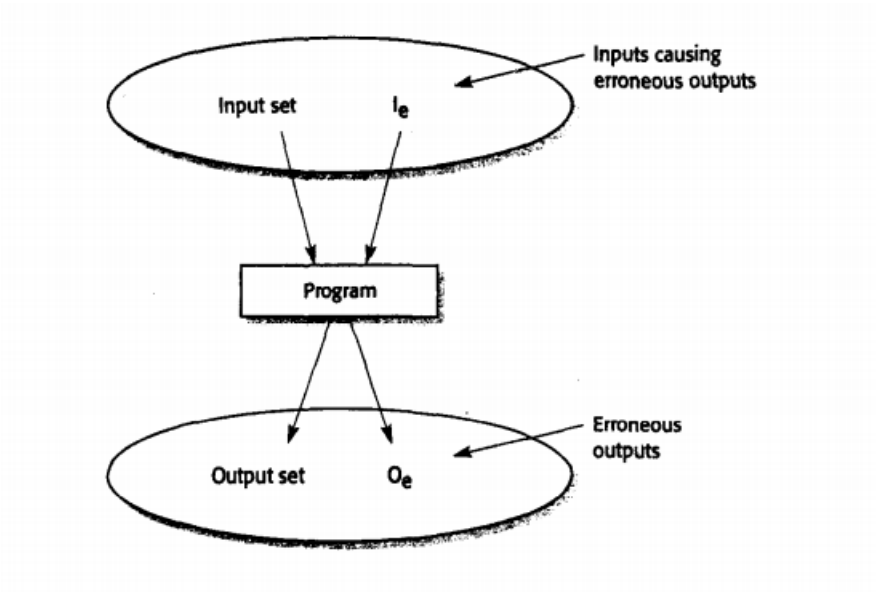


Fig 5 A system as an input/output mapping

13 Some of these inputs or input combinations, shown in the shaded ellipse in Figure given above, cause erroneous outputs to be generated. The software reliability is related to the probability that, in a particular execution of the program, the system input will be a member of the set of inputs, which cause an erroneous output to occur.

14 Each user of a system uses it in different ways. Faults that affect the reliability of the system for one user may never be revealed under someone else's mode of working as shown in figure below.

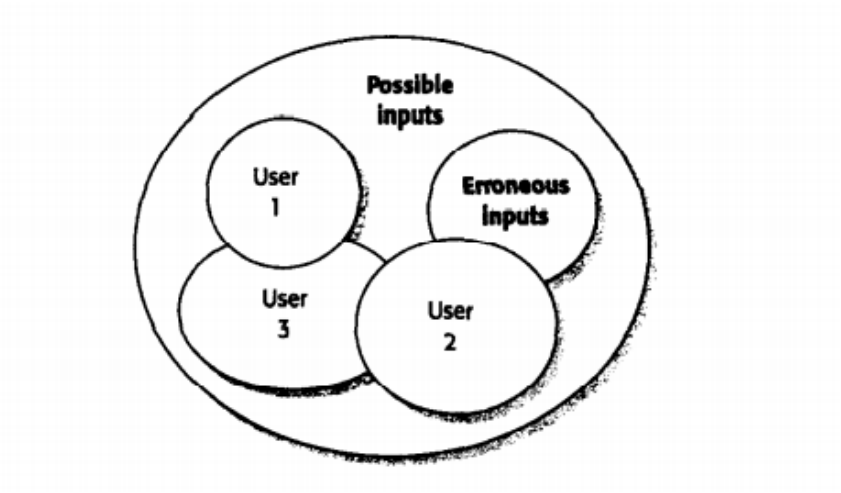


Fig 6 Software Usage Pattern

15 The overall reliability of a program, therefore, mostly depends on the number of inputs causing erroneous outputs during normal use of the system by most users. Software faults that occur only in exceptional situations have little effect on the system's reliability.

16 Users in a soci0-technical system may adapt to software with known faults, and may share information about how to get around these problems. They may avoid using inputs that are known to cause problems so program failures never arise.

3.5 Safety And Security Of Software Systems

1 Safety-critical systems are systems where it is essential that system operation is always safe. That is, the system should never damage people or the system's environment even if the system fails. Examples of safety-critical systems are control and monitoring systems in aircraft, process control systems in chemical and pharmaceutical plants and automobile control systems.

2 Hardware control of safety-critical systems is simpler to implement and analyse than software control. However, we now build systems of such complexity that they cannot be controlled by hardware alone. Some software control is essential because of the need to manage large numbers of sensors and actuators with complex control laws. An example of such complexity is found in advanced, aerodynamically unstable military aircraft.

3 Safety-critical falls in to two classes

1 Primary, safety-critical software -This is software that is embedded as a controller in a system. Malfunctioning of such software can cause a hardware malfunction, which results in human injury or environmental damage. I focus on this type of software.

2 Secondary safety-critical software This is software that can indirectly result in injury. Examples of such systems are computer-aided engineering design systems whose malfunctioning might result in a design fault in the object being designed. This fault may cause injury to people if the designed system malfunctions.

4 System reliability and system safety are related but separate dependability attributes. Of course, a safety-critical system should be reliable in that it should conform to its specification and operate without failures.

5 There are several other reasons why software systems that are reliable are not necessarily safe

5.1 The specification may be incomplete in that it does not describe the required behaviour of the system in some critical situations. A high percentage of system malfunctions are the result of specification rather than design errors.

5.2 Hardware malfunctions may cause the system to behave in an unpredictable way and may present the software with an unanticipated environment. When components are close to failure they may behave erratically and generate signals that are outside the ranges that can be handled by the software.

5.3 The system operators may generate inputs that are not individually incorrect but which, in some situations, can lead to a system malfunction. The software carried out the mechanic's instruction perfectly. Unfortunately, the plane was on the ground at the time-dearly, the system should have disallowed the command unless the plane was in the air.

6 The key to assuring safety is to ensure either that accidents do not occur or that the consequences of an accident are minimal. This can be achieved in three complementary ways

6.1 Hazard avoidance The system is designed so that hazards are avoided. For example, a cutting system that requires the operator to press two separate buttons at the same time to operate the machine avoids the hazard of the operator's hands being in the blade pathway.

6.2 Hazard detection and removal- The system is designed so that hazards are detected and removed before they result in an accident. For example, a chemical plant system may detect excessive pressure and open a relief valve to reduce the pressure before an explosion occur.

6.3 Damage limitation -The system may include protection features that minimise the damage that may result from an accident. For example, an aircraft engine normally includes automatic fire extinguishers. If a fire occurs, it can often be controlled before it poses a threat to the aircraft.

7 It is impossible to make a system 100% safe, and society has to decide whether or not the consequences of an occasional accident are worth the benefits that come from the use of advanced technologies.

8 Table 2 discusses different safety terminologies

Sr.No	Term	Description
1	Accident	An unplanned event or sequence of events which results In human death or Injury, damage to property or to the environment. A computer-controlled machine injuring Its operator is an example of an accident.
2	Hazard	A condition with the potential for causing or contributing to an accident. A failure of the sensor that detects an obstacle In front of a machine Is an example of a hazard.
3	Damage	A measure of the loss resulting from a mishap. Damage can range from m21ny people killed as a result of an accident to minor Injury or property damage.

4	Hazard Severity	An assessment of the worst possible damage that could result from a particular hazard. Hazard severity can range from catastrophic where many people are killed to minor where only minor damage results.
5	Hazard Probability	The probability of the events occurring which create a hazard. Probability values tend to be arbitrary but range from probable (say 1/100 chance of a hazard occurring) to implausible (no conceivable situations are likely where the hazard could occur).
6	Risk	This is a measure of the probability that the system will cause an accident. The risk Is assessed by considering the hazard probability, the hazard severity and the probability that a hazard will result in an accident.

9 Security

9.1 Security is a system attribute that reflects the ability of the system to protect itself from external attacks that may be accidental or deliberate. Security has become increasingly important as more and more systems are connected to the Internet. Internet connections provide additional system functionality (e.g., customers may be able to access their bank accounts directly), but Internet connection also means that the system can be attacked by people with hostile intentions.

9.2 Examples of attacks might be viruses, unauthorised use of system services and unauthorised modification of the system or its data. Security is important for all critical systems. Without a reasonable level of security, the availability, reliability and safety of the systems may be comprised if external attacks cause some damage to the system.

9.3 The reason for this is that all methods for assuring availability, reliability and safety rely on the fact that the operational system is the same as the system that was originally installed. If this installed system has been compromised in some way (for example, if the software has been modified to include a virus), then the arguments for reliability and safety that were originally made can no longer hold.

9.4 There are three types of damage that may caused through external attack

1 Denial of service- The system may be forced into a state where its normal service, become unavailable. This, obviously, then affects the availability of the system.

2 Corruption of programs or data- The software components of the system may be altered in an unauthorised way. This may affect the system's behaviour and hence its reliability and safety. If damage is severe, the availability of the system may be affected.

3 Disclosure of confidential information -The information managed by the system may be confidential, and the external attack may expose this to unauthorised people. Depending on the type of data, this could affect the safety of the system and may allow later attacks that affect the system availability or reliability.

9.5 There are comparable approaches that may be used to assure the security of a system

1 Vulnerability avoidance -The system is designed so that vulnerabilities do not occur. For example, if a system is not connected to an external public network then there is no possibility of an attack from members of the public.

2 Attack detection and neutralisation- The system is designed to detect vulnerabilities and remove them before they result in an exposure. An example of vulnerability detection and removal is the use of a virus checker that analyses incoming files for viruses and modifies these files to remove the virus.

9.6 Table 3 discuses some security terminologies

Sr.No	Term	Description
1	Exposure	Possible loss in a computing system. This can be loss or damage to data or can be a loss of time and effort if recovery is necessary after a security breach.
2	Vulnerability	A weakness in a computer-based system that may be exploited to cause loss or harm.
3	Attack	An exploitation of a system's vulnerability. Generally, this is from outside the system and is a deliberate attempt to cause some damage.
4	Threats	Circumstances that have potential to cause loss or harm. You can think of these as a system vulnerability that is subjected to an attack.
5	Control	A protective measure that reduces a system's vulnerability. Encryption would be an example of a control that reduced a vulnerability of a weak access control system.

Questions

Q1 Explain the four dimensions of Dependability?

Q2 What is Safety Critical Systems?

Q3 What are the three principal types of critical system? Explain the differences between these?

Q4 Suggest six reasons why dependability is important in critical systems.

Q5 Giving reasons for your answer, suggest which dependability attributes are likely to be most critical for the following systems: • An Internet server provided by an ISP with thousands of customers • A computer-controlled scalpel used in keyhole surgery • A directional control system used in a satellite launch vehicle • An Internet-based personal finance management system.

❖ ❖ ❖

UNIT 2

4

SOFTWARE PROCESSES

Unit Structure:

4.0 Objectives

After going through this chapter, you will learn about:

- Distinguish and comprehend the various software process models that have developed.
- Advantages and drawbacks of each process model
- realize why processes should be structured to cope with changes in the software requirements and design.
- understand how the Rational Unified Process incorporates good software engineering practice to create compliant software processes.

4.1 Introduction

A software process is a set of related activities that leads to the creation of a software product.

When we speak about and explain processes, we typically refer to the activities that take place during them, such as defining a data model, designing a user interface, and so on, as well as the order in which these activities take place. Method definitions, in addition to operations, can also include:

1. **Products,** which are the consequences of a process activity. For example, the conclusion of the activity of architectural design may be a model of the software building.
2. **Roles**, which reflect the duties of the people involved in the process. Examples of responsibilities are project manager, configuration manager etc.
3. **Pre- and post-conditions,** which are declarations that are true before and after a process activity has been performed or a product produced. For example, before architectural design begins, a pre-condition may be that all needs have been approved by the customer; after this activity is finished, a post-condition might be that the UML models depicting the architecture have been examined.

   Process standardisation, which reduces the diversity of software processes within an enterprise, may strengthen software processes. This improves coordination and reduces training time, as well as making automated process support more cost-effective.

   Standardization is also a crucial first step in implementing modern software engineering approaches and techniques, as well as good software engineering practise.

4.2 Fundamental Activities of Software Process

There are many distinct software processes, but all must include four activities that are basic to software engineering:

1. **Software specification** the functionality of the software and restrictions on its function must be defined.
2. **Software design and implementation** the software to meet the requirement must be produced.
3. **Software validation** the software must be authenticated to ensure that it does what the customer wants.
4. **Software evolution** the software must evolve to meet switching customer needs

4.3 Different Software Process Models

A software process model is a representation of a software process that is abstract. Several general process models are introduced in this section, and they are viewed from an architectural standpoint.

These models may be used to describe various software development approaches. They can be thought of as process models that can be customised to build more unique software engineering processes.

In this chapter the following process models will be introduced:

1. **The waterfall model**- In this model of software process the essential process activities of specification, development, validation, and evolution are represented as chronological process phases such as requirements specification, software design, implementation, testing and so on.

2. **Evolutionary development**. This approach incorporates the activities of specification, development, and validation. An initial system is rapidly created from abstract specifications. Then the initial system is developed by customer inputs to produce a system that fulfils the customer's needs.

3. **Component-based software engineering** also called as reuse oriented software engineering the process models that use this approach are based on the survival of a considerable number of reusable components. The system development process focuses on combining these components into a system rather than developing them.

1.3.1 Waterfall model

The first published model of the software development process was stemmed from more general system engineering processes. Because of the flow from one phase to another, this model is known as the 'waterfall model' or software life cycle.
The waterfall model is an example of a plan-driven process—in principle, you must plan and schedule all the development activities before starting work on them.

The main stages of the waterfall model directly signify the fundamental development activities:

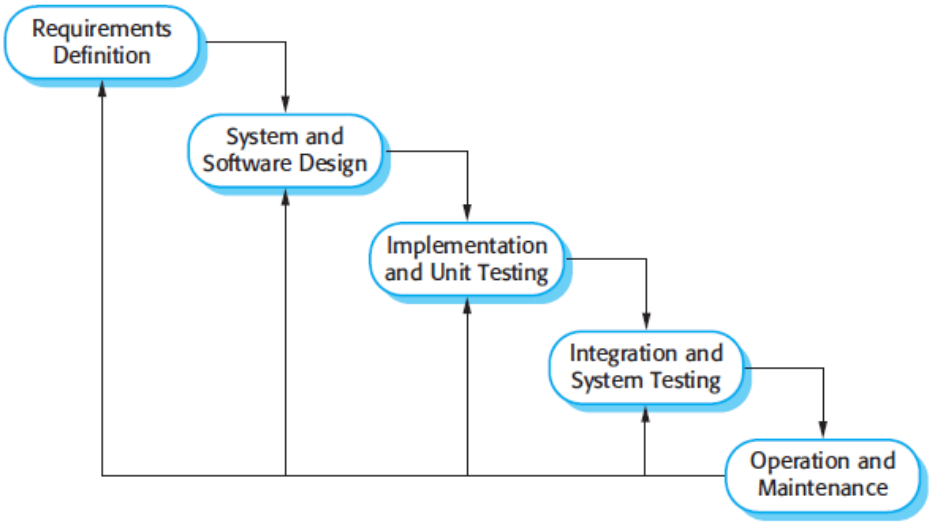
1. **Requirement's analysis and definition**
   the system's services, limitations, and goals are established by meeting with system users. They are then defined in detail and serve as a system specification.
2. **System and software design**

the systems design process distributes the requirements to either hardware or software systems by establishing an overall system architecture. Software design includes recognizing and describing the fundamental software system concepts and their interactions.

3. **Implementation and unit testing**
   During this stage, the software design is understood as a set of programs or program units. Unit testing means verifying that each unit meets its specification.

4. **Integration and system testing**
   The individual program units or programs are unified and verified as a complete system to ensure that the software requirements have been convened. After testing, the software system is brought to the customer.

5. **Operation and maintenance**
   The system is fitted and put into real use. Maintenance entails rectifying errors which were not discovered in prior stages of the life cycle, improving the completion of system units, and improving the system's services as new needs are discovered.



The waterfall model is stable with other engineering process models and documents are produced at each phase. This makes the process evident so managers can monitor progress against the advancement plan. Its major problem is the inflexible dividing of the project into different stages. Pledges must be made at an early stage in the process, which becomes difficult to react to altering customer requests.

1.3.2 Evolutionary development

Evolutionary development is based on the concept of creating an initial implementation, submitting it to user feedback, and improving it through several versions until a suitable system is created.

The activities of specification, creation, and validation are interspersed with rapid feedback between them.

Evolutionary growth can be divided into two categories:

1. **Exploratory development**

   The process's goal is to collaborate with the consumer to understand their needs and produce a finished product.

   The construction of the system begins with the sections that are well understood.

   The framework changes as the customer suggests new functionality.

2. **Throwaway prototyping**

   In this case, the evolutionary development process' goal is to comprehend the customer's ambiguous requirements, i.e., to validate and extract the system's requirements concept.

   The prototype is based on experimenting with poorly understood customer requirements.

   When it comes to creating applications that satisfy customers' immediate needs, an evolutionary approach to software development is often more successful than a waterfall approach.

   The benefit of an evolutionary software development process is that the specification can be created incrementally.

   As users gain a deeper understanding of their dilemma, the software framework may represent this. The evolutionary method, on the other hand, has flaws. For tracking progress, regular deliverables are needed.

   When systems are built rapidly, producing documents that represent every iteration of the system is not cost-effective.

   Repetitive change tends to corrupt the software composition.

1.3.3 Reuse-oriented software engineering

Reusable software components and an integrating structure for their composition are the foundations of reuse-oriented approaches. These components may include systems that perform specific functions, such as word processing or spreadsheets.

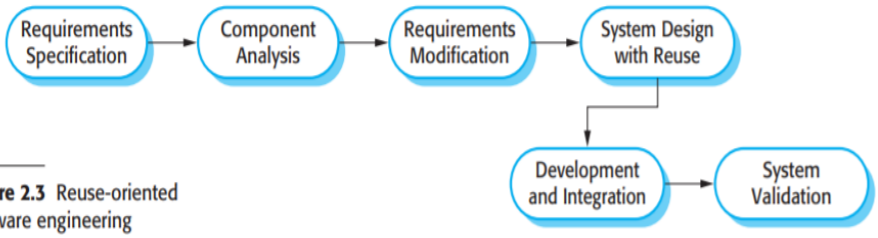


Figure 2.3 Reuse-oriented software engineering

Though the preliminary requirements specification stage and the validation stage are analogous with other software processes, the transitional stages in a reuse-oriented process are different. These stages are:

1. **Component analysis** A quest for components to implement the specifications specification is conducted based on the requirements specification. In most cases, there is no exact match, and the modules that can be used only include a portion of the necessary features.

2. **Requirement's modification**. Using the information about the components that have been identified, the specifications are analysed at this point. They are then tweaked to reflect the components that are accessible. If changes are not necessary, the component review operation can be re-entered to look for other options.

3. **System design with reuse** the system's architecture is designed, or an existing framework is reused, during this process. The designers consider the components that are reused and structure the system accordingly. If reusable components are not available, new software can have to be created.

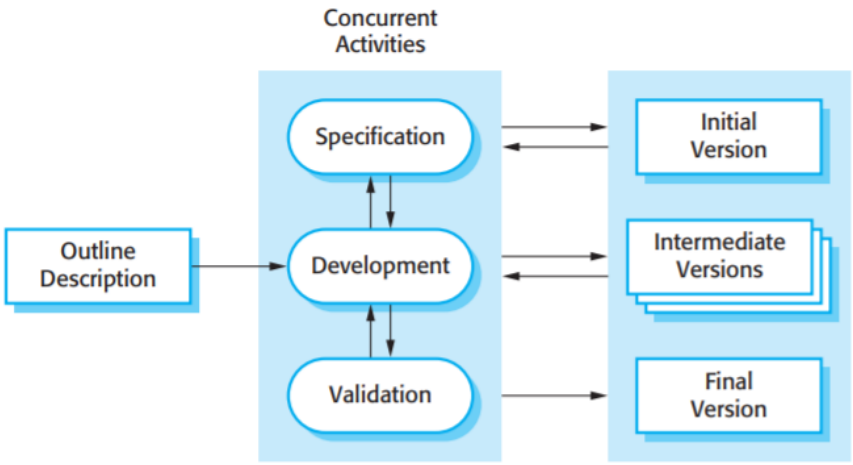
3. **Development and integration** The components and COTS systems are combined to build the new system, and software that cannot be procured externally is created. In this model, system integration may be a part of the development process rather than a separate operation.

   The obvious benefit of reuse-oriented software engineering is that it reduces the amount of software that needs to be built, lowering costs and risks. It normally also means that the programme is delivered faster. However, specifications compromises are unavoidable, and this could result in a system that fails to meet users' actual needs.
   Furthermore, when new iterations of recycled parts are not under the control of the company that uses them, some control over system evolution is lost.

4.4 Process iteration

4.4.1 Incremental Development

Incremental development is based on the idea of creating an initial implementation, revealing this to user comment and evolving it through several editions until a sufficient system has been created.



Incremental development has three important benefits :

1. the value of accommodating changing customer requirements is reduced. the quantity of study and documentation that has got to be redone is far but is required with the waterfall model.
2. it's easier to urge customer feedback on the event work that has been done. Customers can discuss demonstrations of the software and see what proportion has been implemented. Customers find it difficult to gauge progress from software design documents.
3. More rapid delivery and deployment of useful software to the customer is feasible, albeit all the functionality has not been included.

Customers can use and gain value from the software before is feasible with a waterfall process.

The incremental approach has two problems:

1. the method is not visible. Managers need regular deliverables to live progress. If systems are developed quickly, it is not cost-effective to supply documents that reflect every version of the system.
2. System structure tends to worsen as new increments are added. Unless time and money are spent on refactoring to enhance the software, regular change tends to corrupt its structure.

Integrating further software changes becomes progressively more difficult and expensive.

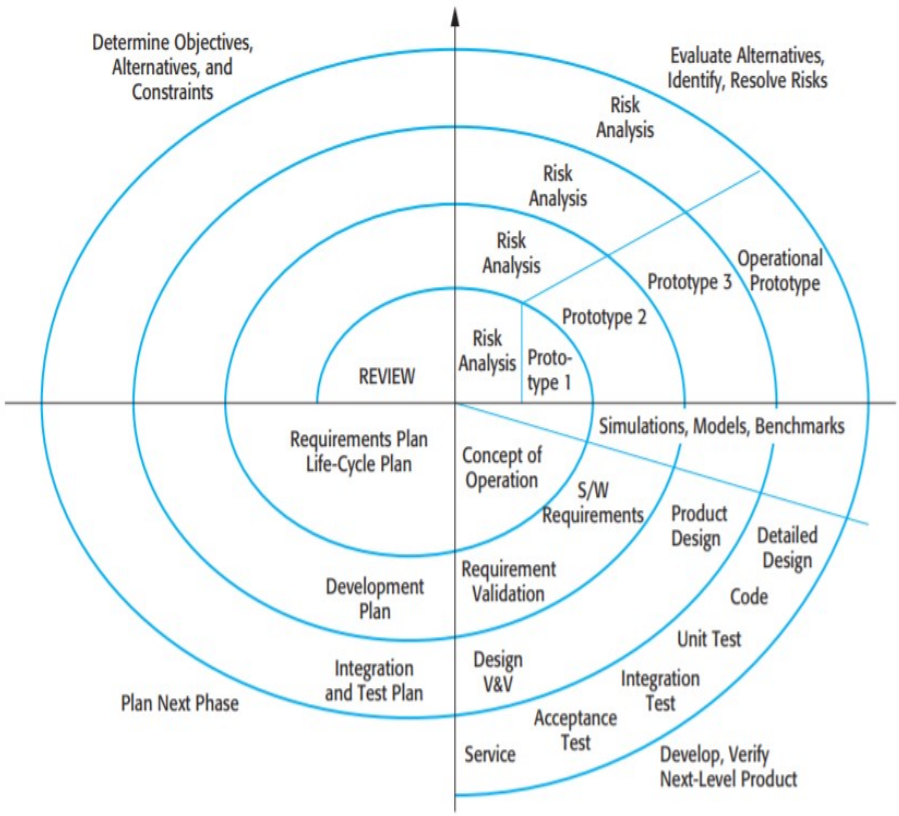
The problems of incremental development become particularly acute for giant, complex, long-lifetime systems, where different teams develop different parts of the system.

Large systems need a stable framework or architecture and therefore the responsibilities of the various teams performing on parts of the system got to be clearly defined with reference to that architecture. This must be planned as opposed to developed incrementally.

4.4.2 Spiral Model

A risk-driven software process framework was suggested by Boehm. The software process is characterized as a spiral, rather than a sequence of activities with some turn back from one activity to another. Every loop in the spiral signifies a phase of the software process.

Thus, the innermost loop might be involved with system feasibility, the following loop with requirements definition, the later loop with system design, and so on. The spiral model is combining change avoidance with change acceptance.



Each loop in the spiral is split into four sectors:

1. **Objective** setting Specific intentions for that phase of the project are defined. Limitations on the process and the product are identified and a detailed management plan is drawn up. Project risks are found. Option strategies, varying on these risks, may be planned.

2. **Risk assessment and reduction** for each of the identified project risks, a detailed analysis is carried out. Steps are taken to reduce the risk. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

3. **Development and validation** After risk assessment, a development model for the system is chosen. For example, throwaway prototyping may be the best growth approach if user interface risks are dominant. If safety risks are the key consideration, expansion based on formal transformations may be the most appropriate process, and so on.
   If the main identified risk is sub-system combination, the waterfall model may be the best growth model to use.

4. **Planning** The project will be reviewed, and a decision made whether to proceed with a further loop of the spiral. If it is decided to proceed further, plans are drawn up for the next stage of the development.

Process Activities

The four process activities of specification, development, validation, and evolution are structured differently in various development processes.

4.4.3 Software specification

Software specification or requirements engineering is the process of comprehension and defining what services are necessary from the system and identifying the limitations on the system's operation and development.

There are four major activities in the requirements engineering process:

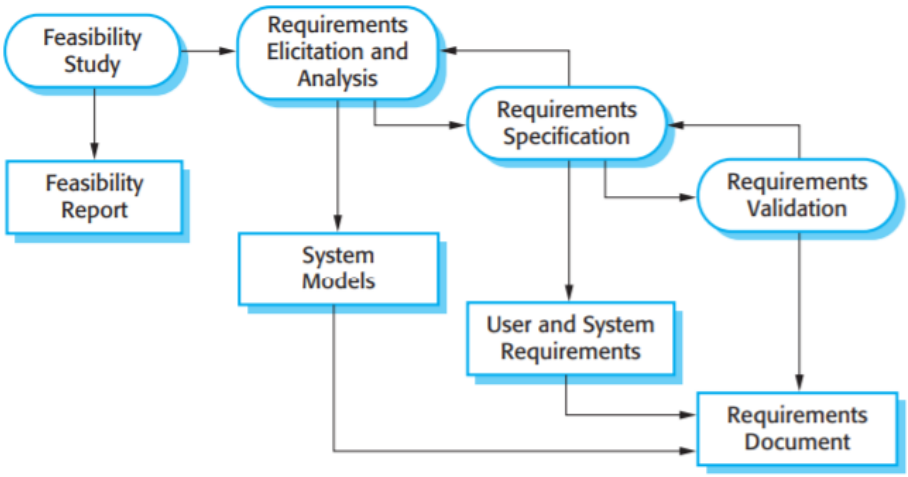
1. **Feasibility study**
   The defined user needs are assessed to see whether they can be met with current software and hardware technologies.

The analysis examines whether the proposed system is cost-effective from a business perspective and whether it can be developed under current budgetary constraints.

A feasibility study should be inexpensive and easy to complete. The outcome should help you decide whether or not you want to pursue a more in-depth investigation.

4.2 **Requirement's elicitation and analysis**

This is the method of determining device specifications by looking at current programmes, talking to potential users and procurers, doing task analysis, and so on. This could include creating one or more device models and prototypes. These aid in the comprehension of the method to be specified.



3. **Requirement's specification**:

Requirement's specification is the activity of translating the information collected during the analysis activities carried out into a document that defines a set of prerequisites.

Two types of requirements can be included in this document. User requirements are abstract declarations of the system requirements for the customer and end-user of the system; system requirements are a more precise description of the features that are to be provided.

4. **Requirement's validation**

This activity checks the conditions necessary for realism, consistency, and completeness. During this process, errors in the conditions set out in the document are inevitably discovered. It must then be altered to correct these problems.

4.4.4. Software design and implementation

A software design is a detailed description of the structure of the software to be carried out, the data models and structures used by the system, the interface between the system components and, sometimes, the algorithms used.

Activities carried out in the design process vary, depending on the type of system being developed.

1. **Architectural design**, where you recognize the overall structure of the system, the major components their relationships, and how they are distributed.

2. **Interface design**, where you define the boundaries between system components. This interface design must be unambiguous. With a precise interface, a component can be used without other elements having to know how it is implemented. Once boundary specifications are agreed, the components can be designed and developed concurrently.

4. **Component design**, where you take each part of the system and design how it will work. This may be as basic as a statement of the intended features to be implemented, with the programmer responsible for the design. It may also be a list of modifications to make to a reusable part or a comprehensive concept model. The architecture model can be used to create an implementation automatically.

5. **Database design,** where you build the data structures of the system and how they will be interpreted in a database Whether an existing database is to be reused or a new database is to be built, the work may vary.

4.4.5 Software validation

Software validation, or verification and validation in general, is used to demonstrate that a device meets its specifications and the needs of the consumer who purchased it.

It entails inspecting the processes at each level of the software development cycle. Most validation costs are incurred after the system has been implemented and reviewed.

The programme is put through its paces in a three-stage process.

The components of the system, the integrated system, and then the whole system are all checked. Component flaws are typically discovered early in the process, while interface issues are discovered during device integration.

1. **Component testing-** Specific components are tested to ensure that they operate correctly. Each component is tested separately, without other system components.

2. **System testing-** the components have been integrated to make up the system. This testing process is worried with finding errors that result from interactions between components and component interface difficulties. It is also concerned with confirming that the system meets its functional and non-functional requirements.
3. **Acceptance testing-** It is considered a functional testing of system. The system is tested with data provided by the system client.

Component development and testing are usually done concurrently. Programmers create their own test data and run it against the code as it is written. However, in many process models, such as the V-model, Test Driven Development, and Extreme Programming, the design of test cases begins before the development phase.
If you're using an incremental development strategy, each increment should be evaluated as it's built, with the tests centred on the criteria for that increment.

4.4.5 Software Evolution

Software is designed to be adaptable and changeable. The software that serves the company must develop and adapt as the requirements change due to changing business circumstances. While there has been a distinction made between creation and evolution as less and less structures are entirely new, this distinction is becoming increasingly meaningless.

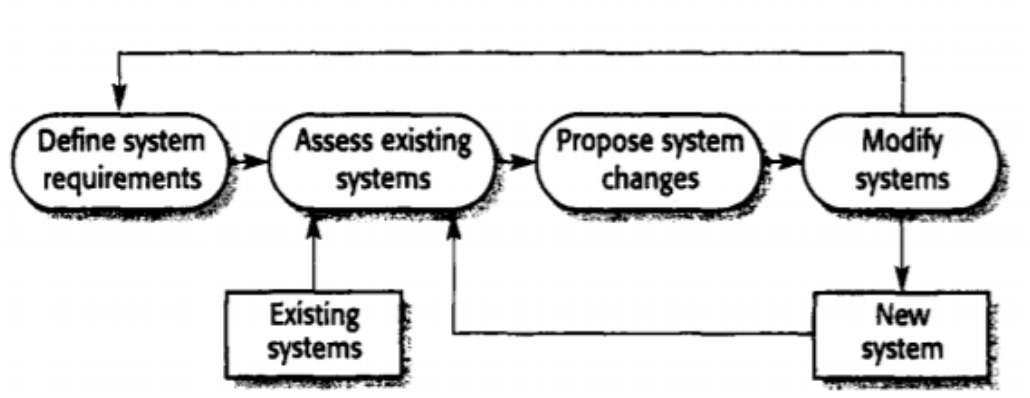


Figure Reference: Sommerville 8e Chapter 4

4.5 The Rational Unified Process

The Rational Unified Process is an illustration of a modern process model that has been developed from work on the UML and the related Unified Software Development Process.

The RUP is usually described from three perspectives:

1. A dynamic perspective, which shows the stages of the prototype over time.
2. A static perspective, which illustrates the process activities that are presented.
3. A practice perspective, which suggests good practices to be used during the process.

The RUP is a phased model which identifies four discrete phases in the software process. However, unlike the waterfall model where phases are associated with process activities, the phases in the RUP are more closely linked to the business rather than technical issues.

These are as follows:

Inception

The inception phase's aim is to create a business case for the system. You should recognise and describe all external entities (people and systems) that will communicate with the system. The information is then used to evaluate the system's contribution to the market. If this contribution is insignificant, the project will be terminated at this stage.

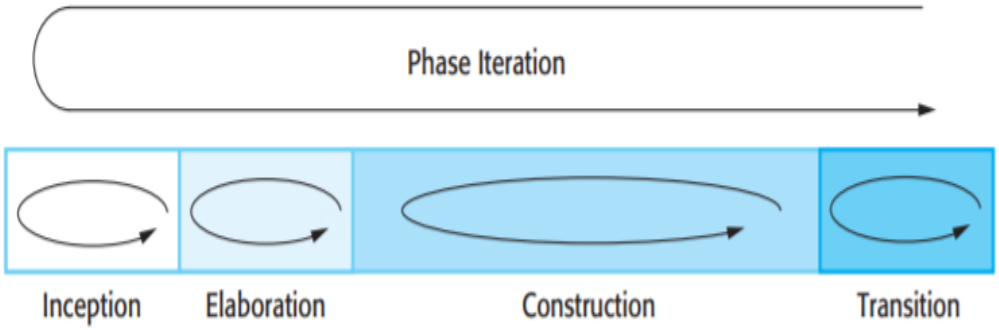


Figure Reference: Sommerville 8e Chapter 4

Elaboration

The elaboration phase's objectives are to gain a better understanding of the problem domain, create a system architectural context, create a project plan, and define key project risks. After completing this process, you should have a system requirements model, which may be a collection of UML use-cases, an architectural overview, and a software development plan.

Construction

Device design, programming, and testing are all part of the construction process. During this process, different parts of the system are built in parallel and then integrated. You should have a functioning software system and related documentation ready to offer to users at the end of this process.

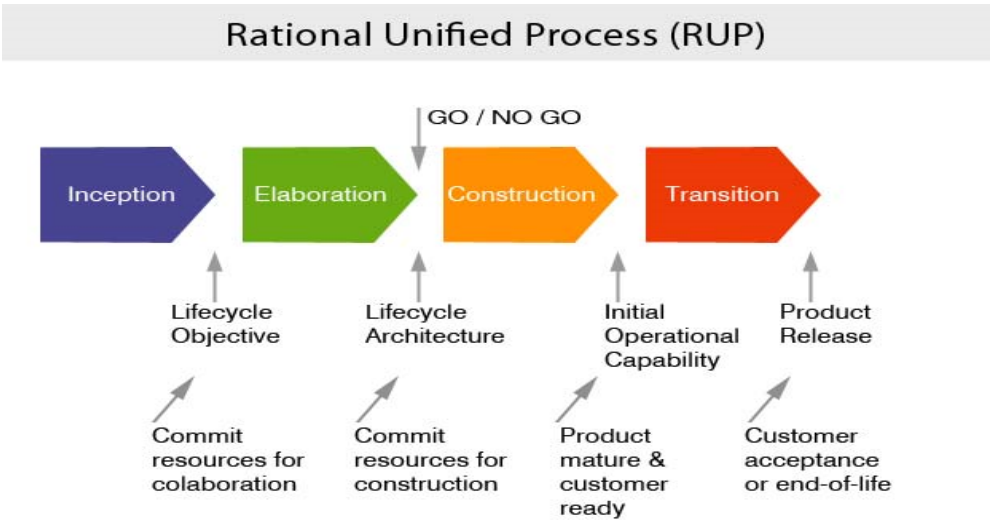


Fig Reference: https://infostride.com/what-is-rup/

Transition

The RUP's final step focuses on transferring the technology from the development community to the consumer community and putting it to use in a real-world setting. This is something that most software process models overlook, but it is a costly and sometimes troublesome operation.

You should have a recorded software system that is operating correctly in its operational environment at the end of this process. The RUP from a practise perspective defines good software engineering practises that can be used in system development.

Six fundamental best practices recommended are:

1. **Develop software iteratively.** Plan increases of the system based on customer priorities as well as develop and deliver the highest priority system characteristics early in the development process.

2. **Manage requirements**. Unambiguously document the customer's requirements and keep track of modifications to these requirements. Examine the impact of changes on the system before you accept them.

3. **Use component-based architectures.** Structure the system structure into components.

4. **Visually model software**. Use graphical UML models to present static and energetic views of the software.

5. **Verify software quality.** Ensure that the software meets the organizational quality standards.

6. **Control changes to software.** Manage changes to the software using a transformation management system and configuration management procedures and instruments.

The RUP is not a universally applicable method, but it does reflect a new generation of generic processes. The differentiation of phases and workflows, as well as the understanding that installing applications in a user's environment is part of the process, are the most significant developments.

Phases are complex and have specific objectives. Workflows are static technological activities that are not associated with a single step but can be used to accomplish the goals of each phase in the development process.

CASE

Computer-Aided Software Engineering (CASE) is the name given to software that is used to support software process as well as activities like requirements engineering, design, program development and testing. CASE tools therefore include design editors, data vocabularies, compilers, debuggers, system building tools and so forth. CASE technology offers software process support by how to automate some process activities and by supplying information about the software that is being developed.

The set of application programs to mechanise software development lifecycle activities and are used by managers in a project, engineers, and experts to shape a software system is called CASE tools and the software development cycle stages can be abridged using several tools such as design, analysis, project management, database management, papers, etc. and the use of these tools speeds up the project development to achieve the desired results.

CASE Tools offer an exceptional array of features that promote the development and business community through its Automated Diagram Support feature. The different popular features that contribute to the development community are listed below:

- Checks for syntactical correctness.
- Data dictionary support
- Examinations for consistency and fullness.
- Routing to linked graphs.
- Layering
- Requirement's tracking
- Automatic generating reports
- System modelling
- Performance assessment

CASE classification

CASE classifications help us to understand the types of CASE tools and their role to play in supporting software process activities. There are several ways to categorize the CASE tools, each of which provides us a different view on these tools.

CASE tools from three of these perspectives:

1. Afunctional perspective in cases in which CASE tools are categorized depending on their specific function.
2. A process perspective place in which tools are classified in accordance with the procedure activities that they support.

3. An integration standpoint where CASE tools are classified according to exactly how they are organized in the integrated units that provide support for one or several process activities.

Functional Classification Tools:

Tool types	**Examples**
Planning tools	PERT tools, assessment tools, spreadsheets
Editing tools	Text editors, map editors, word processing
Change management tools	Requirement's tracking tools, change self-control systems
the administration tools	Version control systems, system building tools
Prototype Development tools	Very high-level linguistic, user experience generators
Program analysis tools	Cross-reference generators, static analysis equipment, dynamic analyzers
Documentation tools	Page design programs, picture-editing
Restructuring tools	Cross-reference systems, package restructuring systems

Benefits of CASE

Cost savings are a critical advantage of a CASE environment at all stages of growth.

- Using CASE tools contributes to a significant increase in efficiency.
- The chances of human error are greatly decreased at various stages of software development.
- CASE resources aid in the creation of high-quality, consistent documents.
- CASE tools automate the majority of a software engineer's tasks.

Takeaways:

- General process simulations describe the organization of the application processes. Examples of these common examples include the waterfall model, incremental development, and reuse-oriented development.
- The tasks that go into creating a software system are referred to as software processes. Abstract representations of these processes are known as software process models.
- The process of creating a software specification is known as requirements engineering. The aim of specifications is to convey the customer's system requirements to the system developers.
- The transformation of a specifications specification into an executable software system is the focus of the design and implementation processes. As part of this transformation, systematic design approaches can be used.
- Software validation is the process of verifying that the system conforms to his specification and that it meets the actual needs of the customers of the system.
- Software evolution takes place when you modify existing software systems to meet the new requirements. Changes are ongoing and the software must grow to remain useful.
- Processes should include activities designed to cope with change. This may involve a prototyping phase that will help prevent poor decisions on requirements and design. Processes can be structured for iterative development and provision so that changes may be made without interrupting the system as one.
- The Rational Unified Process is a modern-day generic process model which is organized into phases-inception, elaboration, construction, and transition but distinguishes between activities (requirements, analysis, and design, etc.) beginning in these phases.

References:

Boehm, B. and Turner, R. (2003). Balancing Agility and Discipline: A Guide for the Perplexed. Boston: Addison-Wesley.

Rettig, M. (1994). 'Practical Programmer: Prototyping for Tiny Fingers'. Comm. ACM, 37 (4), 21–7.

Royce, W. W. (1970). 'Managing the Development of Large Software Systems: Concepts and Techniques'. IEEE WESTCON, Los Angeles CA: 1–9.

❖ ❖ ❖

UNIT 2

5

PROJECT MANAGEMENT

5.0 Objectives

- know the principal responsibilities of software project managers
- have been introduced to the notion of risk-management and some of the risks that can arise in software projects
- comprehend the factors that influence personal motivation and what these may possibly mean for software project managers.
- appreciate key issues that have an impact on the team working, such as squad composition, organization, and communication.

Software project management is an essential component of software engineering. Projects need to be managed because professional software engineering is constantly subject to administrative budget and schedule the restrictions.

The success criteria for project management apparently vary from project to project but, for the most part projects, important goals are:

1. Deliver the software to the client at the agreed time.
2. Keep overall expenses within budget.
3. Deliver software which meets the client's expectations.
4. Maintain a happy and very well-functioning developing the team.

5.1 Risk Management

Risk management is one of the most important job opportunities for a project manager. Risk management involves predicting risks that might affect the timetable for the project or the quality of the software being established, and then acting to prevent these risks.

Three related types of risk:

1. Project risks

Risks that influence the project schedule or resources. An example of a development risk is the loss of an experienced designer. Discovering a replacement designer with appropriate knowledge and experience may take a long time and, therefore, the software design will take a long time to complete.

2. Product risks

Risks that influence the quality or performance of the software being created. An example of a product risk is the failure of a purchased component to work as expected. This may affect the overall performance of the system in such a way that it is slower than expected.

3. Business risks

Risks that affect the company expanding or procuring the software. For example, the competitor introducing a new product is a corporate risk. The introduction of a competitive product may mean that the underlying

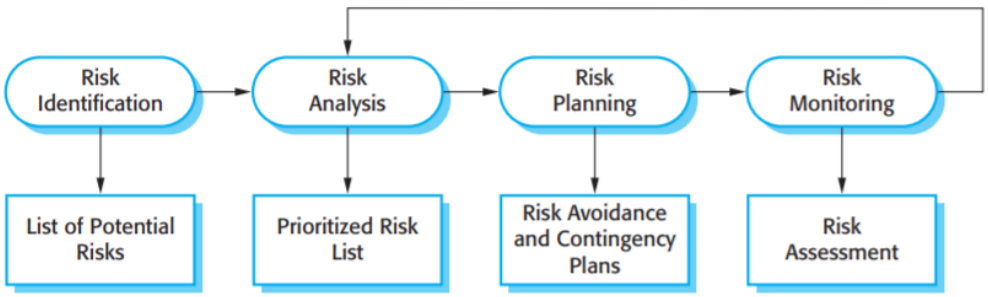
assumptions made about sales of existing software products might be unduly optimistic.

The stages involved in risk management are:

1. Risk identification You need to identify possible project, produce, and business risks.
2. Risk analysis You should assess the probability and consequences of these risks.
3. Risk planning You should make proposals to address the risk, either by avoiding it or else minimizing its impact on the project.
4. Risk monitoring You should regularly evaluate the risk and your plans for probability mitigation and modify these when you learn more about the risk.

5.1.1 Risk Management Process

The risk management process is the iterative process that is progressing throughout the entire project. Once you have been prepared in accordance with an initial risk management plan, you are the ability to monitor the problem to detect emerging dangers.



5.1.1.1 Risk Identification:

Risk identification is the initial phase of the risk-management process. It is worried about pinpointing the risks that could constitute a serious threat to the software engineering process, the computer software being developed, or development within the enterprise. Hazard identification may be a team process where a team get together to brainstorm the potential risks.

There are at least six kinds of risk that might be included in a risk checklist:

1. **Technology risks**

   Risks that are derived from the software or hardware technologies that will be used to develop the system.

2. **People risks**

   Risks that are linked to the people in the development team.

3. **Organizational risks**

   Risks that derive from the administrative environment where third-party software is being developed.

4. **Tools risks**

   Risks that are derived from the software tools and other support software utilized to develop the system.

5. **Requirements risks**

   Risks that stem from the changes to the customer requirements and the whole process of managing the requirements transformation.

6. **Estimation risks**

   Risks that are derived from the administration estimates of the resources needed to construct the system.

Examples of various types of risk:

Technology	• The database used in the system could not process as many transactions per second as expected. • Reusable software components include the following defects that mean they cannot be reused as planned.
People	• It is impossible to recruit employees with the skills required. • Key staff are ill and unavailable at critical moments. • Required training for staff is no longer available.
Administrative	• The organization remains restructured so that different management have been responsible for the project. • Organizational financial problems come into effect reductions in the budget for the project.
Tools	• The code that is generated by the software code generation tools is ineffective.

	• Software tools cannot go to work together in an integrated way.
Needs	• Changes to the necessary conditions that require major design rework are proposed. • Customers fail to understand the impact of requirements changes.
Estimation	• The time necessary to develop the software will be underestimated. • The rate of defect overhaul is underestimated. • The size of the software has been miscalculated.

5.1.1.2 Risk Analysis

During the risk analysis process, you have to take into account each identified risk and make a judgment about the probability and seriousness of that risk. It is not possible to make precise, a number that represents the assessment of the probability and significance of each risk.

Rather, you should assign the risk to one of several bands:

1. The probability of the risk might be assessed as very low (10%), low (10–25%), moderate (25–50%), high (50–75%), or very high (> 75%).
2. The effects of the risk might be assessed as a disastrous, serious (would cause major delays), acceptable (delays are within allowed contingency), or inconsequential.

Examples for Risk Analysis

Risk	**Probability**	**Effects**
Organizational financial problems come into effect reductions in the budget for the project	Low	Catastrophic
Key staff are ill at critical points in the project	High	Catastrophic
The time required to develop the software is so that you can underestimate the	High	Serious
The size of the computer software has underrated	High	Tolerable

The database used in the system cannot process as many trades per second as expected	Moderate	Serious
Faults in recyclable application components must be repaired before those components are reused	Moderate	Serious

5.1.1.3 Risk Planning

The risk planning process considers each of the key risks that have been identified and has been developing strategies to manage such risks. For each of the risks, you must think of actions that you might take to minimize the disruption to the project if the problem found in the risk occurs there is no simple process that can be followed for contingency planning.

Risk	**Strategy**
Administrative financial problems	Prepare a briefing document for senior management showing how the project is getting a very important contribution to the goals of the business and introducing reasons why cuts to the project's budget would not be cost-effective
Requirements changes	Derive tracking information to assess requirements change impact. maximize concealing information in the design.
Organizational restructuring	Prepare a briefing paper for senior management showing how the project is making a very significant contribution to the goals of the business.
Database performance	Study the possibility of buying a higher-performing database.
Misjudged development time	Investigate buying-in component parts; inquiry into the use of a program generator.

It relies on the decision and experience of the project manager. possible risk management policies which have been identified for the key risks shown in. These policies are divided into three categories:

Avoidance strategies Following these policies means that the probability that the risk will emerge will be reduced. An example of a risk prevention strategy is the strategy to deal with defective components.

Minimization strategies Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is the strategy for staff illness.

Contingency plans Following these tactics means that you are ready for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial difficulties.

5.1.1.4 Risk Monitoring:

Risk monitoring is the whole process of checking that your beliefs about the product, process, and business risks have not changed.
You should periodically assess each of the identified risks to decide whether that risk is becoming probable. You should also think about whether the effects of the danger have changed. To do this, you must look at other considerations, such as the number of requirements alter requests, which give you clues about the risk likelihood and its effects.

5.2 Managing People

The people who work in a software organization are its biggest assets. It costs a lot to recruit and retain good people and it is up to software managers to ensure that the organization receives the best possible return on his investment. It is important that software project managers recognize the technical issues that influence the work of software engineering.

There are four key factors in people management:

1. **Consistency:** People in a project team should all be dealt with in a comparable way. No one expects all rewards to be the same, but people should not feel that their contribution towards the organization is undervalued.

2. **Respect:** Different people have different abilities and managers must respect those differences. All members of the team should be given the chance to contribute. In some cases, of course, you will find that people just do not fit into a team and they cannot continue, but it is important not to jump to conclusions about this in the early stages in the project.

3. **Inclusion:** People contribute efficiently when they feel that others listen to them and take account of their proposals. It is important to develop a functional environment where all views, even the ones that of the most junior staff, are taken into consideration.

4. **Honesty:** As a supervisor, you must always be honest about what is going well and what is going badly in the team. You also need to be honest about your level of technical expertise and willing to defer to staff with more knowledge when necessary. If you try to cover up ignorance or difficulties, you will eventually be discovered out and will lose the respect of the group.

5.2.1 Motivating People

Motivation means coordinating the work and the working environment to encourage people to work as effectively as possible. If people are not motivated, they will not have an interest in the work they are doing. They will be working slowly, be more likely to make mistakes, and would not contribute to the broader goals of the team or the organization.

People working in software development organizations are usually not hungry or physically threatened by their environment. Therefore, making sure that people's social, appreciation, and self-realization needs are fulfilled is most important from the management point of view.

1. To satisfy **social needs,** you must give people time to meet their co-workers and deliver places for them to meet. This is relatively straightforward when all of the members of a developing team work in the same place but, increasingly, group members are not housed in the same building or even the same town or state.

They may work for the various organizations or from home most of the time. Social networking systems and teleconferencing may be used to facilitate the messages but my experience with electronic systems is that they are the most effective once people know each other. You therefore will be necessary to arrange some face-to-face meetings early in the project so that people can directly communicate with other members of the team. Through this direct interaction, people become an integral part of a social group and accept the objectives and priorities of that group.

2. To satisfy **esteem needs,** you need to show people that they are valued by the organization. Public recognition of achievements is a simple yet effective way of doing this. Obviously, people must also feel that they are paid at a level that reflects their skills and experience.

3. Finally, to satisfy **self-realization needs,** you must give people responsibility for their work, to allocate them demanding tasks, and provide a training programme where people may develop their skills. Training is an important encouraging influence as people like to gain new knowledge and learn new skills.

Personality type also influences motivation there are three types of personality types:

1. **Task-oriented people**, who are motivated by the work they are doing. In software engineering, these are motivated by the scholarly challenge of software development.

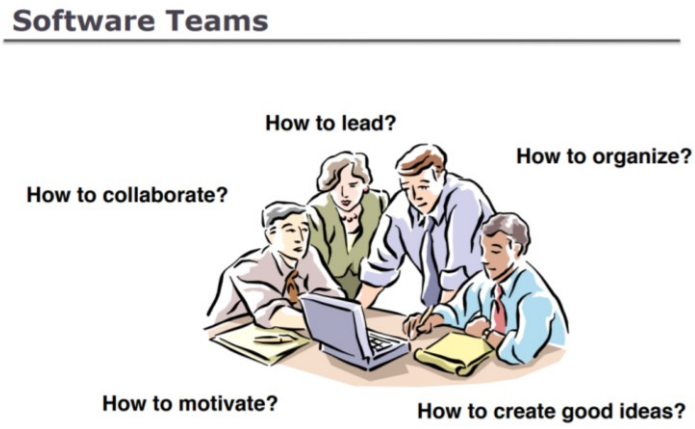
2. **Self-oriented people**, who are principally inspired by personal success and recognition. They are interested in software development as a means of accomplishing their own goals. This does not mean that these people are egotistical and think only of their own concerns. Rather, they frequently have longer-term goals, such as career progression, that motivate them, and they wish to be a success in their work to help achieve these objectives.

3. **Interaction-oriented people**, who are motivated by the existence and actions of co-workers. As software development becomes more user-cantered, communication-oriented individuals are becoming more engaged in the software engineering.

Motivation Balance:

Individual motivations are made up of elements of each class. The balance can change depending on personal circumstances and external events. However, people are not just motivated by personal factors but also by being part of a group and culture. People go to work because they are motivated by the people that they work with.

5.3 Teamwork



Ref: https://cs.hac.ac.il/staff/solange/se/22-project-management.pdf

When small groups are being used, communication problems are reduced. Everybody knows everyone else and the whole group can get around a table for a meeting to discuss the project and the software that they are developing. In a cohesive group, participants think of the group as more important than the individuals who are group members. Members of a well-led, cohesive band are loyal to the group. They identify with the collective goals and other group members. They are trying to protect the group, as an entity, from external interference.

The advantages of forming a cohesive group are:

1. The group can set up its own quality standards Because these standards are set up by the consensus, they are much more likely to be observed than external requirements imposed on the group.

2. Individuals learn from and backing each other People in the group learn from each other. Inhibitions caused by ignorance are minimized as mutual learning is encouraged.

3. Knowledge is shared Continuity can be maintained if a group member leaves. Others in the group may take over critical tasks and ensure that the design is not unduly interrupted.

4. Refactoring and continual improvement is urged Group members work collectively to be able to provide high-quality results and resolve problems, irrespective of the persons who originally created the design or program.

   One of the most effective ways of supporting cohesion is to be inclusive. This means that you must treat group members as responsible and trusted and make information freely accessible.

Whether or not a group is effective depends, to some extent, on the nature of the development and the organization doing the work. If one organization is in a state of turmoil with continual reorganizations and employment insecurity, it is very difficult for team members to concentrate on the software development.

However, apart from project and organizational issues, there are three generic factors that affect team working:

1. The people in the group You need a combination of people in a project group as software development involves diverse activities like negotiating with clients, programming, testing, and the documentation.
2. The group organization A group should be organized so that individuals can contribute to the best of their abilities and tasks can be completed as expected.
3. Technical and managerial communications Good communication between the group members, and between the software engineering team and other project stakeholders, is essential.

5.3.1 Selecting Group members:

A manager or team leader's job is to develop a cohesive group and organize their group so that they can work together effectively. This involves establishing a group with the right balance of technical skills and personalities and organizing that group so that the members get together effectively.

A group that has complementary personalities might work better than a group that is selected solely on technical ability. People who are motivated by the work are likely to be the strongest technologically. It is occasionally impossible to choose a group with matching personalities.

If this is the case, the project manager should control the group so that individual goals do not take precedence over organizational and group objectives. This control is easier to achieve if all group members participate in each stage of the project. Individual initiative is most likely after group members are given instructions without being aware of the portion that their task plays in the overall project.

5.3.2 Group Organization:

The way that a group is organized influences the decisions that are made by that collective, the ways that information is exchanged, and the interaction between the development group and the outside project stakeholders.

1. Should the project manager be the technological leader of the group? The technical director or system architect is responsible for the critical technical decisions that have been made during software development. And Occasionally, the project manager has the skill and the experience to take on this role.

However, for large projects, it is best to appoint the chief engineer to be the project architect, who's going to take responsibility for technological leadership.

2. Who will be involved in the production critical technical decisions, and how will these be made? Will decisions be made by the system engineer, the project manager, or by reaching consensus between a wider variety of team members?

3. How will relationships with the external stakeholders and senior firm management be handled? In many cases, the project manager will be held responsible for these interactions, aided by the system architect if there is one. However, the alternative organizational model is to establish a dedicated role preoccupied with external liaison and appoint somebody with appropriate interaction abilities to that role.

4. How can groups integrate individuals who are not collocated? It is now commonplace for groups to include members from various organizations and people to work from home as well as in a shared office. This must be taken into consideration in the group decision-making processes.

5. How can knowledge be shared between the group? Group organization affects information-sharing as certain methods of organization are better to share than others. However, you must avoid too much information-sharing as individuals become overloaded and unnecessary information distracts them.

http://www.softwareengineering-9.com/Web/Management/Selection.htmlfrom their work.

Types of groups:

Small programming groups are typically organized in an informal way. The group's leader gets involved in the software development in conjunction with other group members. In an informal group, the work to be carried out is discussed by the group, and tasks are assigned according to ability and experience.

Extreme programming groups are continually informal groups. XP enthusiasts claim that formal structure inhibits information exchange. In XP, many choices that are usually seen as management decisions have been transferred to group members.

Informal groups can be highly successful, particularly when most participants in the group are experienced and competent. Such a group requires decisions by consensus, which improves consistency and performance.

Hierarchical groups are the bands that have a hierarchical structure with the collective leader at the top of the hierarchy. He or she has more formal permission than the group members and so can direct their work. There is a clear organizational

framework and decisions have been made towards the top of the pyramid and carried out by the people lower bring down the hierarchy.

Democratic and hierarchal group organizations do not formally acknowledge that there may be very significant differences in technical ability amongst group members. The best computer programmer may be up to 20 times more dynamic as the worst developers.

5.3.3 Group Communication:

It is essential that the band members communicate effectively and effectively with each other and with other project stakeholders. Group members have to exchange information on the status of their work, the design decisions that have been made, and changes to previous design decisions. They must resolve issues that arise with other relevant stakeholders and inform these stakeholders of changes to the system, the group, and the shipment plans.

The efficiency of transmission is influenced by:

1. **Group size** as a group gets bigger, it becomes more difficult for members to communicate effectively. The number of one-way communication connections is n * (n - 1), where n is the group size, so, with a group of eight members, there are 56 possible the communication paths.

   This means that it is quite possible that some individuals will rarely communicate with each other. Status differences among group members mean that communications are frequently one-way.

   Managers and experienced engineers tend to dominate communications with less experienced staff, who may be reluctant to begin a conversation or make critical comments.

2. **Group structure** People in unofficially structured groups are communicating more effectively than people in groups with a formal, hierarchical relationship. In hierarchical groups, the messages tend to flow up and knock down the hierarchy. People at the same level may not talk to each other.

   This is a specific problem in a large project with several development groups. If people working on various subsystems only communicate through their managers, then there are more likely to have delays and misconceptions.

3. **Group composition** People with the very same personality types may clash and, as a result, communications can be inhibited. Communication is also usually better in mixed groups than in single gender groups. Women are often

more collaboration oriented than men and may act as interaction controllers and mediators for the group.

4. The physical **work environment** the organization of the working environment is a major factor in facilitating or inhibiting communications. The available channels for communication There are many different forms of communication—face-to-face, e-mail messages, formal documents, telephone, and Web 2.0 technologies such as social networking and wikis. As project teams are becoming more and more distributed, with team members doing remotely, you need to make use of a range of technologies to facilitate communications.

   Project leaders usually work to tight time-limits and, consequently, they may try to use communication channels that don't take up too much of their time. They may therefore rely on discussions and formal documents to pass on information to project staff and stakeholders.

   Although this may be an effective approach to communication from a project manager's perspective, it is not usually very effective. There are often excellent reasons why people cannot attend meetings and so they do not listen to the presentation. Long documents are often never read since readers do not know if the documents are relevant.

   When multiple versions of the same document are produced, readers find it difficult to keep track of the changes. Effective communication is achieved when communications are two way, and the people involved can discuss issues and information and create a common understanding of proposals and problems.

   This can be done through meetings, although these are often overshadowed by powerful personalities. It is sometimes impractical to organize meetings at short notice. More and more project players include remote members, one that also makes meetings more challenging.

Takeaways:

- Good software project management is vital if software engineering projects are to be created on schedule and within budget.
- Software management is different from other engineering management. Software is intangible. Projects may be novel or innovative so there is no body of experience to tour guide their management. Software processes are not as experienced as traditional engineering processes.

- Risk management is currently recognized as one of the most valuable project management tasks.
- Risk management involves identifying and evaluating major project risks to determine the probability that they will occur and the implications for the project if that risk does arise. You should make plans to avoid, manage, or transaction with the likely risks if or when they arise.
- People are motivated by interacting with other people, the recognition of management and their peers, and by being given opportunities for personal growth.
- Software engineering groups should be small and consistent. The key factors that influence the effectiveness of a group are the people within this group, the way that it is organized, and communications between the group members.
- Communications within a group have been influenced by factors such as the status of group members, the size of the group, the gender structure of the group, personalities, and accessible through correspondence channels.

Questions:

- You are asked by your manager to provide software to a schedule that you know can only be met by asking your project team to work unpaid overtime. All the team members have young children. Discuss whether you need to accept this demand from your manager or whether you should convince your team to give their time to the association rather than to their families. What factors could be significant in your decision?
- As a programmer, you will be offered promotion to a project management position, but you feel that you can make a more efficient contribution in a technical rather than a managerial role. Discuss whether or not you should accept the advancement.

- Fixed-price contracts, where the contractor bids a fixed price to complete a system development, may be used to move project risk from client to contractor. If anything goes wrong, the contractor has to pay. Suggest how the use of such contracts may increase the likelihood that product risks will arise.

References

Aron, J. D. (1974). The Program Development Process. Reading, Mass.: Addison-Wesley.

Baker, F. T. (1972). 'Chief Programmer Team Management of Production Programming'. IBM Systems J., 11 (1), 56–73.

Brooks, F. P. (1975). The Mythical Man Month. Reading, Mass.: Addison-Wesley.

Hall, E. (1998). Managing Risk: Methods for Software Systems Development. Reading, Mass.: Addison-Wesley.

Marshall, J. E. and Heslin, R. (1975). 'Boys and Girls Together. Sexual composition and the effect of density on group size and cohesiveness'. J. of Personality and Social Psychology, 35 (5), 952–61. Maslow, A. A. (1954). Motivation and Personality. New York: Harper and Row.

Ould, M. (1999). Managing Software Quality and Business Risk. Chichester: John Wiley & Sons.

❖ ❖ ❖

6

SOFTWARE REQUIREMENTS

Unit Structure:

6.0 Objectives

- know the ideas of user and system requirements and why these requirements need to be written in different ways
- understand the differences between functional and non-operational software requirements
- understand how the necessary conditions may be organized in a software specification for the document
- appreciate the principal specifications for the engineering activities of elicitation, analysis and validation, and the relationships between these activities
- understand why the terms and conditions the management is necessary and how it supports other standards laid down in engineering events

Introduction:

Requirement's engineering is conceded with instituting what the system should do, its desired and essential emergent properties, and the restrictions on system operation and the software development processes. You can consequently think of requirements engineering as the communications process between the software clients and customers and the software developers.

Requirement's engineering is not just a technical process. The system requirements are influenced by users' likes, dislikes, and prejudices, and depending on the political and organisational issues. These are fundamental human traits, and new technologies, such as use-cases, scenarios and proper methods do not help us much in solving these problems.

The term 'requirement' is not used regularly in the software industry. In some cases, a requirement is simply a high-level, abstract declaration of a service that a system should provide or a constraint on a system. Some of the problems that occur during the requirements engineering process are a result of failing to make a clear distinction between these different levels of description. There are two terminologies utilized in the requirement engineering User Requirements and System Requirements.

User requirements and system requirements may be defined as follows:

1. **User requirements** are statements, in a natural language plus diagrams, of what facilities the system is expected to provide to system users and the limitations under which it must operate.

2. **System requirements** are a more complete description of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define precisely what is to be implemented. It may be part of the agreement between the system buyer and the software developers.

Different levels of requirements are helpful because they communicate information about the system to the different kinds of reader example:

User Requirement Definition:

The chemist billing system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification:

1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 12.00 on the last business day of the month.

1.3 A report shall be created for each clinic and shall make a list the individual drug names, the total number of prescriptions, the number of doses approved, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) the individual reports shall be created for every dose unit.

1.5 Access to all cost reports shall be limited to authorized users listed on a administration access control list.

6.1 Functional and non-functional requirements

Software system requirements are frequently categorized as functional requirements or non-functional requirements:

Functional requirements The following are the statements of services the system should provide, how the system should react to inputs, and how the system should perform situations. In some cases, the functional requirements may also explicitly define what the system should not do. Functional system requirements vary from general specifications covering what the system should do in order to very particular requirements reflecting local ways of working or the organization's existing systems. For example, here are examples of functional requirements:

1) Authentication and authorization of user whenever he/she logs into the system.

2) System should raise query when hazardous issue is highlighted.

3) A Verification email and OTP is sent to user whenever he/she registers for the first time on some software system or during ecommerce transaction an SMS is sent for the amount deduction.

The functional requirements specification of a system must be both complete and consistent. Completeness means that all services required by the user must be defined. Consistency means that requirements should not have conflicting definitions.

In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is simple to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is an individual or a role that is affected by the system in some way. Stakeholders have different— and frequently inconsistent—needs. These inconsistencies may not be evident when the requirements are first specified, so conflicting requirements are included in the specification.

Non-functional requirements These are constraints on the services or features offered by the system. They include timing constraints, constraints on the developmental process, and constraints imposed by standards. Non-functional requirements often be applied to the system, rather than individual system features or services.

Non-functional requirements, as the name suggests, are the necessary conditions that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system characteristics such as reliability, response time, and warehouse occupancy.

Alternatively, they may define constraints on the computer system implementation such as the capabilities of I/O devices or the data statements used in interfaces with other systems. Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system. Non-functional requirements are frequently more critical than individual functional requirements.

System users can usually find methods to work around a system function that does not really meet their needs. However, failing to meet a non-functional requirement can imply that the whole system is unusable.

For example

1) Emails should be sent with a latency of no greater than 12 hours from such an activity.

2) The processing of each enquiry should be done within 20 seconds

3) The site should load in 4 seconds when the number of simultaneous users are greater than 11000

The implementation of these requirements can be diffused throughout the system.

There are two reasons for this:

1. Non-functional requirements may affect the overall structural design of a system rather than the individual components. For example, to ensure that the performance requirements are fulfilled, you may have to organize the system to reduce the communication between the components.

2. A single non-functional requirement, such as a security requirement, could produce several related functional requirements that define new system services that are necessary. In addition, it may also generate in accordance with the provisions that limit the existing requirements.

   Non-functional requirements arise through user needs, due to the fact of budget constraints, organizational policies, the need for the interoperability with other software or hardware systems, or outside factors such as safety regulations or confidentiality legislation.

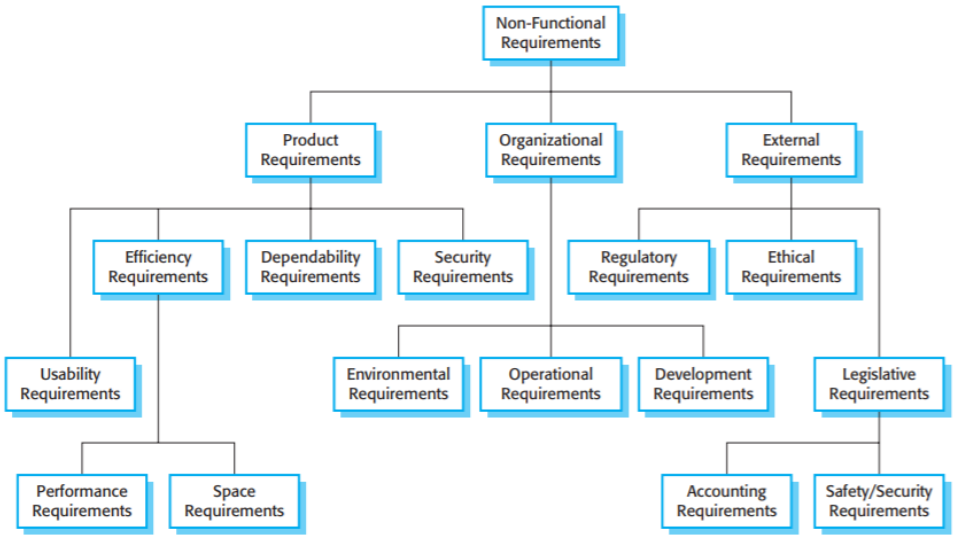


Fig: Classification of Non-functional Requirements

Non-functional requirements may come from the necessary characteristics of the software or from outside sources:

1. **Product requirements** These requirements stipulate or constrain the behaviour of the software.

   Examples include performance conditions laid down on how fast the system must execute and how much memory it requires, reliability in accordance with the provisions that set out the acceptable failure rate, security requirements, and the usability requirements. The product prerequisite is an availability requirement that defines when the system should be available and the allowed down time each day.

It says nothing about the features and clearly identifies a restriction that must be considered by the system architects.

2. **Organizational requirements** These requirements are comprehensive system requirements derived from policies and procedures in the customer's and programmer's organization.

   Operational process specifications determine how the system can be used, implementation process requirements define the programming language, development environment, or process criteria that will be used, and environmental requirements define the system's operating environment. How users authenticate themselves to the system is defined by the organisational requirement.

3. **External requirements** This broad heading encompasses all criteria arising from factors outside of the system and its creation process. This may include regulatory standards that specify what must be done in order for a regulator, such as a central bank, to allow the system for use; legal requirements that must be met to ensure that the system works within the law; and ethical requirements that ensure that the system is appropriate to its users and the general public.

A common problem with non-functional requirements is that customers or customers often propose these requirements as general goals, for example the ease of use, the ability of the system to recoup from failure, or rapid user response. The table below show the metrics that you can use to determine non-functional system properties.

Property	**Measure**
Speed	Processed deals/second User/event reply in the time Screen refresh time
Size	Bytes Number of ROM chips
Reliability	Mean time to failure. Probability of inaccessibility Rate of failure occurrence Availability
Robustness	Time to restart after failure. Percentage of events is caused by the failure. Probability of information from the corruption on failure
Portability	Percentage of target reliant statements Number of target systems
Ease of Use	Training period in which the Number of help frames

It is difficult, in practice, to separate functional and non-functional requirements throughout the requirements document. If the non-functional requirements are specified separately from the functional requirements, the relationships between them may be hard to understand. However, you should explicitly emphasize the requirements that are clearly related to emergent system properties, such as per performance or reliability. Non-functional requirements such as reliability, safety, and confidentiality requirements are also essential while defining the conditions necessary and should not be side-lined.

6.2 User Requirements:

The user requirements for a system should describe the functional and non-functional requirements so that they are easily understood by system users without thorough technical knowledge. They should only specify the external behavioural patterns of the system and should avoid, as far as possible, system design characteristics.

Consequently, if you are writing user requirements, you must not use the software jargon, structured notations, or formal notations, or explain the requirement by describing the system implementation. List of problems that occur when requirements are written in natural language.

1. **Lack of clarity** It is sometimes difficult to use language in an accurate and unambiguous manner without making the document wordy and hard to read.

2. **Requirement's confusion** Functional requirements, non-functional requirements, system objectives and design information might not be clearly distinguished.

3. **Requirement's amalgamation** Several different requirements can be expressed together as a single requirement.

User requirements that include too much data constrain the freedom of the system developer to provide innovative solutions to user problems and are difficult to understand. The user requirement should simply focus on the key and all the essential amenities to be provided.

To minimise misunderstandings when writing user requirements, following steps need to be taken:

- Invent a standard format and ensure that all requirement definitions follow that format.
- Standardising the format makes exceptions less likely and the necessary conditions easier to check. The format I use shows the initial requirement in boldface, including a declaration of rationale with each user requirement and a reference to the more detailed system requirement specification.
- You may also include information on who proposed the requirement (the requirement source) so that you will know whom to consult if the requirement must be changed.
- Use language steadily.
- You should always distinguish between compulsory and desirable requirements. Mandatory requirements are requirements that the system must support and are typically written using 'shall'. Appropriate requirements are not essential and are written using 'should'.
- Use text highlighting (bold, italic or colour) to pick out crucial parts of the requirement.

Avoid, as far as possible, the use of computer terminology. Inevitably, however, the comprehensive technical terms will invade the user requirements.

6.3 System Requirements:

System requirements are expanded versions of the user needs that are used by software engineers as the starting point for the system design. They add details and explain how the user requirements should be offered by the system. They may be used as part of the contract for the application of the system and should therefore be a complete and consistent specification of the whole system Ideally, the system requirements should simply explain the external behaviour of the system and its operational constraints. They should not be worried about how the system should be constructed or implemented. However, at the level of detail necessary to completely specify a complex software system, it is impossible, in practice, to exclude all information about the project.

Natural language is often used to write system specifications for the specifications as well as user requirements. However, because system requirements are more detailed than user requirements, natural language specifications may be confusing and hard to understand:

Natural language understanding depends on the specification readers and writers that uses the same words for the same notion. This leads to misunderstandings

because of an ambiguity of natural language. A natural linguistic requirements specification is over flexible. You could tell you the same thing in completely different ways. It is up to the reader to find out when requirements are the same and when they are distinct. There is no easy way to modularise natural-language requirements. It may be difficult to find all the related terms and conditions set forth. To discover the consequence of a transformation, you may need to look at every requirement rather than at simply a group of connected requirements.

Structured language Specification:

Structured natural language is a way of writing system specifications for the where the freedom of the requirements writer is limited, and all the demands are written in a standard way. The advantage of this approach is that it maintains most of the fluency and understandability of natural language but ensures that some degree of uniformity remains imposed on the specification.

Structured language notations restrict the terminology that can be used and use templates to specify system requirements. They can incorporate control constructs derived from programming languages and graphical emphasizing to partition the specification.

Insulin Pump/Control Software/SRS/3.3.2	
Function	Compute insulin dose: Safe sugar level
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1)
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered
Destination	Main control loop
Action: CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.	
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2
Side effects	None

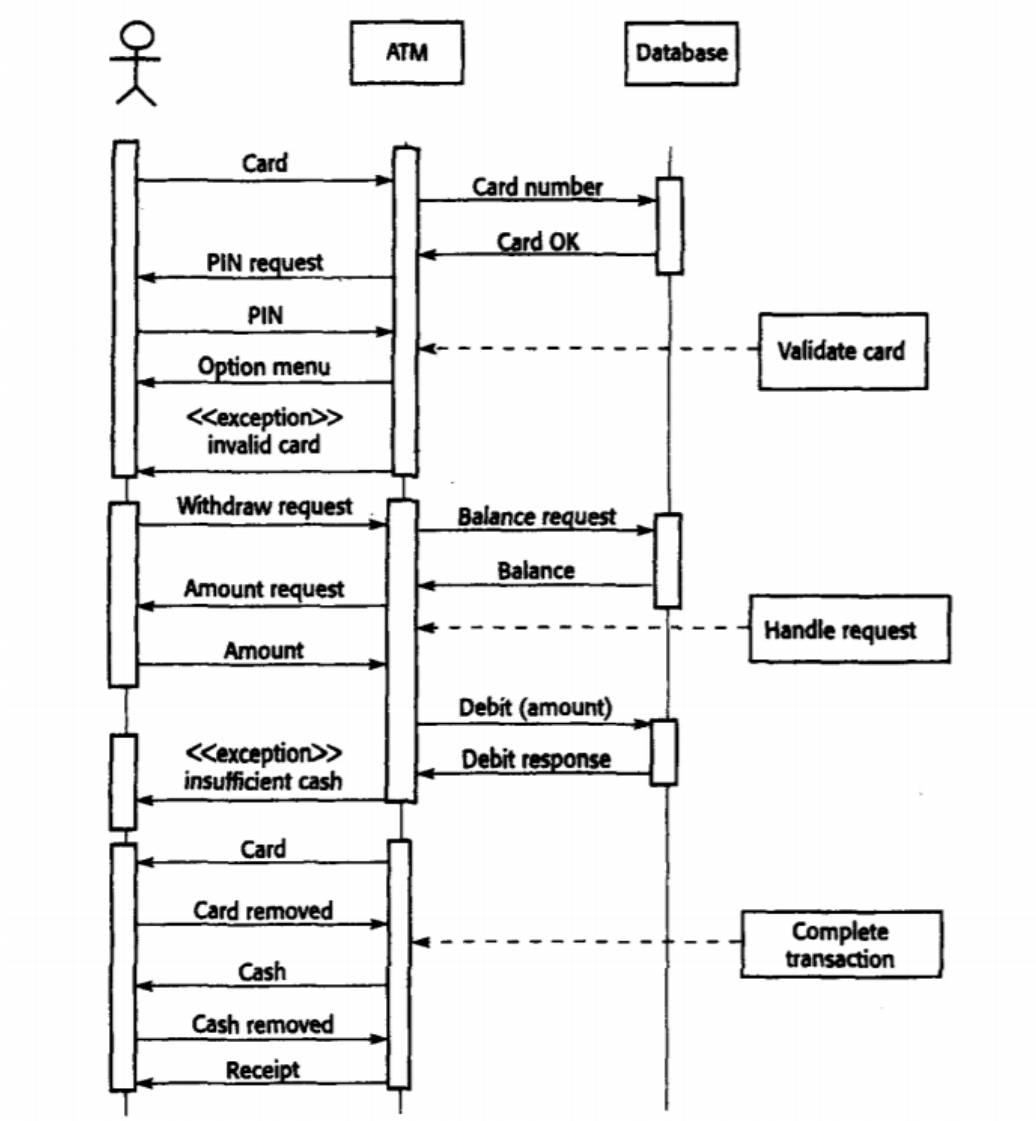
Reference: Sommerville 8e Chapter 6 Software Requirements

When a standard form can be used to specify the functional requirements, the following data should be included:

1. Description of the feature or entity being specified
2. Explanation of its inputs as well as where these come from
3. Description of its outputs and somewhere these go to
4. Suggestion of which other entities is used (the requires part)
5. Explanation of the action to be carried out
6. If a functional approach is used, a pre-condition setting out what should be true before the function is called and a post-condition if you specify what is true after the function is called
7. Description of the side effects (if any) of the operation

Graphical models are most useful when you need to show how state changes using sequence diagram there are three basic subsequence used:

1. Validate card the user s card is authenticated by checking the card number and user's PIN.
2. Handle request the user s request is handled through the system. For a withdrawal, the database must be queried to check the user's balance and to debit the sum withdrawn. Notice the exception here if the requestor does not have enough cash in their account.
3. Complete transaction the user s card will be returned and, when it is removed, the cash and receipt are delivered.



Reference: Sommerville 8e Chapter 6 Software Requirements

6.4 Interface Specification:

Almost all software systems must operate with existing systems which have already been implemented and installed in an environment. If a new system and the existing systems must work together, the: interfaces of existing systems must be precisely specified. These specifications should be defined early in the process and incorporated into the requirements document.

There are three types of interface: that may have to be defined:

1. **Procedural interfaces** where existing systems or sub-systems provide a range of services that are accessed by calling interface procedures. These interfaces have sometimes called Application Programming Interfaces.

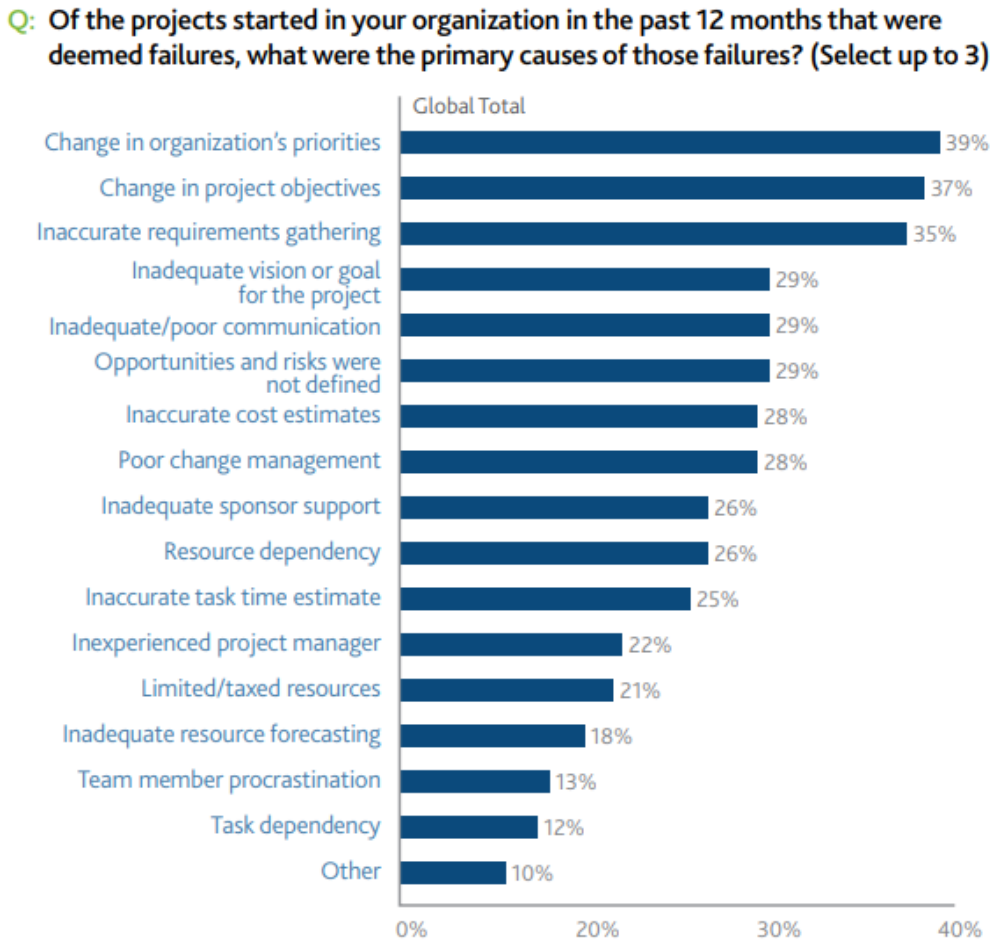
2. **Data structures** that are passed from a particular sub-system to another. Graphical information from the models are the best entries for this type of description. If necessary, system descriptions in Java or python may be automatically generated from these descriptions.

3. **Representations of data** that have been established for an already existing sub-system. These interfaces are most common in the integrated, real-time system. Some programming languages like Ada support this level of specification. However, the best way to explain these is probably to use a diagram of the structure along with annotations explaining the function of every group of bits.

6.5 The Software Requirements Document

How can a client and provider reach a shared understanding on the concept of the product? It is not easy when they speak several languages. A customer defines a product at a high-level concept level, focusing on the external system behaviour: what it will be doing and how end-users will work with it. Meanwhile, developers think of the product is in terms of its own internal characteristics. That is why a Business Analyst steps in to convert a customer's needs into requirements, and further turn them into tasks for developers. This is initially done by writing **software requirements specifications.**

Poorly specified requirements inevitably lead to some functionality not being included in the application. Every assumption should be clearly communicated rather than just implied. For instance, NASA's Mars Climate Orbiter mission failed due to conflicting units of measure. Nobody specified beforehand that the mindset-control system and navigation software must both use the same metric or imperial units. For some, it went without saying, everyone else did not find it as obvious. This is a widespread problem that keeps happening even to the best specialists if not prevented.



Inaccurate requirements gathering is one of the top causes for project failure, Source: PMI's Pulse of the Profession

The manufacture of the requirements stage of a software development process is **Software Requirements Specifications (SRS)** (also called a **requirements document**). This report lays a foundation for software development activities and is constructing when whole requirements are elicited and analyzed. **SRS** is a formal report, which serves as a representation of software that enables the customers to review whether it (SRS) is in accordance with their requirements. Also, it includes user requirements for a system in addition to the detailed requirements of the system requirements.

The SRS is a specification for a particular software product, program, or a set of applications which perform functions in a specific environment. It serves several purposes depending on who is writing it. First, the SRS could be written by the client of a system.

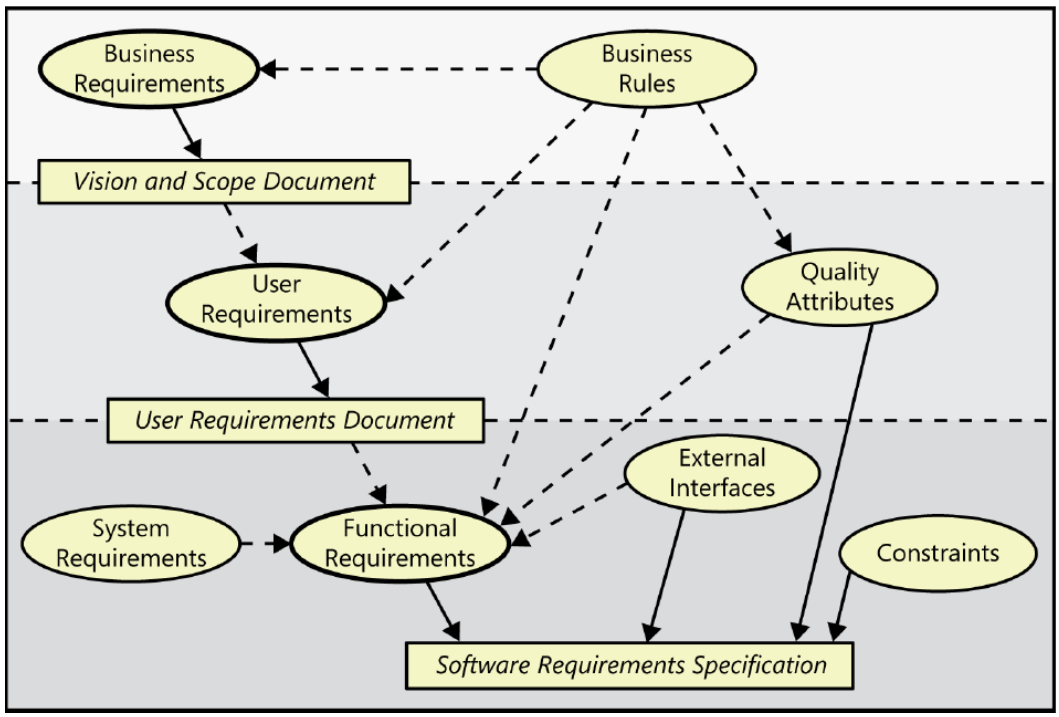
Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is used to define the needs and expectation of the users. The second case, SRS, is written for different purposes and serves as a contract detail between customer and developer.

SRS is at the bottom of the software requirements pyramid, which goes like this:

Top tier – the high-level business requirements (BRs) dictating the aim behind the product,

Middle tier – user requirements (URs) that picture an end-user and their needs, and

Bottom tier – SRSs that specify the product's features are available in tech terms.

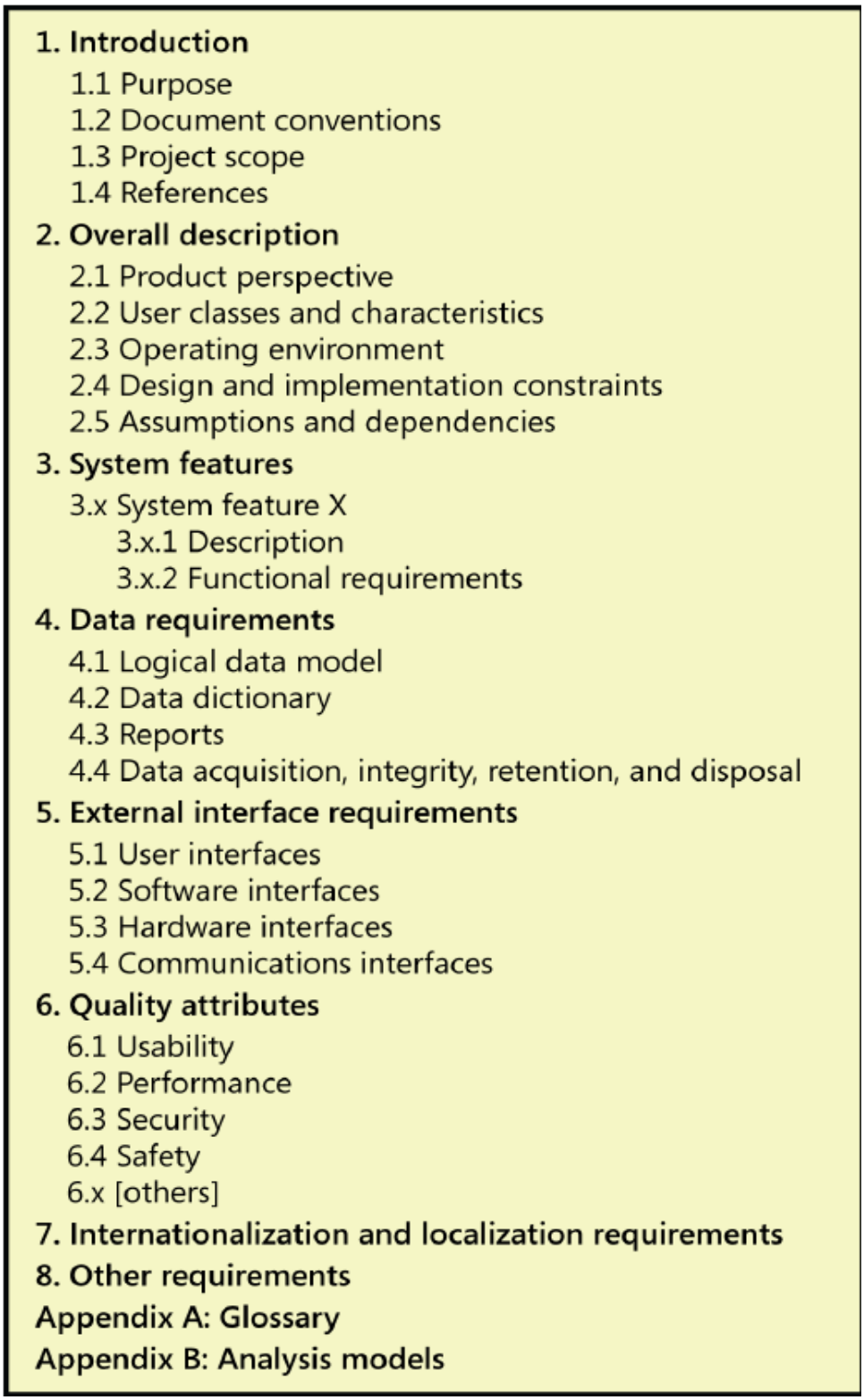


Solid arrows show how requirement types (ovals) are sorted into the documents (rectangles). While dotted arrows show which requirement type is the origin of or influences another one, Source: Software Requirements by Karl Wiegers Joy Beatty

SRS Software and Template

An SRS template provides a handy reminder of what kinds of knowledge to explore. Every software development organization would adopt one or more standard SRS templates for their projects. There are various templates available

depending on the design class. Here is an example of an SRS template that works well for many types of projects.



Source: Software Requirements by Karl Wiegers Joy Beatty

Structure with explanation of SRS:

Chapter	Description
Preface	This should define the anticipated readership of the document and describe its version history, including a rationale for the creation of a new edition and a summary of the changes made in each version.
Introduction	This should describe the necessity for the system. It should briefly describe its functions and explain how it will work with other systems. It should explain how the system fits into the overall business or strategic, to achieve the goals of the organisation commissioning the software
Glossary	This should define the technological terms used in the document You do not have to make any assumptions about the experience or expertise of the reader
User Requirement Definition	The service provided for the user as well as the non-functional system requirements should be outlined in this section. This description may use natural language, diagrams or other notations that are to be understood by customers. Product and process standards that should be followed should be specified.
System architecture	This chapter is required to submit an extremely high-level overview of the anticipated system structure showing the allocation of functions across system modules. Architectural components that are recycled must be highlighted.
System Requirements Specification	This should describe the functional and that are not related to-functional demands in more detail. If necessary, more detailed information can also be added to the non-functional

	requirements, e.g., interfaces with other systems may be defined.
System Models	This should set out a single result or multiple system models showing the relationships between the components in the system and the system and its environment These might be object models, data-flow models, as well as the semantic data models.
System Evolution	This should describe the basic assumptions on which the system is based and anticipated changes due to hardware evolution, changing user needs, etc.
Appendices	These should provide detailed, particular information which is related to the application which is being developed. Examples of appendices that can be included are hardware and database descriptions. Hardware requirements define the minimal and optimal structures for the system. Database requirements define the logical organisation of the data that are used in the system and the relationships between data
Index	several indexes to the document can be included. As well as a normal alphabetic index there might be an index of diagrams, an index of functions, etc.

Takeaways:

- Requirements for a software system set out what the current system should do and define the restrictions on its operation and implementation.
- Functional requirements are the declarations of the services that the system will have to provide or are descriptions of how certain calculations must be carried out.
- Non-functional requirements frequently constrain the entire system being developed and the development process being used. These could be product requirements, organizational requirements, or external requirements. They

often relate to the evolving properties of the system and therefore apply to the scheme.

- The software requirements document that is an agreed statement of the system requirements. It should be organized so that both approach customers and software developers will be able to use it.

Reference:

Beck, K. (1999). 'Embracing Change with Extreme Programming'. IEEE Computer, 32 (10), 70–8.

Crabtree, A. (2003). Designing Collaborative Systems: A Practical Guide to Ethnography. London: Springer-Verlag.

Kotonya, G. and Sommerville, I. (1998). Requirements Engineering: Processes and Techniques. Chichester, UK: John Wiley and Sons.

Larman, C. (2002). Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process. Englewood Cliff, NJ: Prentice Hall.

Questions:

- Write a set of functional and non-functional specifications for the banking system, setting out its expected reliability and response time.
- With an example write the customer requirements of ATM machine
- Write a short note the significance of SRS.
- List the advantages of SRS.

❖ ❖ ❖

UNIT 3

7

REQUIREMENT ENGINEERING PROCESS

Unit Structure:

7.0 Objectives

This Chapters help you to understand.
What is Requirement and why it is Important.
How to manage requirement and its process.
Different types of System Model and how it works.
Architectural design and different component used.
How to Structure a system and its different styles.

Requirement Engineering Processes

7.1 Introduction

Requirement engineering is a process Requirements engineering is the systematic use of proven principles, techniques, languages, and tools for the cost effective analysis, documentation, and on-going evolution of user needs and the specification of the external behaviour of a system to satisfy those user needs.

The process to determine the requirement specification of the software is called as requirement engineering process. Both the software engineer and customer take an active role in software requirements engineering, a software engineer performs requirements analysis.

The use of Requirement Engineering Process:

If you don't analyse, it's highly likely that you'll build a very elegant software solution that solves the wrong problem.

The result is: wasted time and money, personal frustration, and unhappy customers.

Requirement engineering process includes four high level activities:

1) If the system is useful to the business (Feasibility study).
2) Discovering requirements (Elicitations and Analysis).
3) Converting these requirements into some standard form (Specification).
4) Checking that the requirements actually define the system that the customer wants (Validation).

Requirement Involved in Requirement Engineering

Depending on the type of system being developed and the specific practices of the organization(s) involved the activities or tasks that are involved in requirements engineering vary widely.

Activities involved In requirement engineering

1. Requirements inception
2. Requirements elicitation
3. Requirements analysis and negotiation
4. System modelling
5. Requirements specification
6. Requirements validation
7. Requirements management

1. Requirements Inception

Inception is a task where the requirement engineering requests a set of queries to find a software process. In requirement inception task it understands the problem and assesses with the proper solution. It cooperates with the relationship between the customer and the inventor. The overall scope and the nature of the question are decided by the developer and customer.

2. Requirements Elicitation

In requirements engineering, this is the exercise of investigating and learning the requirements of a system from stakeholders. users, customers, and other.

The requirements elicitation is also sometimes referred to as "requirement gathering".

3. Requirements Analysis and Negotiation

The tasks that are involved in requirements analysis and compromise are identification of requirements including new ones if the development is iterative and solving conflicts with stakeholders.

4. System modelling

There some engineering fields or specific study in situations in picture that need the product to be completely designed and modelled before its construction or fabrication starts. Hence, the design phase must be performed in advance.

Many fields might derive models of the system with the Lifecycle Modelling Language, whereas others might use UML.

5. Requirements specification

Requirements are documented in a formal artefact called Requirements Specification.

Nevertheless, it will become official only after validation.

A requirement specification can contain both written and graphical (models) information if necessary. Example is Software Requirements Specification (SRS).

6. Requirements validation

It is the process of checking whether the documented requirements and models are consistent and meet the needs of the stakeholder.

Only if the final draft passes the validation process, the RS becomes official.

7. Requirement Management

This phase manages all activities related to requirement after it is put into use.

The two important task of requirement engineering process are

1. **Finding the requirement specifications.**
2. **Reviewing the requirements to ensure their correctness.**

7.1.1. Requirement analysis:

Requirements analysis is a software engineering task that bridges the gap between system level requirements engineering and software design.

Requirements engineering activities result in the specification of software's operational characteristics (function, data, and behaviour), indicate software's interface with other system elements, and establish constraints that software must meet.

The analyst must define all externally observable data objects, evaluate the flow and content of information, define and elaborate all software functions, understand software behaviour in the context of events that affect the system, establish system interface characteristics, and uncover additional design constraints. Each of these tasks serves to describe the problem so that an overall approach or solution may be synthesized.

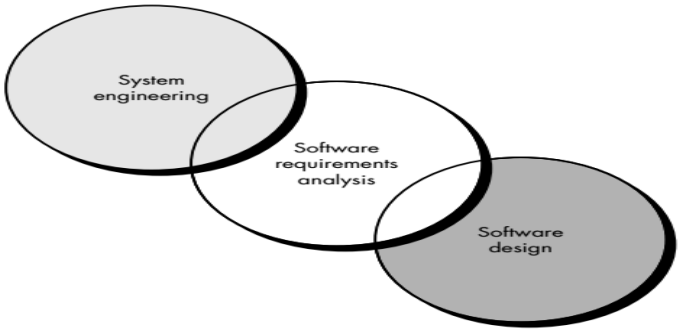


Fig: Analysis Concepts and Principles

For example, an inventory control system is required for a major supplier of auto parts. The analyst finds that problems with the current manual system include

(1) inability to obtain the status of a component rapidly

(2) two- or three-day turnaround to update a card file

(3) multiple reorders to the same vendor because there is no way to associate vendors with components, and so forth.

Once problems have been identified, the analyst determines what information is to be produced by the new system and what data will be provided to the system. For instance, the customer desires a daily report that indicates what parts have been taken from inventory and how many similar parts remain.

The customer indicates that inventory clerks will log the identification number of each part as it leaves the inventory area.

All analysis methods are related by a set of operational principles:

1. The information domain of a problem must be represented and understood.
2. The functions that the software is to perform must be defined.
3. The behaviour of the software (because of external events) must be represented.
4. The models that depict information, function, and behaviour must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
5. The analysis process should move from essential information toward implementation detail.

Requirement engineering process consists of following processes:

1. Requirement elicitation and analysis
2. Requirement validation
3. Requirement management

7.1.2. Feasibility study

For all new systems the requirement engineering process should start with feasibility study.

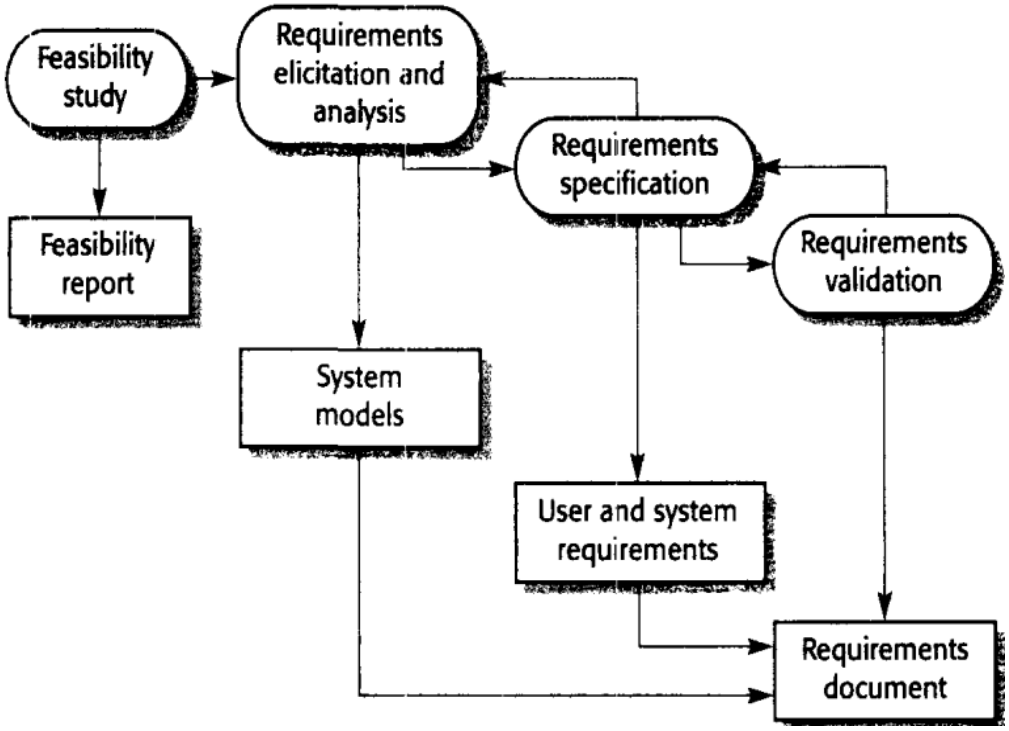


Fig. Feasibility Study

The input to the feasibility study is preliminary requirements, an outline description of how the system is intended to support business process.

The results are the report that recommends whether it is worth carrying on with the project.

Thus, a feasibility study is a short-focused study that aims to answer several questions like system contribution, system implementation and system integration.

If the system does not support the business objectives it has no real value to the business.

Carrying out a feasibility study involves information assessment, information collection and report writing.

The information assessment phase identifies the information that is required and once the information is gathered discussion with various sources can be done.

7.1.3 Requirement Elicitation and analysis

In this activity software engineers work with customers and system end users to find about their application domain.

This activity may involve a variety of people in the organization

The term stakeholder is used to refer to a person or group who will be directly or indirectly affected by the system.

It is defined as a process of requirement gathering which focuses on "What is the scope and objective of the software?" and "how it can be accomplished?".
The users and the developers are the stakeholders who are the interested in the successful development of the software.

The various tools in elicitation are meetings, interviews, questionnaire, observation

collaboration, video conferencing.

The output of requirement elicitation includes the following:

- Statement of need and feasibility.
- Statement of the scope for the system.
- List of stakeholders participated in the process of requirement gathering.
- Description of the system's technical environment.
- Specification of non-functional requirements.

7.1.4 Eliciting and understanding stakeholder requirements is difficult because

Stakeholders often don’t know what they want from the computer system hence they may make unrealistic demands.
Stakeholders naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers without experience of customer domain must understand these requirements.
Different stakeholders have different requirements which they may express in different ways. Requirement engineers must consider all potential sources of requirements and discover commonalities and conflict.
Political factors may influence the requirements of the system.

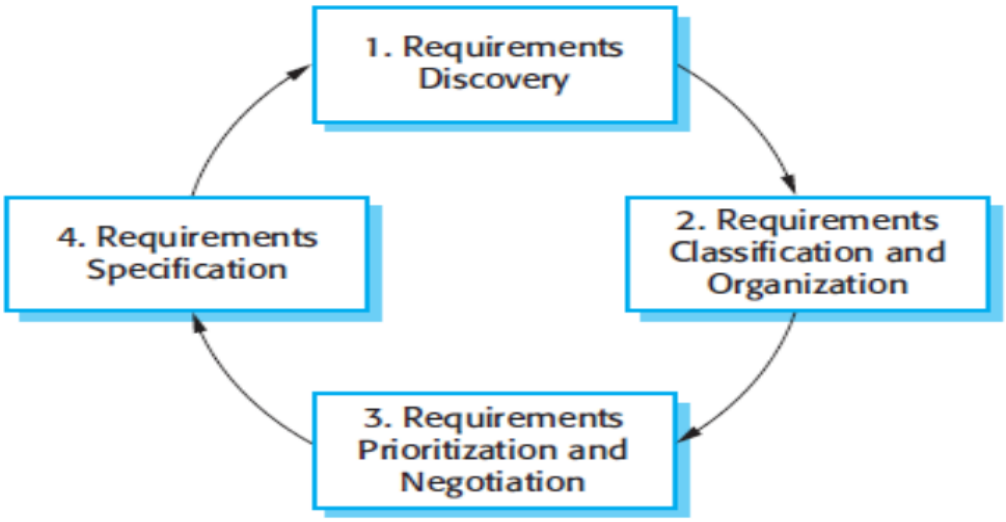
The three main problems faced during the requirement gathering/elicitation process

a) Problem of scope

b) Problem of understanding

c) Problem of instability

The activities used in the elicitation and analysis process are:



Requirements discovery – This is the process on interacting with stakeholders in the system to collect their requirement. Domain requirements from the stakeholders and documentation are discovered during this activity.

It is the process of gathering information about the proposed and existing system and filtering the user requirements from it. Sources of information during the requirements discovery phase includes documentation, system stakeholders and specification of similar systems.

The different ways of gathering information are

1. **Viewpoints** – It can be used to classify stakeholders and other sources of requirements.

There are three different types of viewpoints.

– Interactor viewpoints represent people or other systems that interact directly with the system

– Indirect viewpoints represent stakeholders who do not use the system themselves but who influence the requirements in some way

– Domain viewpoints represents domain characteristics and constraints that influence the system requirements.

- Viewpoints provide different types of requirements. Interactor viewpoints provide detailed system requirements whereas indirect viewpoints are more likely to provide higher level organizational requirements and constraints and domain viewpoints normally provide domain constraints that apply to the system.

  The initial identification of viewpoints that are relevant to a system can sometimes be difficult. Viewpoints that provide requirements may come from the marketing and external affairs departments.

For a non-trivial system there is huge number of viewpoints

- Interviewing – In this portion questions are put to the stakeholders about the system.

  Interviews can be of open type (no predefined set of questions) or closed type (predefined set of questions).

- Interviews are good for getting an overall understanding of what the stakeholders do, how they might interact with the system and the difficulties that they face with current systems.

- Interviews are not so good for understanding the requirements from the application domain because there are subtle power and influence relationships between the stakeholders in the organization

- **Effective interviews have two characteristics**

  – They are open minded avoid perceived ideas about the requirements and are willing to listen to stakeholders

  – They prompt the interviewee to start discussion with a question

- Information from interviews supplements other information about the system from documents, user observations and so on.

- Sometimes interviews may be the only source of information but still it may miss an important thing and hence it should be used alongside other requirements techniques

- Scenarios – They are particularly useful for adding detail to an outline requirements description.

- Several form of scenarios provide different types of information at different levels of detail about the system.
- The scenario starts with an outline of the interaction and during elicitation details are added to create a complete description of that interaction
- Scenario based elicitation can be carried out informally where the requirements engineer works with stakeholders to identify scenarios
- Scenarios may be written as text, supplemented by diagrams, screen shots etc.
- Use-cases – They are scenario based technique for requirements elicitation.
- They have now become a fundamental feature of the UML notation for describing object oriented system models.
- Use-cases identify the individual interactions with the system
- They can be documented with text or linked with UML models
- The UML is a de facto standard for object oriented modelling so use cases and use case based elicitation is increasingly used for requirements elicitation

Requirements classification and organization – This activity takes the unstructured collection of requirements and organizes them into clear 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 separate activities.

Requirement prioritization and negotiation – 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 must meet to resolve differences and agree on compromise requirements.

Requirements documentation – The requirements are documented and input into next round of spiral.

Requirement Analysis:

It is a structures and organized method for identifying the set of resources to satisfy the users need.

It acts as a transformation between the system need and the design concept.

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

7.2 Requirements Validation

It is concerned with showing that the requirements define the system that the customer wants.
It overlaps analysis in that it is concerned with finding problems with the requirements.
It is important because errors in requirements documentation can lead to extensive rework costs when they are discovered during development process.
The cost of fixing a requirements problem is much greater than repairing design or coding errors

• Various checks carried out on requirements in requirements document are

1. **Validity checks** A user may think that a system is needed to perform certain func tions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stake holder community

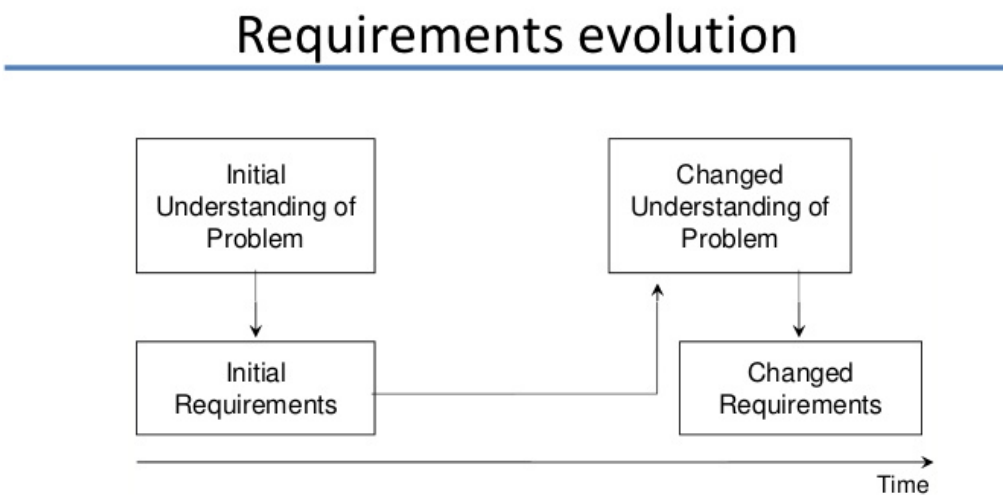
2. **Consistency checks** Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.

3. **Completeness checks** the requirements document should include requirements that define all functions and the constraints intended by the system user.

4. **Realism checks** Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These

checks should also take account of the budget and schedule for the system development.

5. **Verifiability** To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.



There are several requirements validation techniques that can be used individually or in conjunction with one another:

1. Requirements reviews the requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.

2. Prototyping In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

3. Test-case generation Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be re-considered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

7.2.1 Principles of Requirements Validation.

1: Involvement of the correct stakeholders.

2: Unravelling the identification and the correction of errors.

3: Validation from different views.

4: Satisfactory change of documentation type.

5: Construction of development artifacts.

6: Recurring authentication (Validation).

7.2.2 Requirements Validation Technique

- **Requirements reviews** – The requirements are analysed systematically by a team of reviewers
- **Prototyping** – In this approach to validation an executable model of the system is demonstrated to end users and customers
- **Test case generation** – Tests for the requirements are devised as a part of validation process. Developing tests from the user requirements before any code is written in an integral part of extreme programming
- Requirement review is a manual process that involves people from both client and contractor organizations.
- They check the requirements document for irregularities and errors.
- Requirements reviews can be informal (involves contractors discussing requirements with stake holders) and formal (the development team walks through the system requirements)
- Reviewers may also check for

  – Verifiability

  – Comprehensibility

  – Traceability

  – Adaptability

7.2.3 Requirements Management

Requirements for the large system keeps on changing due to which stakeholders understanding of problem is constantly changing. It is hard for the users and system customers to anticipate what effects the new system will have on organization.

Requirement management is the process the principal concerns are of managing the changes to requirements.

- Managing the relationships between requirements.
- Managing priorities between requirements.
- Managing the dependencies between different documents, specification and other documents produced in the systems engineering process.
- Managing changes to decided requirements.

New needs and priorities are discovered

– Large systems have different users having different requirements and priorities.

– After delivery new features may be added for user support if the system is to meet its goal.

– Business and technical environment changes after installation and these changes must be reflected in the system.

7.2.4 Requirements Management Planning

- Planning is an essential first stage in the requirements management process.
- During the requirement management stage, following are decided: -

1. **Requirements reviews** the requirements are analysed systematically by a team of reviewers who check for errors and inconsistencies.

2. **Prototyping** In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.

3. **Test-case generation** Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and

should be re-considered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

Requirements management needs automated support and the software tools, and this should be chosen during the planning phase.

The needed tool support is: -

1) **Requirements storage: -** The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.

2) **Change management: -** The process of change management is simplified if active tool support is available.

3) **Traceability management: -** some tools are available which use natural language processing techniques to help discover possible relationship between requirements.

7.2.5 Requirement Revies

The various techniques used for requirement validations are

a) Requirement reviews

b) Perspective-based reading

c) Validation through prototypes

d) Using checklists for validation

Requirements review is a technique used to validate the requirements by a group of people. It is a formal process which involves readers from the both sides of clients and developers. Reviews help customers and developers to resolve problems at early stages of SDLC.

Requirements review process consists of following points

a) **Plan Review:** Team is selected, time and place are decided for the review meeting with all the requirements to be checked.

b) **Distribute Documents:** Brochures are distributed among the review team members so, each member gives there review on the documentation.

c) **Prepare for Review:** Each Persons read the relevant documents for inconsistencies, conflicts, omissions, and other problems before review meeting.

d) **Hold Review Meeting:** Individual remarks and glitches are discussed, set of actions to address the problem is agreed.

e) **Follow-up Actions:** Checks for the work completion and the progresses of the task given to the individual person of team.

f) **Rescript Document:** Documents were checked by Team members for rechecking or reviewing purpose

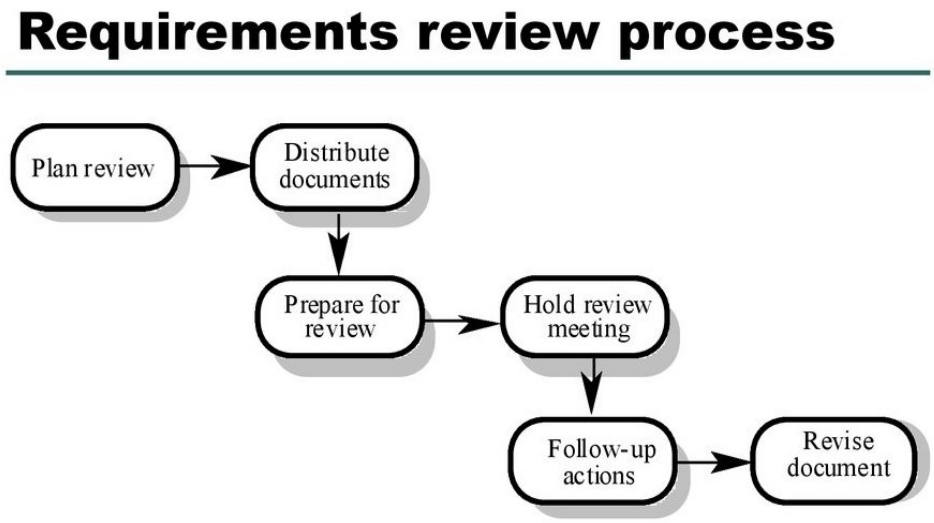


Fig: Requirement Review Process

7.3 Requirement change Management

Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation.

The advantage of using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

It begins from the changes being made in the environment in which finished product is considered to be used, business changes, regulation changes, errors in the original definition of requirements, limitations in the technology, and changes in security environment and so on.

The activities that are included in requirements change management are receiving change requests from the stakeholders, recording the received change requests, analysing and determining the desirability and process of implementation, implementation of change request, and quality assurance for the implementation and closing the change request.

After this the data of the change requests are complied, analysed and appropriate metrics are derived then fit into the organizational knowledge repository.

7.3.1 Principal stages to a change management process:

1. **Problem analysis and change specification**: -

   During this stage, the problem or the change proposal is analysed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal or decide to withdraw the request.

2. **Change analysis and costing**: -

   The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether to proceed with the requirements change.

3. **Change implementation:**

   The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization.

Graded Questions:

1. What are the underlying principles that guide analysis work?
2. Explain Analysis Concept and Principles with neat diagram.
3. What is requirement Management? Explain.
4. Explain Requirements Validation Technique
5. What are different Principal stages to a change management process?

Reference Books:

Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
Software Engineering, Pankaj Jalote Narosa Publication
Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

❖ ❖ ❖

UNIT 3

8

SYSTEM MODELS

Unit Structure:

8.0 Objectives

Objective

In this chapter you will understand how system were created and what are the different element, what are the different system model and how and where this model is useful

8.1 Introduction to System Models

Every computer-based system can be modelled as an information transform using an input-processing-output template.

Using a representation of input, processing, output, user interface processing, and self-test processing, a system engineer can create a model of system components that sets a foundation for later steps in each of the engineering disciplines.

The system engineer allocates system elements to each of five processing regions within the template:

(1) user interface

(2) input

(3) system function and control

(4) output, and (5) maintenance and self-test.

Like nearly all modelling techniques used in system and software engineering, the system model template enables the analyst to create a hierarchy of detail.

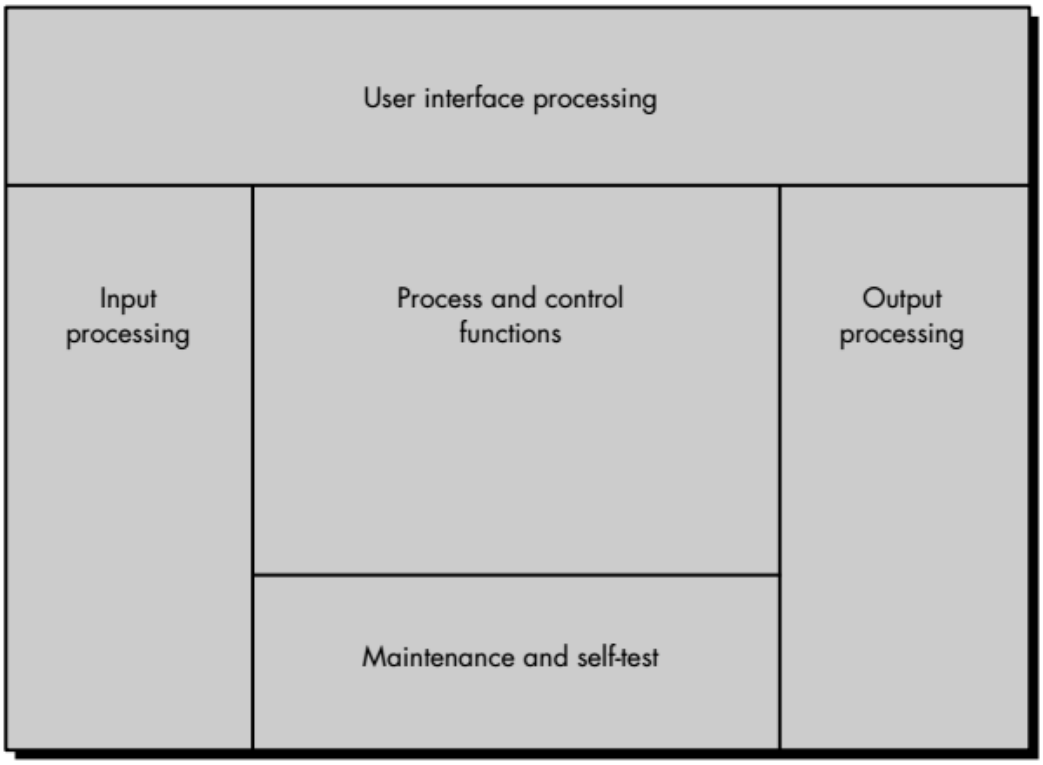


Fig: System Modelling Template

System modelling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

System modelling uses graphical notation, Unified Modelling Language (UML). Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineer implementing the system and after implementation to document the system's structure and operation.

You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.

2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.

   The UML has many diagram types and so supports the creation of many different types of system model.

8.1.1 The three essential elements of system models are

a) Environmental Model:
It defines the scope of the proposed system and its boundaries. It consists of statement of purpose, context diagram and events of the system

b) Behavioural Model:
It describes the functional requirements, internal behaviour and data entities of the system. It consists of ER diagram, DFD, State Transition diagram

c) Implementation Model:
It describes the design specification of the software and consist of software architecture, data design, interface design and component design

d) Structural Model:
It emphasises on modelling the structure of the data that is processed by the system.

8.1.2 Features of Modelling Techniques

1. Easy to use: Can use the model and understand the model working and can give proper inputs and receive the outputs from the system.
2. Contain few about powerful modelling objects to enable easy learning: Structure of the model must be easy to understand and handle properly, so if any error occur end user can perform respective task to remove or avoided the errors
3. Help to handle complexity of the system: Can able to work on all the modules or phases of the system, so can receive the proper output from the system without making any error or mistake during the function of the system.

8.2 Type of System Model

Model are of following types:

- **Data processing Model:** Data processing model showing which component and system are used to processes the data and how the data processed at different stages.
- **Composition Model**: Composition model display where is the interface and connection between the entities and how entities are calm other entities.
- **Architectural Model:** Architectural model shows what are the main components of the system and main sub-systems in the project model.
- **Classification Model:** Classification model showing how entities are interrelated to each other and have common characteristics.
- R**esponse Model:** Response model showing the system's reaction to events.

UML five diagram types could represent the essentials of a system:

1. Activity diagrams, which show the activities involved in a process or in data processing.
2. Use case diagrams, which show the interactions between a system and its environment.
3. Sequence diagrams, which show interactions between actors and the system and between system components.

4. Class diagrams, which show the object classes in the system and the associations between these classes.

5. State diagrams, which show how the system reacts to internal and external events.

8.2.1 Context Models

- At an early stage in the requirements elicitation and analysis process boundaries of the system must be decided involving system stakeholders
- In some cases, the boundary between a system and its environment is relatively clear.

For example, where an automated system is replacing an existing manual or computerized system the environment of the new system is usually the same as the existing system.

whereas in other cases the stakeholders decide the boundary.

- For example, in the library system the user decides the boundary whether to include library catalogues for accessing or not
- Once some decisions on the boundaries of the system has been made part of the activity is definition of that context. Producing a simple architectural model is the first activity.

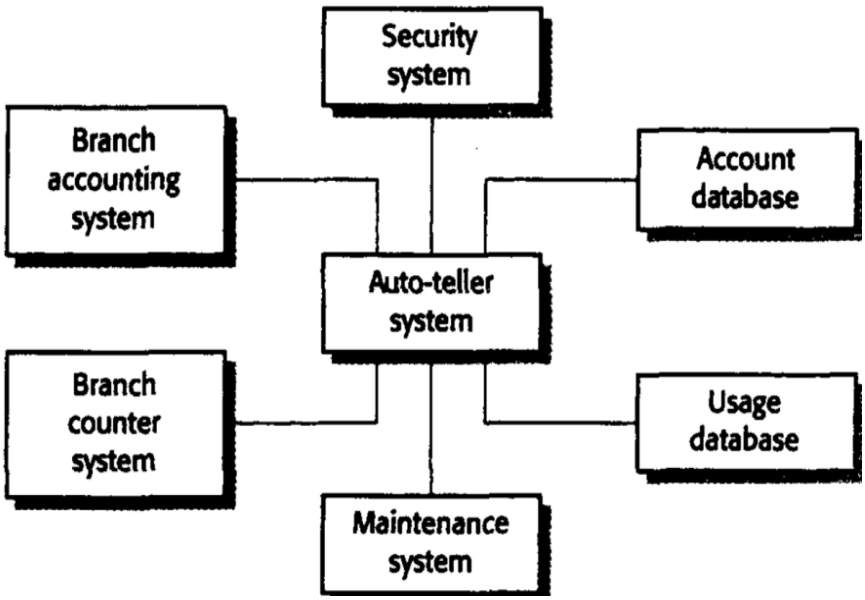


Fig: Context Model for ATM

In the above figure each ATM is connected to account database, local branch accounting system, a security system, and a system to support machine maintenance.

- The system is also connected to usage database that monitors how the networks of ATM is used and to a local branch counter system.
- This counter system provides services such as backup and printing.
- These therefore need not be included in the ATM system itself.
- Architectural models describe the environment of the system but do not show the relationships between the other systems in the environment and the system that is being specified.
- Simple architecture models are supplemented by other models such as process models that show the process activities by the system.

8.2.2 Behavioural Model

- Used to describe the overall behaviour of the system These are of two types

  1. Data Flow Models which model the data processing in the system
  2. State Machine Models which model how the system reacts to events

- Some systems are driven by data, so data flow model is enough to represent their behaviour.
- Real time systems are often driven with minimal data processing and hence state machine model is most effective.

8.2.2.1 Data Flow Models

- An intuitive way of showing how data is processed by a system
- At the analysis level they should be used to model the way in which data is processed in the existing system
- Notations used in this model represents functional processing (rounded rectangles) data stores (rectangles) and data movements between functions (labelled arrows)
- Used to show how data flows through a sequence of processing steps

- The data is transformed or processed before moving to the next step which are known as software processes or functions
- Data flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system.
- They are simple and intuitive and is usually possible to explain them to potential system users.

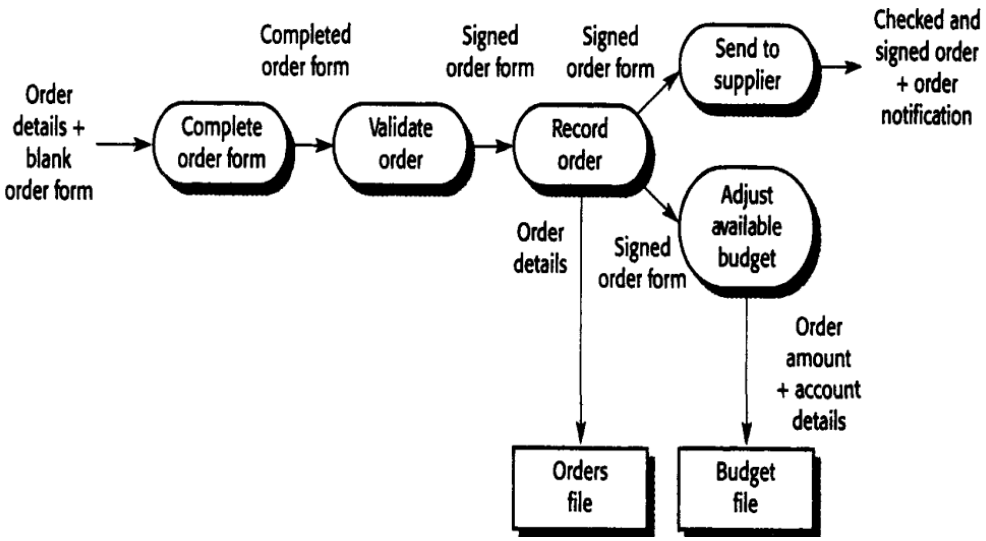


Fig: Data Flow Model for Order Processing

8.2.2.2 State Machine Models

- It describes how a system responds to internal or external events and shows the system states and events that cause transitions from one state to another but does not show the flow of data within the system
- This type of model is often used for modelling real time systems
- These models are an integral part of real time design methods and assumes that at any time the system is in one of the numbers of possible states
- The problem with the state machine approach is that the number of possible states increases rapidly and therefore some structuring is needed.

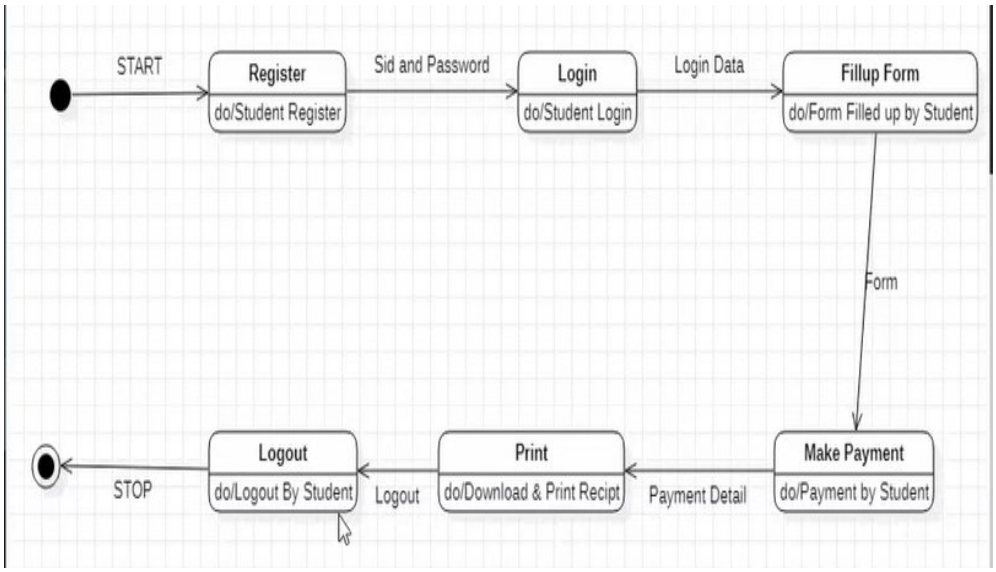


Fig: Form Filling and Payment by Student

In the Given figure we can see that the start state and stop state, start state begin with the registration of the student after registration student will receive his student ID and password, using which he will log in into a system where he will fill up the admission form, for which he had to pay the payment accordingly as done with the payment.

Then need to print the payment slip and field form ask at work is done he can log out from the system and the state will be stopped state machine is nothing, but which shows us of progress in the system.

This diagram can be drawn using star UML software.

8.3 Data Models

1. An important part of system modelling is defining the logical form of data processed by the system. These are called as semantic data models.
2. The most widely used data modelling technique is ERA (Entity-Relation-Attribute) modelling.
3. The relationship models devised from this system are in 3NF and hence they been widely used.
4. Because of the explicit typing and the recognition of the subtypes and super types it is also straight forward to implement these models using object-oriented databases.

5. Data models lack detail and more descriptions of ERA must be maintained by using data dictionaries.
6. Data dictionaries are used to develop system models.
7. It is simply an alphabetical list of names included in model.
8. The advantages of data dictionary are it checks for the uniqueness and warns against name duplications and it stores all data in a single place.
9. Data model is dependent on data, data relationship, data semantics and data constraints constraints.
10. Data model provides the various details such as information to be stored and is of primary use when the final product is created.
11. A data model determines the structure of data explicitly. In the data modeling graphical notation are used and the data model are also specified.
12. Data model organizers element of data and standardize how they relate to one another to the property of real-world entity.
13. Sometime data model can be considered to be a formalization of the object and relationship found in a particular application domain for instance the customer, process products and older found in manufacturing organization.

The following figure is an example of data model.

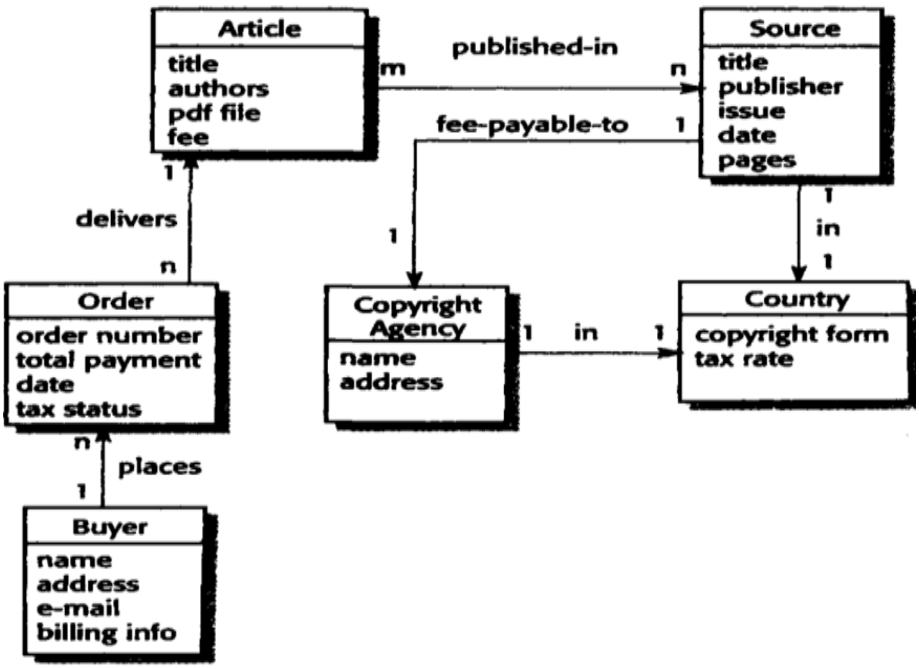


Fig: Semantic Data Model

8.4 Object Model

- Expressing the system requirements using object model, designing using objects and developing using languages like C++ and Java.
- Object models developed during requirements analysis are used to represent both data and its process. They combine some uses of dataflow and semantic models.
- They are also useful for showing how entities in the system may be classified and composed of other entities. Objects are executable entities with attributes and services of the object class and many objects can be created from a class.
- The following diagram shows an object class in UML as a vertically oriented rectangle with three sections – name of the object, class attributes, operations associated with the object.

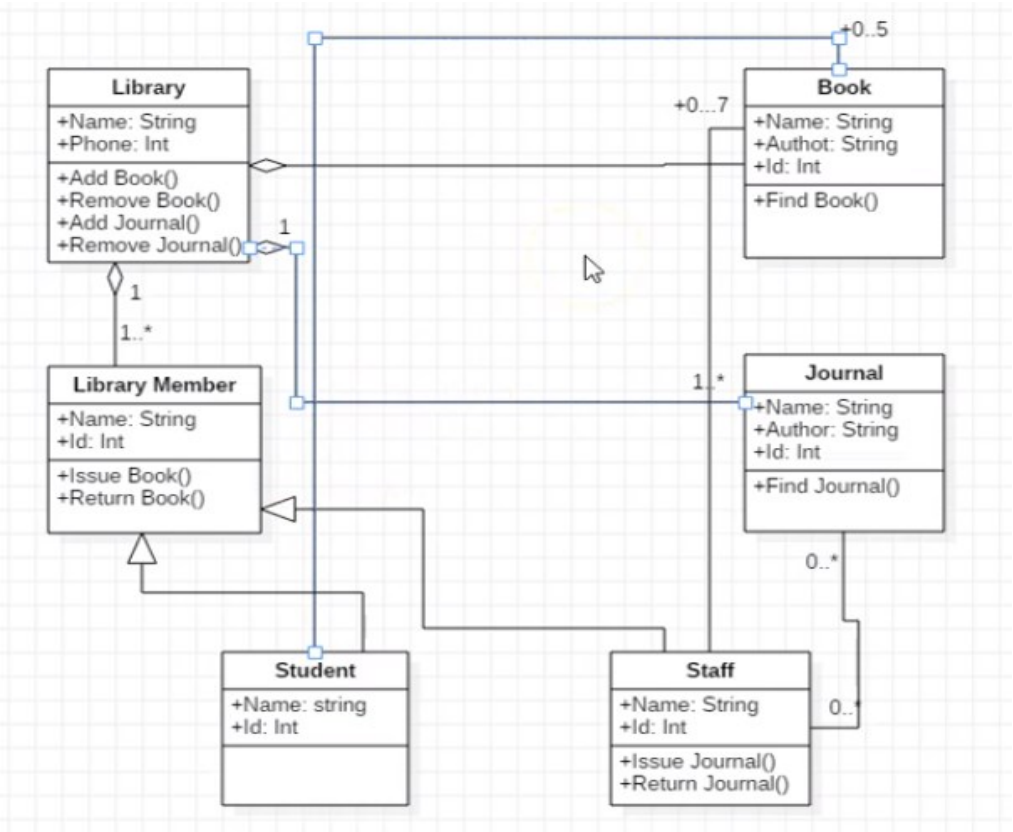


Fig: Library Management System

In the about library management system we can have multiple objects with their attributes and operations for example we have a library class which include attribute like name and phone an operation like add book remove book add Journal remove Journal which help us to provide oral information about that class.

This object model we can call it as inheritance model as well in object oriented modeling it is imperative to identify the class off object that are important in domain that is Bing studied.

The class of the object are then organized into taxonomies and it is nothing but a classification scheme that shows how an object class is related to other class through common attribute and service.

The object class are capable to inherit their attribute and service from one or more yeah super class you can see in the example where library members he's a super class and student and staff are the subclass which inherited the attribute of superclass.

8.5 Structured Methods

- Systematic way of producing models of an existing system or proposed system
- Provide a framework for detailed system modelling as part of requirements elicitation and analysis
- Have their own preferred set of system models and usually define a process that are used to derive these models and set of rules and guidelines that apply to the models
- CASE tools usually used support model editing, coding, report generation and some model checking capabilities
- Have been applied in many large projects because they use standard notations and ensure standard design documentation.
- Suffer from following weakness

  – Do not provide effective support for understanding non-functional requirements

  – Do not include guidelines whether a method is appropriate, nor do they advise on how they can be adapted for a particular project.

  – Produce too much documentation.

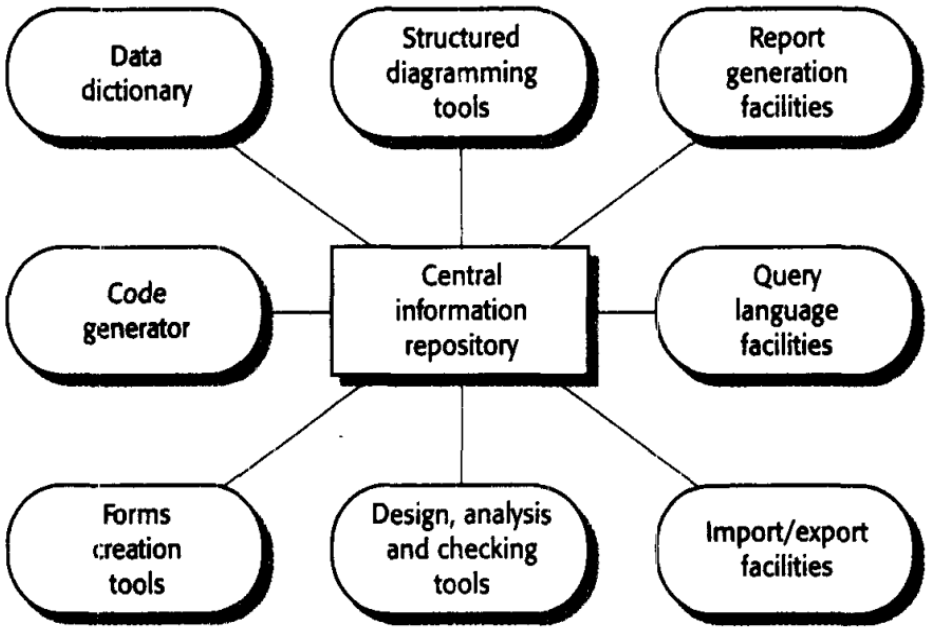


Fig: Central Information Repository

Diagram Editors

These editors are used to create object models, data models, and behavioural models and so on, they are not just drawing tools but are aware of the types of entities in the diagram.

The editors are capable of capturing information about these entities and save this information in the central repository.

Design Analysis and checking tools

These tools process the design and report on errors and anomalies.

They may be integrated with the editing system so that user errors are trapped at an early stage in the process.

Repository Query Languages

These languages facilitate the designers determine designs to and associated design information in the repository.

Data Dictionary

It has responsibility of maintaining information about the entities used in a system design.

Report Definition and generation tools

These tools accept information from the central store and automatically generate system documentation.

Forms definition tools

These tools facilitate screen and document formats to be specified.

Import/Export Facilities

These facilities allow the interchange of information the central repository with other development tools

Code generators

The code generator are responsible for generating code or code skeletons automatically from the design captured in the central store.

8.6 Interaction Model

All systems involve interaction of some kind.

This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system.

Interaction Model helps in, Modelling user interaction is important as it helps to identify user requirements.

Modelling system to system interaction highlights the communication problems that may arise.

Modelling component interaction helps us understand if a proposed system structure.

There are two related approaches to interaction modelling:

1. Use case modelling, which is mostly used to model interactions between a system and external actors (users or other systems).

2 Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together.

The details of the interactions involved in a high-level use case may be documented in a sequence diagram.

The UML also includes communication diagrams that can be used to model interactions.

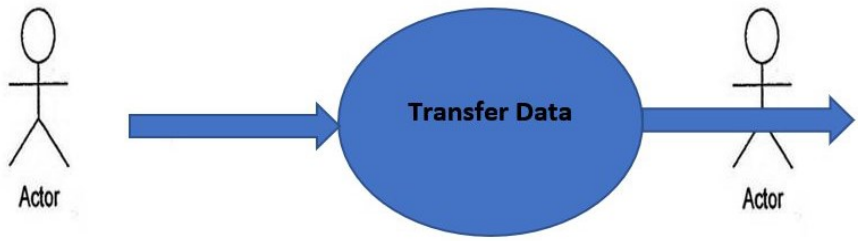
We won't discuss these here as they are an alternative representation of sequence charts. In fact, some tools can generate a communication diagram from a sequence diagram.

8.7 Use case Modelling

use case modelling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

Each use case represents a discrete task that involves external interaction with a system.

In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.



Medical Receptionist System

Patient Record

Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction, but it is also now used to represent other external systems and hardware.

Use case diagrams give a simple overview of an interaction so you have to provide more detail to understand what is involved.

Can use case modeling the actor there are two types of actor when who initiate and another who will react to the system there must be at least one interaction with the use cases with actors the initiator always keep left side of the system and reaction actor always keep right side of the system.

The use cases in the system must be arrange in proper logical order only so that actor can have a proper interaction with use cases within the system

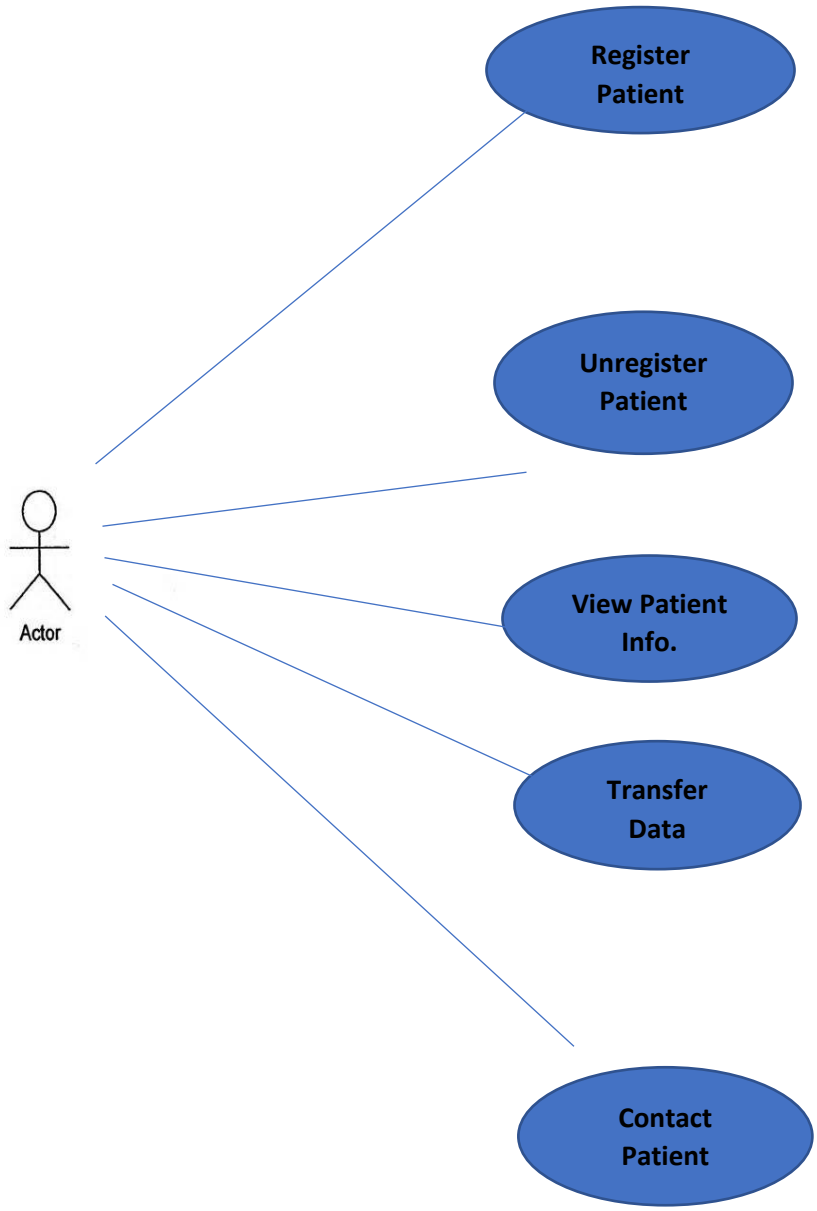


Fig: Use case involving the role 'medical receptionist'

8.8 Sequence diagram

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves.

The UML has a rich syntax for sequence diagrams, which allows many kinds of interaction to be modelled. I don't have space to cover all possibilities here, so I focus on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance.

Below Figure is an example of a sequence diagram that illustrates the basics of the notation.

This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information

The objects and actors involved are listed along the top of the diagram, with a dot ted line drawn vertically from these.

Interactions between objects are indicated by annotated arrows.

The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation).

You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. In this example, we can see the notation used to denote alternatives.

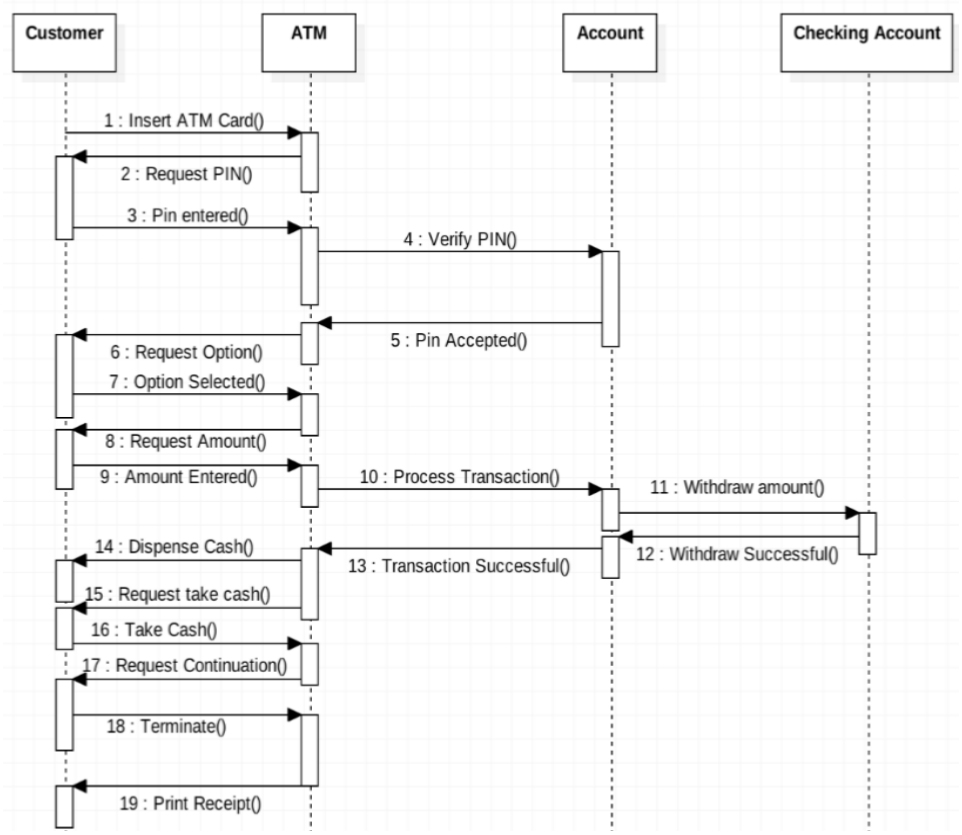


Fig: Sequence Diagram

In the given sequence diagram, we can see the sequence of steps to complete the process in ATM system.

Graded Question

1. What are the advantages of Architectural Design? Which factors are dependable during the design?
2. What is the design perspective of architectural design? Explain.
3. What are the three system organization styles of architectural design? Explain in brief.
4. Explain use case diagram with neat diagram.
5. Write a short note on object model.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

❖ ❖ ❖

UNIT 3

9

ARCHITECTURAL DESIGN

Unit Structure:

9.0 Objectives

1. Understand the concept of software architecture
2. Understanding the design
3. Know the architectural patterns

9.1 Introduction to Architectural design

Architectural design is concerned with understanding how a system should be organised and design overall structure of system. In the model of software development process architectural design is the first stage in software design process.

Application systems are intended to meet a business or organizational need. All businesses have much in common-they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector specific applications.

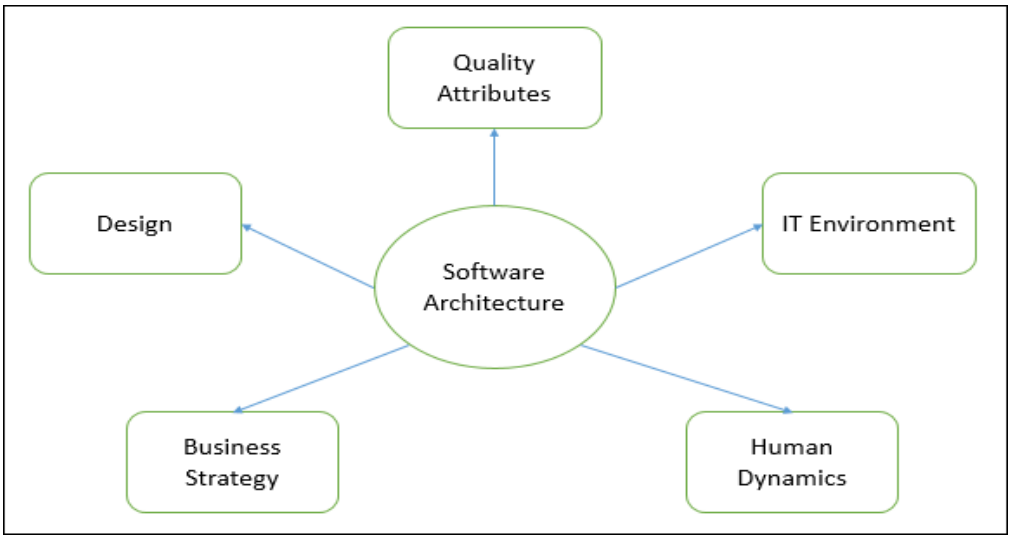
Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architect may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

9.1.1 Architectural Design

The architecture of a system describes its major components, their relationships structures, and how they interact with each other. Architectural design is also called as high-level design.

Software architecture and design includes several related factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In Architecture, non-functional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.

9.1.2 Software architectures can be designed at two levels of concept:

- Architecture in the small is concerned with the architecture of individua programs. At this level, we are concerned with the way that an individua program is decomposed into components.
- Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components.

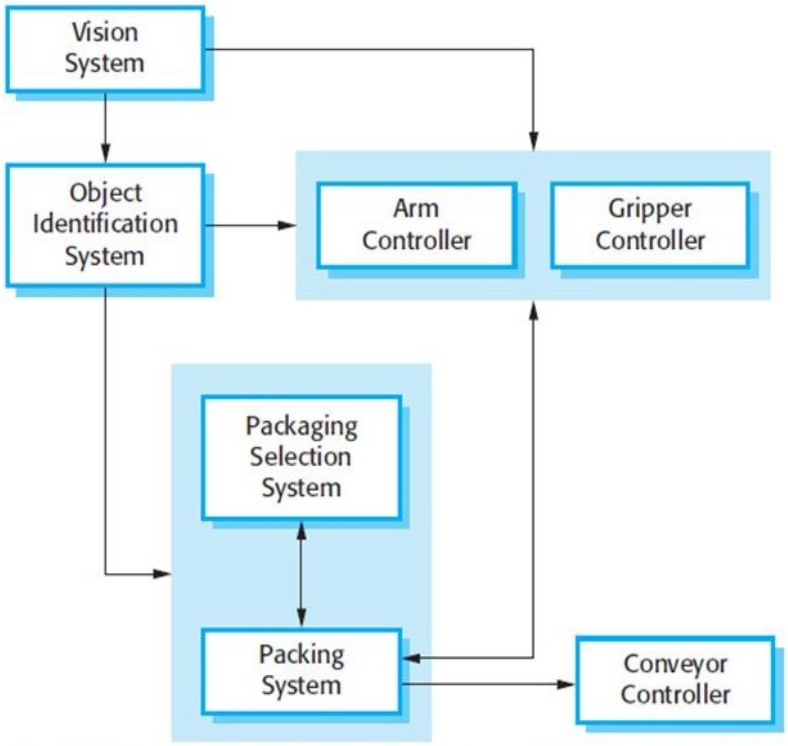


Fig: The architecture of a packing robot control system

There are Three advantages of explicitly designing and documenting software architecture:

1. **Stakeholder** communication the architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. **System analysis** Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether the system can meet critical requirements such as performance, reliability, and maintainability.

3. **Large-scale reuse** A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

The apparent contradictions between practice and architectural theory a because there are two ways in which an architectural model of a program is used:

1. As a way of facilitating discussion about the system design A high-level architectural view of a system is useful for communication with system stake holders and project planning because it is not cluttered with detail. Stake holders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.

2. As a way of documenting an architecture that has been designed The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

9.2 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the follow ing fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?

6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Because of the close relationship between non-functional requirements and software architecture, the architectural style and structure that you choose for a system should depend on the non-functional system requirements:

9.2.1 The various attributes of architecture are as follows:

1. Performance

If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of com ponents, with these components all deployed on the same computer rather than distributed across the network.

This may mean using a few relatively large com ponents rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.

2. Security

If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.

3. Safety

If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.

4. Availability

If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems.

5. Maintainability If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may readily be changed.

9.3 Architectural View

There are different opinions as to what view are required, some of the view that are suggested.

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.

3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.

4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

9.3.1 Architectural Design Processes are as follows

1. System Structuring
2. Control Modelling
3. Modular Decomposition

9.4 System Organization

System Organization reflects the basic strategies that is used to structure a system.

Three organisational styles are widely used:

9.4.1 Shared data repository style: Sub-System must exchange the data.
- Shared data is held in a central database
- Each sub-system maintains its own database and pass it to other sub-system

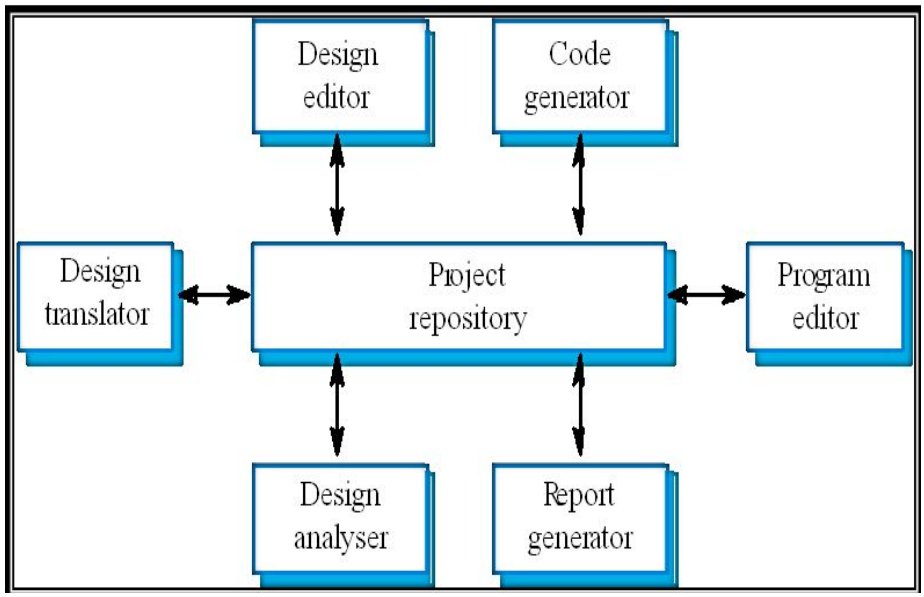


Fig: A repository architecture for an IDE

9.4.2 A shared service and server's style (Client-Server Model)

It is a distributed system model which shows how data and processing is distributed across a range of components.

A system that follows the client-server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major com ponents of this model are:

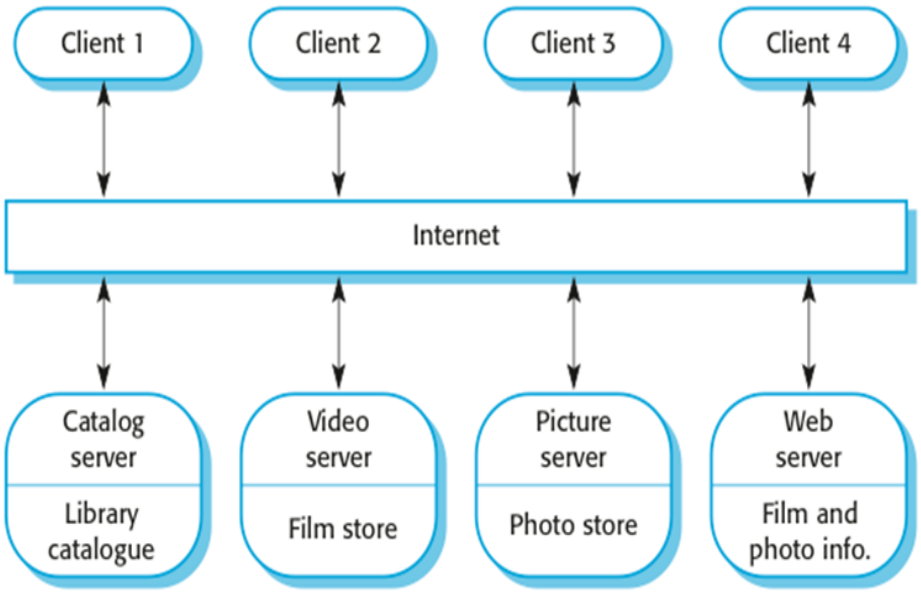
1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.

2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.

3. A network that allows the clients to access these services. Most client-server systems are implemented as distributed systems, connected using Internet protocols.

   Client-server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of
   the system. Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services.

Clients access the services provided by a Server through remote procedure calls using a request-reply protocol such as the http protocol used in WWW.

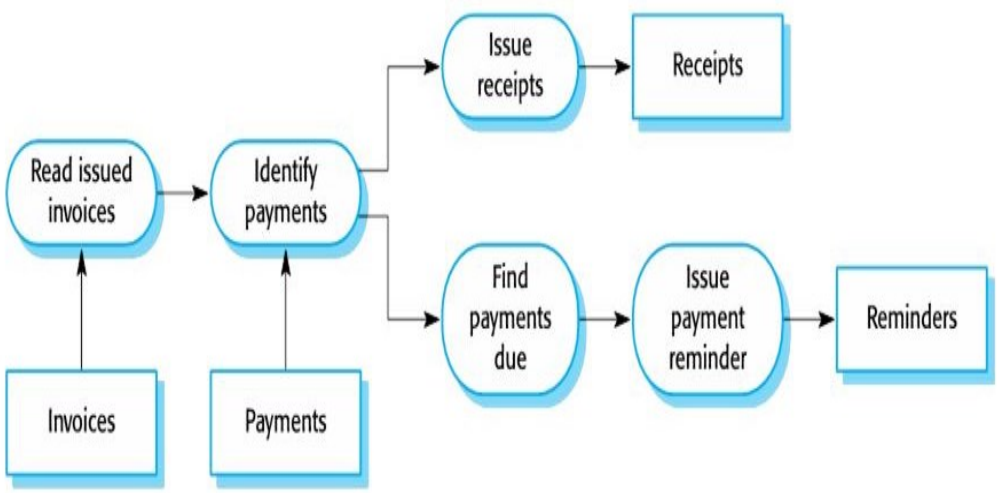


9.4.3 Pipe and Filter architecture

This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform, Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name 'pipe and filter come from the original Unix system where it was possible to link processes using 'pipes', These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term 'filter is used because a transformation filters out the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data processing systems (e.g., a billing system). The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently.



An example of this type of system architecture, used in a batch processing application, is shown in above Figure. An organization has issued invoices to customers.

Once a week, payments that have been made are reconciled with the invoices.

For those invoices that have been paid; a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed.

Although simple textual input and output can be modelled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections.

It is difficult to translate this into a form compatible with the pipelining model.

9.4.4 An abstract machine or layered style.

It is used to organise the system systematically into layers.

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. separates elements of a system, allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The layered architecture pattern is another way of achieving separation and independence.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer.

As layered systems localize machine dependencies in inner layers. this makes it easier to provide multi-platform implementations of an application system. Only

the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

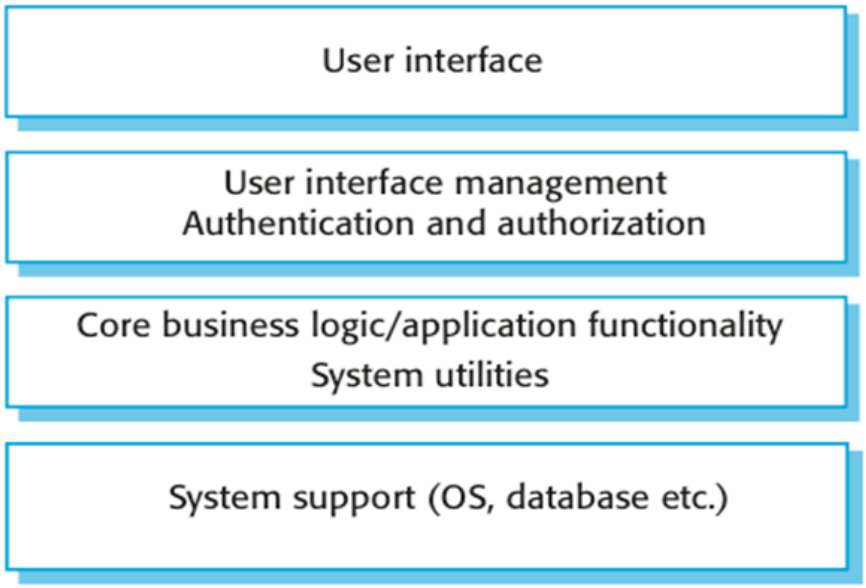


Fig: A generic layered architecture

Above Figure is an example of a layered architecture with four layers.

The lowest layer includes system support software-typically database and operating system support.

The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components.

The third layer is concerned with user interface management and providing user authentication and authorization, with the top layer providing user interface facilities.

Of course, the number of layers is arbitrary. Any of the layers in Figure could be split into two or more layers.

9.5 Modular Decomposition Styles

Two modular decomposition models covered

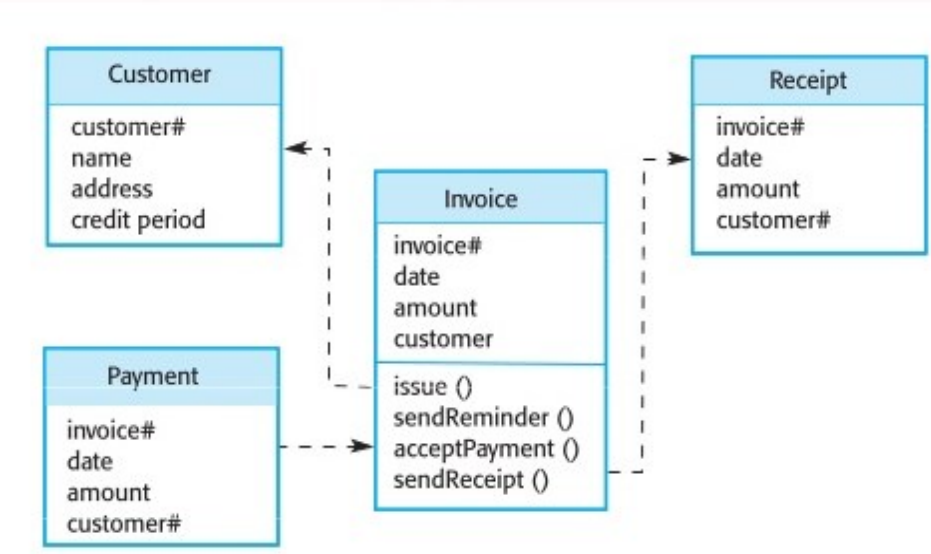
1. An object model where the system is decomposed into interacting objects
2. A data -flow model where the system is decomposed into functional modules which transform inputs to outputs. Also known as the pipeline model.

If possible, decisions about concurrency should be delayed until modules are implemented

9.5.1 Object -oriented decomposition:

1. Structure the system into a set of loosely coupled objects with well -defined interfaces
2. Object-oriented decomposition is concerned with identifying object classes, their attributes, and operations
3. When implemented, objects are created from these classes and some control model used to coordinate object operations.

9.5.2 Invoice processing system



Advantages:

1. Objects are loosely coupled, the implementation of the objects are modify without affecting other objects.
2. Objects can be reused.
3. Direct implementation of architectural components.

Disadvantages:

1. To use services, objects must explicitly reference the name and the interface of other objects.
2. Interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated.

9.5.3 Function Oriented Pipelining or Data flow model:

1. Functional transformation processes their inputs to produce outputs.

2. Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.

3. It may be referred to as a pipe and filter model.

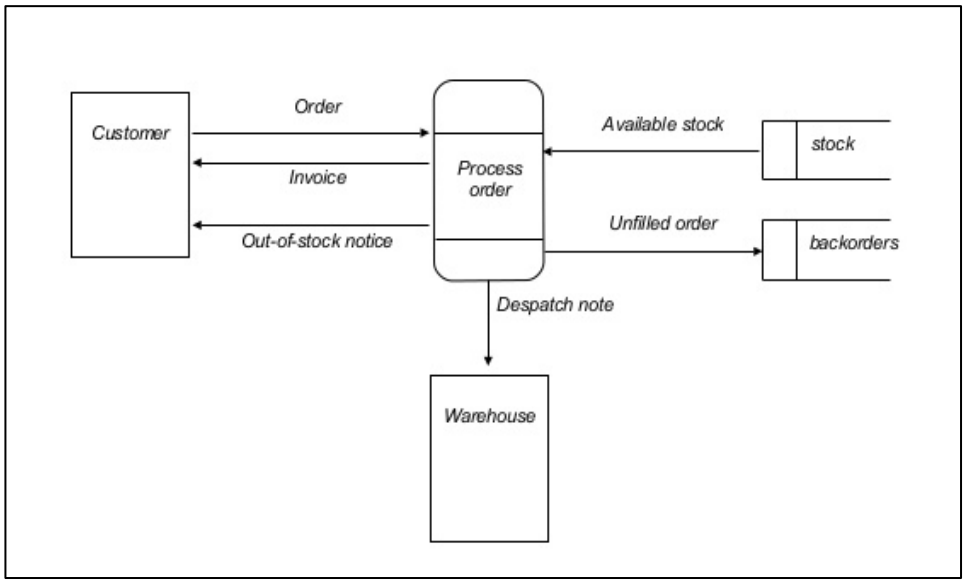


Fig: Data Flow Model

9.6 Reference architectures

- Reference models are derived from a study of the application domain rather than from existing systems.
- May be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated.
- OSI model is a layered model for communication systems

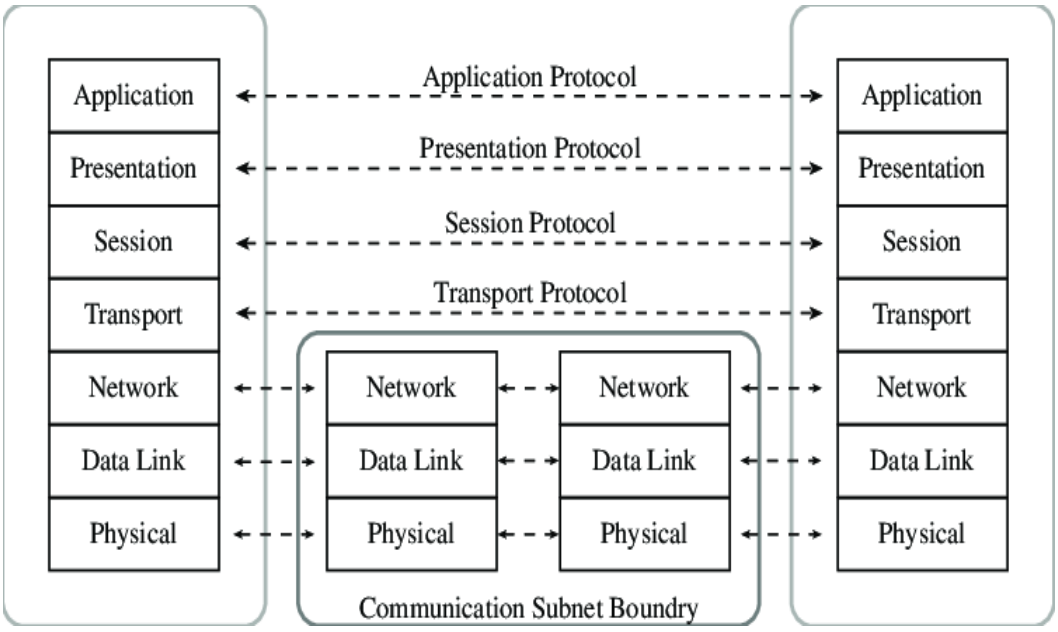


Fig: OSI Reference Model

- The software architect is responsible for deriving a structural system model, a control model, and a sub-system decomposition model.
- Large systems rarely conform to a single architectural model

9.7 Application architecture

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

9.7.1 As a software designer, you can use models of application architectures in several Ways which are as follows.

1. As a starting point for the architectural design process If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.

2. As a design checklist If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.

3. As a way of organizing the work of the development team. The application architectures identify stable structural features of the system architectures

and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.

4. As a means of assessing components for reuse If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. As a vocabulary for talking about types of applications If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

9.7.2 There are many types of application system, there are architecture of two type of application.

1. Transaction processing applications:

Transaction processing applications are database-cantered applications that process user requests for information and update the information in a database.

These are the most common type of inter active business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained.

These systems include e-commerce system, information systems, and booking system

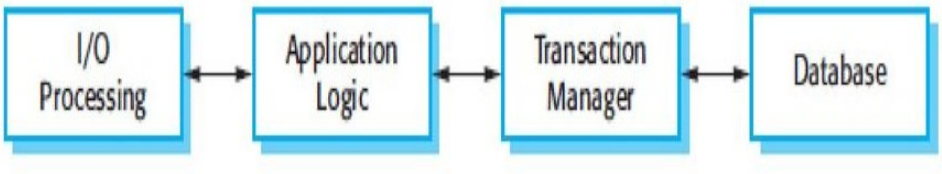


Fig: The structure of transaction processing application

2. Language processing systems:

Language processing systems are systems in which the user's intentions are expressed in a formal language (such as Java). The language processing system processes this language into an internal format and then interprets this internal representation.

The best-known language pro cessing systems are compilers, which translate high-level language programs into machine code.

However, language processing systems are also used to interpret command languages for databases and information systems, and mark-up languages such as XML.

Transaction Processing Systems

Transaction processing system are designed to process user request for information from a database, or request to update database.

a database transaction is a sequence of operation that is treated as a signal single unit, all of the operation in a transaction have to be completed before the database changes and made permanent. This ensures that failure of operation within the transaction does not lead to inconsistency in the database.

Hey for a user perspective a transaction is any coherent sequence of operation that satisfy a goal, It the user transaction does not required the database to be changed then it may not be necessary to package this as a technical database transaction

For example, a transaction is a customer request to withdraw money from a bank account using an ATM.

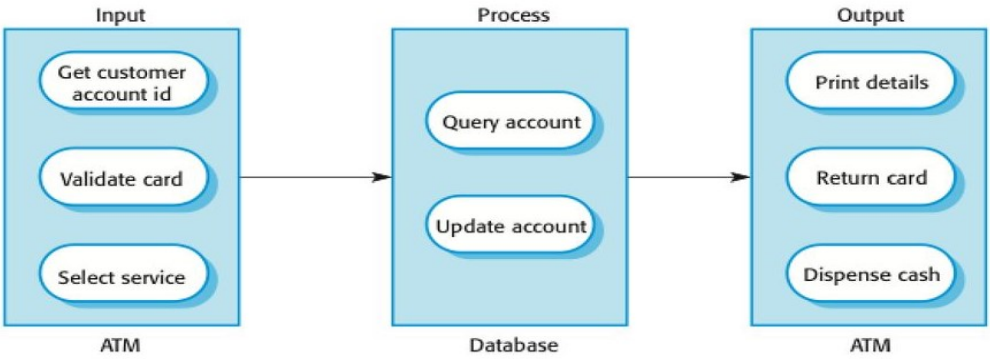


Fig: The Software architecture of ATM system

Transaction processing system may be organised as pipe and filter architecture with system components responsible for input, processing, and output.

The system is composed of 2 component cooperating software component the ATM software and the account processing software in the bank database server.

Summary

Requirement for a software system set out what the system should do and define constraints on its operation and implementation. The Specifications determining part for project success or failure.

Graded Question

1. What are the advantages of Architectural Design? Which factors are dependable during the design?
2. What are the three system organization styles of architectural design? Explain in brief.
3. What is the design perspective of architectural design? Explain.
4. What are the three system organization styles of architectural design? Explain in brief.
5. Explain Object oriented decomposition.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

❖ ❖ ❖

UNIT 4

10

APPLICATION ARCHITECTURE

Unit Structure:

10.0 Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- be introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- Know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems..

10.1 Introduction

Application systems are intended to meet a business or organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector specific applications. Therefore, as well as general business functions, all phone companies need systems to connect calls, manage their network, issue bills to customers, etc. Consequently, the application systems used by these businesses also have much in common. These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software system. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type. The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re-implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements. As a software designer, one can use models of application architectures in a number of ways:

1. As a starting point for the architectural design process If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.
2. As a design checklist If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.
3. As a way of organizing the work of the development team The application architecture identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.

4. As a means of assessing components for reuse If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. As a vocabulary for talking about types of applications If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications. There are many types of application system and, in some cases, they may seem to be very different. However, many of these superficially dissimilar applications actually have much in common, and thus can be represented by a single abstract application architecture. One can illustrate this here by describing the following architectures of two types of application:

    a) Transaction processing applications Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common type of interactive business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained. This class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

    b) Language processing systems Language processing systems are systems in which the user's intentions are expressed in a formal language (such as Java). The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers, which translate high-level language programs into machine code. However, language processing systems are also used to interpret command languages for databases and information systems, and markup languages such as XML (Harold and Means, 2002; Hunter et al., 2007).

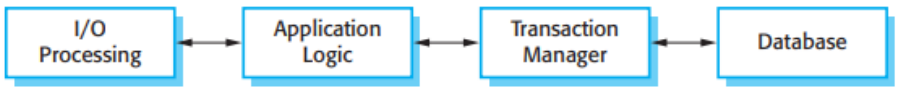


Figure 10.1 The structure of transaction processing applications

It has been chosen these particular types of system because a large number of web-based business systems are transaction-processing systems, and all software development relies on language processing systems.

10.1.1 Transaction processing systems

Transaction processing (TP) systems are designed to process user requests for information from a database, or requests to update a database. Technically, a database transaction is sequence of operations that is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within the transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as 'find the times of flights from London to Paris'. If the user transaction does not require the database to be changed then it may not be necessary to package this as a technical database transaction.

An example of a transaction is a customer request to withdraw money from a bank account using an ATM. This involves getting details of the customer's account, checking the balance, modifying the balance by the amount withdrawn, and sending com mands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed. Transaction processing systems are usually interactive systems in which users make asynchronous requests for service.

Figure 10.2 illustrates the conceptual architectural structure of TP applications. First a user makes a request to the system through an I/O processing component. The request is processed by some application specific logic. A transaction is created and passed to a transaction manager, which is usually embedded in the database management system. After the transaction manager has ensured that the transaction is properly completed, it signals to the application that processing has finished.

Transaction processing systems may be organized as a 'pipe and filter' architecture with system components responsible for input, processing, and output.

For example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server. The input and output

components are implemented as software in the ATM and the processing component is part of the bank's database server.

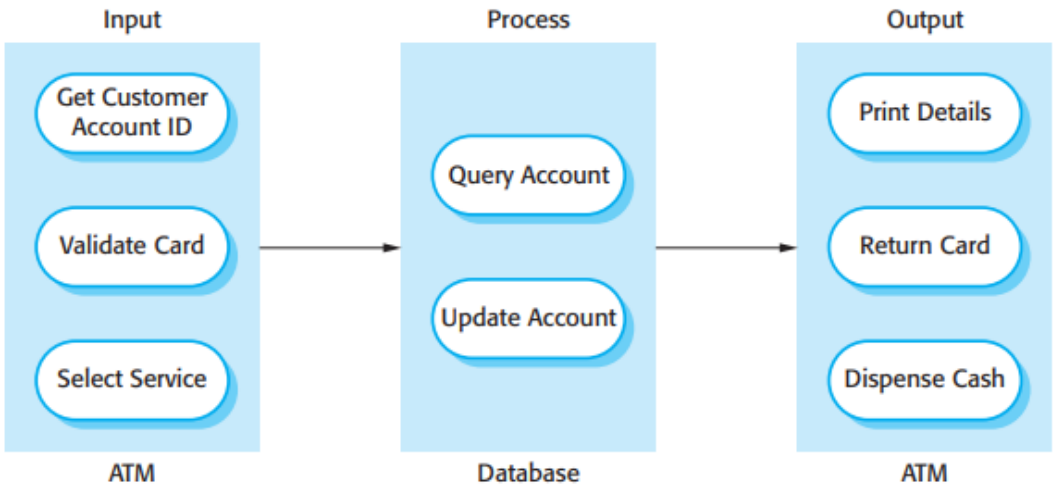


Figure 10.2 The software architecture of an ATM system

10.1.2 Language processing systems

Language processing systems translate a natural or artificial language into another representation of that language and, for programming languages, may also execute the resulting code. In software engineering, compilers translate an artificial programming language into machine code. Other language-processing systems may translate an XML data description into commands to query a database or to an alternative XML representation. Natural language processing systems may translate one natural language to another e.g., French to Norwegian. A possible architecture for a language processing system for a programming language is illustrated in Figure 10.3. The source language instructions define the program to be executed and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

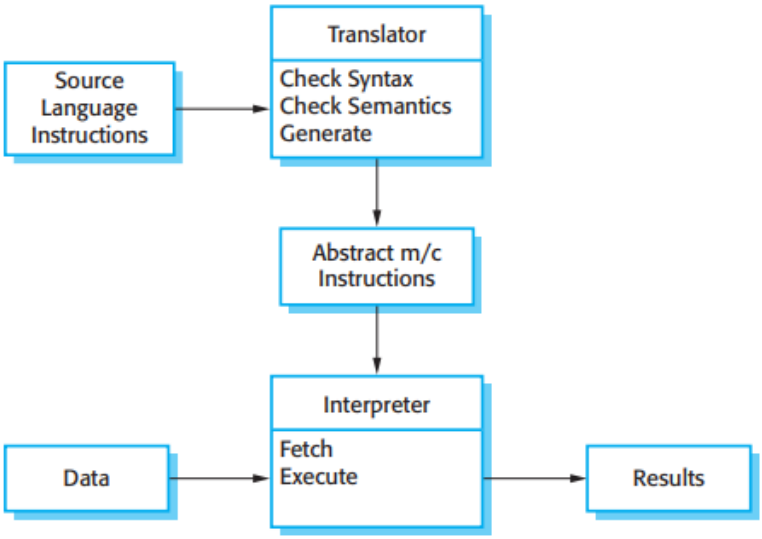


Figure 10.3 The architecture of a language processing system

Of course, for many compilers, the interpreter is a hardware unit that processes machine instructions and the abstract machine is a real processor. However, for dynamically typed languages, such as Python, the interpreter may be a software component. Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 10.4) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them to an internal form.
2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
4. A syntax tree, which is an internal structure representing the program being compiled.

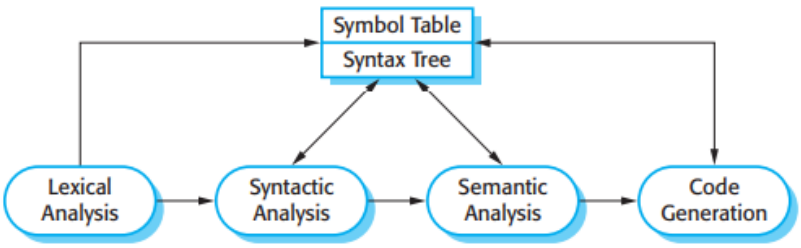


Figure 10.4 A pipe and filter compiler architecture

5. A semantic analyzer that uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.

6. A code generator that ‘walks’ the syntax tree and generates abstract machine code.

Other components might also be included which analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code. In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary, and the generated code is actually the input text translated into another language.

There are alternative architectural patterns that may be used in a language processing system. Compilers can be implemented using a composite of a repository and a pipe and filter model. In compiler architecture, the symbol table is a repository for shared data. The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown in Figure 10.4, and communicate through the shared symbol table. This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another. It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger or a program pretty printer. In this situation, changes from one component need to be reflected immediately in other components. It is better, therefore, to organize the system around a repository, as shown in Figure 10.5 . This figure illustrates how a language processing system can be part of an integrated set of programming support tools. In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository. Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed and a pretty printer can create listings of the program in a format that is easy to read.

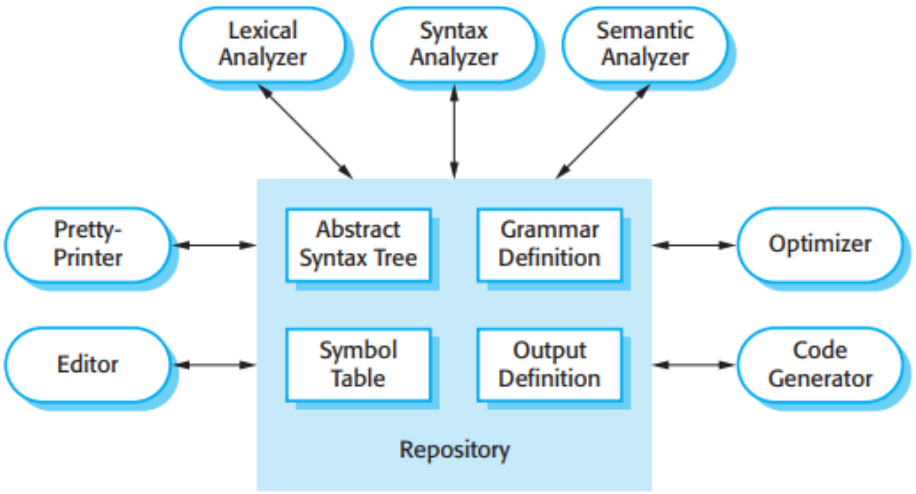


Figure 10.5 A repository architecture for a language processing system

10.2 Data processing System

Systems that are data-centered where the databases used are usually orders of magnitude larger than the software itself.

a) Data is input and output in batches

- Input: A set of customer numbers and associated readings of an electricity meter;
- Output: A corresponding set of bills, one for each customer number.

b) Data processing systems usually have an input-process-output structure.

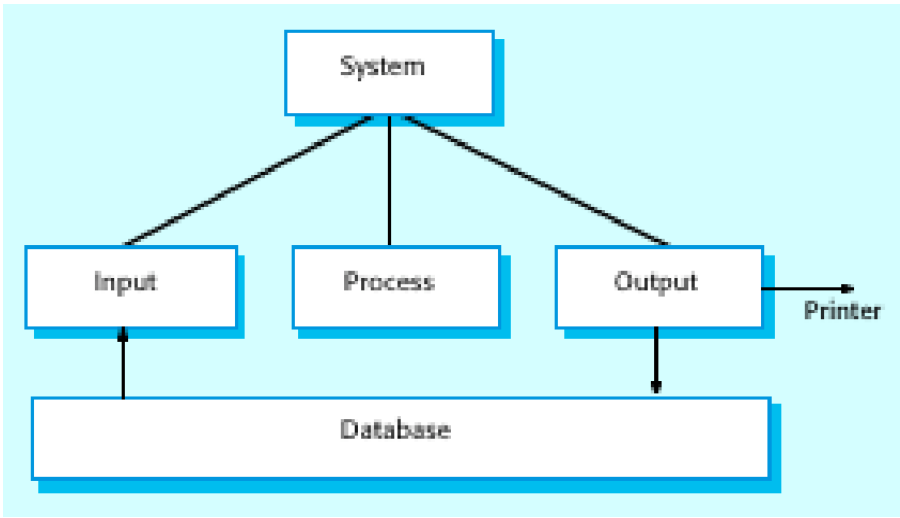


Figure 10.6 Input-process-output model

10.4.1 Input-process-output Model

The input-process-output model has historically been the dominant approach to understanding and explaining team performance and continues to exert a strong influence on group research today. The framework is based on classic systems theory, which states that the general structure of a system is as important in determining how effectively it will function as its individual components. Similarly, the IPO model has a causal structure, in that outputs are a function of various group processes, which are in turn influenced by numerous input variables. In its simplest form, the model is depicted as the following:

Input —> Process —> Output

Inputs

Inputs reflect the resources that groups have at their disposal and are generally divided into three categories: individual-level factors, group-level factors, and environmental factors. Individual-level factors are what group members bring to the group, such as motivation, personality, abilities, experiences, and demographic attributes. Examples of group-level factors are work structure, team norms, and group size. Environmental factors capture the broader context in which groups operate, such as reward structure, stress level, task characteristics, and organizational culture.

Processes

The mediating mechanisms that convert inputs to outputs are known as processes. A key aspect of the definition is that processes represent interactions that take place among team members. Many different taxonomies of teamwork behaviors have been proposed, but common examples include coordination, communication, conflict management, and motivation.

In comparison with inputs and outputs, group processes are often more difficult to measure, because a thorough understanding of what groups are doing and how they complete their work may require observing members while they actually perform a task. This may lead to a more accurate reflection of the true group processes, as opposed to relying on members to self-report their processes retrospectively. In addition, group processes evolve over time, which means that they cannot be adequately represented through a single observation. These difficult methodological issues have caused many studies to ignore processes and focus only on inputs and outputs. Empirical group research has therefore been criticized as treating processes as a “black box” (loosely specified and

unmeasured), despite how prominently featured they are in the IPO model. Recently, however, a number of researchers have given renewed emphasis to the importance of capturing team member interactions, emphasizing the need to measure processes longitudinally and with more sophisticated measures.

Outputs

Indicators of team effectiveness have generally been clustered into two general categories: group performance and member reactions. Group performance refers to the degree to which the group achieves the standard set by the users of its output. Examples include quality, quantity, timeliness, efficiency, and costs. In contrast, member reactions involve perceptions of satisfaction with group functioning, team viability, and personal development. For example, although the group may have been able to produce a high-quality product, mutual antagonism may be so high that members would prefer not to work with one another on future projects. In addition, some groups contribute to member well-being and growth, whereas others block individual development and hinder personal needs from being met.

Both categories of outcomes are clearly important, but performance outcomes are especially valued in the teams literature. This is because they can be measured more objectively (because they do not rely on team member self-reports) and make a strong case that inputs and processes affect the bottom line of group effectiveness.

Steiner's Formula

Consistent with the IPO framework, Ivan Steiner derived the following formula to explain why teams starting off with a great deal of promise often end up being less than successful:

Actual productivity = potential productivity – process loss

Although potential productivity is the highest level of performance attainable, a group's actual productivity often falls short of its potential because of the existence of process loss. Process loss refers to the suboptimal ways that groups operate, resulting in time and energy spent away from task performance. Examples of process losses include group conflict, communication breakdown, coordination difficulty, and social loafing (group members shirking responsibility and failing to exert adequate individual effort). Consistent with the assumptions of the IPO model, Steiner's formula highlights the importance of group processes and reflects the notion that it is the processes and not the inputs (analogous to

group potential) that create the group's outputs. In other words, teams are a function of the interaction of team members and not simply the sum of individuals who perform tasks independently.

Limitations of the IPO Model

The major criticism that has been levied against the IPO model is the assumption that group functioning is static and follows a linear progression from inputs through outputs. To incorporate the reality of dynamic change, feedback loops were added to the original IPO model, emanating primarily from outputs and feeding back to inputs or processes. However, the single-cycle, linear IPO path has been emphasized in most of the empirical research. Nevertheless, in both theory and measurement, current team researchers are increasingly invoking the notion of cyclical causal feedback, as well as nonlinear or conditional relationships.

Although the IPO framework is the dominant way of thinking about group performance in the teams literature, relatively few empirical studies have been devoted to the validity of the model itself. In addition, research directly testing the input-process-output links has frequently been conducted in laboratory settings, an approach that restricts the number of relevant variables that would realistically occur in an organization. However, although the IPO model assumes that process fully mediates the association between inputs and outputs, some research has suggested that a purely mediated model may be too limited. Therefore, alternative models have suggested that inputs may directly affect both processes and outputs.

10.4.2 Data-flow diagrams

DFD is the abbreviation for Data Flow Diagram. The flow of data of a system or a process is represented by DFD. It also gives insight into the inputs and outputs of each entity and the process itself. DFD does not have control flow and no loops or decision rules are present. Specific operations depending on the type of data can be explained by a flowchart. Data Flow Diagram can be represented in several ways. The DFD belongs to structured-analysis modeling tools. Data Flow diagrams are very popular because they help us to visualize the major steps and data involved in software-system processes.

Components of DFD

The Data Flow Diagram has 4 components:

Process

Input to output transformation in a system takes place because of process function. The symbols of a process are rectangular with rounded corners, oval, rectangle or a circle. The process is named a short sentence, in one word or a phrase to express its essence

DataFlow

Data flow describes the information transferring between different parts of the systems. The arrow symbol is the symbol of data flow. A relatable name should be given to the flow to determine the information which is being moved. Data flow also represents material along with information that is being moved. Material shifts are modeled in systems that are not merely informative. A given flow should only transfer a single type of information. The direction of flow is represented by the arrow which can also be bi-directional.

Warehouse

The data is stored in the warehouse for later use. Two horizontal lines represent the symbol of the store. The warehouse is simply not restricted to being a data file rather it can be anything like a folder with documents, an optical disc, a filing cabinet. The data warehouse can be viewed independent of its implementation. When the data flow from the warehouse it is considered as data reading and when data flows to the warehouse it is called data entry or data update.

Terminator

The Terminator is an external entity that stands outside of the system and communicates with the system. It can be, for example, organizations like banks, groups of people like customers or different departments of the same organization, which is not a part of the model system and is an external entity. Modeled systems also communicate with terminator.

Different kinds/levels of data-flow diagrams

There are three main types of data-flow diagram:

Context diagrams — context diagram DFDs are diagrams that present an overview of the system and its interaction with the rest of the “world”.

Level 1 data-flow diagrams — Level 1 DFDs present a more detailed view of the system than context diagrams, by showing the main sub-processes and stores of data that make up the system as a whole.

Level 2 (and lower) data-flow diagrams — a major advantage of the data-flow modelling technique is that, through a technique called "levelling", the detailed complexity of real world systems can be managed and modeled in a hierarchy of abstractions. Certain elements of any dataflow diagram can be decomposed ("exploded") into a more detailed model a level lower in the hierarchy.

10.2 Event processing systems

1. Event processing systems respond to events in the system's environment or user interface. The key characteristic of event processing systems is that the timing of events is unpredictable and the system must be able to cope with these events when they occur.

2. We all use such event-based systems like this on our own computers—word processors, presentation systems and games are all driven by events from the user interface. The system detects and interprets events. User interface events represent implicit commands to the system, which takes some action to obey that command. For example, if you are using a word processor and you double-click on a word, the double-click event means 'select that word'.

3. Real-time systems, which take action in 'real time' in response to some external stimulus, are also event-based processing systems. However, for real-time systems, events are not usually user interface events but events associated with servers or actuators in the system. Because of the need for real-time response to unpredictable events, these real-time systems are normally organised as a set of cooperating processes.

Editing systems are programs that run on PCs or workstations that allow users to edit documents such as text documents, diagrams or images. Some editors focus on editing a single type of document, such as images from a digital camera or scanner. Others, including most word processors, are multi-editors and include support for editing different types including text and diagrams. You can even think of a spreadsheet as an editing system where you edit boxes on the sheet. Of course, spreadsheets have additional functionality to carry out computations.

Editing systems have a number of characteristics that distinguish them from other types of system and that influence their architectural design:

1. Editing systems are mostly single-user systems. They therefore don't have to deal with the problems of multiple concurrent access to data and have

simpler data management than transaction-based systems. Even where data are shared, transaction management is not usually used because transactions take a long time and alternative methods of maintaining data integrity are used.

2. They have to provide rapid feedback on user actions such as ‘select’ and ‘delete’. This means they have to operate on representations of data that is held in computer memory rather than on disk. Because the data is in volatile memory, it can be lost if there is a system fault, so editing systems should make some provision for error recovery.

3. Editing sessions are normally much longer than sessions involving ordering goods, or making some other transaction. This again means that there is a greater risk of loss if problems arise. Therefore, many editing systems include recovery facilities that automatically save work in progress and recover this for the user in the event of a system failure.

10.4 Summary:

- Generic models of application systems architectures help us understand the operation of applications, compare applications of the same type, validate application system designs, and assess large-scale components for reuse.
- Transaction processing systems are interactive systems that allow information in a database to be remotely accessed and modified by a number of users. Information systems and resource management systems are examples of transaction processing systems.
- Language processing systems are used to translate texts from one language into another and to carry out the instructions specified in the input language. They include a translator and an abstract machine that executes the generated language.
- Data processing systems operate in batch mode and have an input-process-output structure
- Event processing systems include editors and real-time systems.

10.5 Self-Assessment Questions:

- Explain application architecture with an example.
- What is transaction processing system?
- Write a short note on Language processing system.
- How does Input-Process-Output model work?
- Explain the components of Data flow diagram.
- What are different levels of DFD?
- What is Event processing system?

10.6 List of References:

- Software Engineering, Ian Somerville, 8th edition, Pearson Education
- Software Engineering, Pankaj Jalote
- Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw-Hill

❖ ❖ ❖

UNIT 4

11

OBJECT-ORIENTED DESIGN

Unit Structure:

11.0 Objectives

The objective of this chapter is to introduce the concepts of OOPs, the process of Object-oriented design. When you have read the chapter, you will:

- be introduced to the OOPs concepts;
- Know the object oriented design process.

11.1 Introduction

Object-oriented (OO) approaches for software development have become extremely popular in recent years. Much of the new development is now being done using OO techniques and languages. There are many advantages that OO systems offer.

An OO model closely represents the problem domain, which makes it easier to produce and understand designs. As requirements change, the objects in a system are less immune to these changes, thereby permitting changes more easily. Inheritance and close association of objects in design to problem domain entities encourage more re-use, i.e., new applications can use existing modules more effectively, thereby reducing development cost and cycle time.

Object-oriented approaches are believed to be more natural and provide richer structures for thinking and abstraction. Common design patterns have also been uncovered that allow reusability at a higher level. The object-oriented design approach is fundamentally different from the function-oriented design approaches primarily due to the different abstraction that is used. It requires a different way of thinking and partitioning. It can be said that thinking in object-oriented terms is most important for producing truly object-oriented designs. In this section, we will first discuss some important concepts that form the basis of object-orientation.

11.2 OO Concepts

Here we will briefly discuss the main concepts behind object-orientation. Students familiar with an OO language will be familiar with these concepts.

11.2.1 Classes and Objects

Classes and objects are the basic building blocks of an OO design, just like functions (and procedures) are for a function-oriented design. Objects are entities that encapsulate some state and provide services to be used by a client, which could be another object, program, or a user. The basic property of an object is encapsulation: it encapsulates the data and information it contains and supports a well-defined abstraction. The set of services that can be requested from outside the object forms the interface of the object. An object may have operations defined only for internal use that cannot be used from outside. Such operations do not form part of the interface.

A major advantage of encapsulation is that access to the encapsulated data is limited to the operations defined on the data. Hence, it becomes much easier to ensure that the integrity of data is preserved, something very hard to do if any program from outside can directly manipulate the data structures of an object. Encapsulation and separation of the interface and its implementation, also allows the implementation to be changed without affecting the clients as long as the interface is preserved.

The encapsulated data for an object defines the state of the object. An important property of objects is that this state persists, in contrast to the data defined in a function or procedure, which is generally lost once the function stops being active (finishes its current execution). In an object, the state is preserved and it persists through the life of the object, i.e., unless the object is actively destroyed.

The state and services of an object together define its behavior. We can say that the behavior of an object is how an object reacts in terms of state changes when it is acted on, and how it acts on other objects by requesting services and operations. Generally, for an object, the defined operations together specify the behavior of the object.

Objects represent the basic runtime entities in an OO system; they occupy space in memory that keeps its state and is operated on by the defined operations on the object.

A class defines a possible set of objects. We have seen that objects have some attributes, whose values constitute much of the state of an object. What attributes an object has are defined by the class of the object. Similarly, the operations allowed on an object or the services it provides, are defined by the class of the object. But a class is merely a definition that does not create any objects and cannot hold any values. Each object, when it is created, gets a private copy of the instance variables, and when an operation defined on the class is performed on the object, it is performed on the state of the particular object. The relationship between a class and objects of that class is similar to the relationship between a type and elements of that type. A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class.

11.2.2 Relationships among Objects

An object, as a stand-alone entity, has very limited capabilities—it can only provide the services defined on it. Any complex system will be composed of

many objects of different classes, and these objects will interact with each other so that the overall system objectives are met. In object-oriented systems, an object interacts with another by sending a message to the object to perform some service it provides. On receiving the request message, the object invokes the requested service or the method and sends the result, if needed. This form of client-server interaction is a direct fall out of encapsulation and abstraction supported by objects. If an object invokes some services in other objects, we can say that the two objects are related in some way to each other. If an object uses some services of another object, there is an association between the two objects. This association is also called a link—a link exists from one object to another if the object uses some services of the other object. Links frequently show up as pointers when programming. A link captures the fact that a message is flowing from one object to another. However, when a link exists, though the message flows in the direction of the link, information can flow in both directions (e.g., the server may return some results).

With associations comes the issue of visibility, that is, which object is visible to whom. The basic issue here is that if there is a link from object A to object B, for A (client object) to be able to send a message to B (supplier object), B must be visible to A in the final program. There are different ways to provide this visibility. Some of the important possibilities are:

- The supplier object is global to the client.
- The supplier object is a parameter to some operation of the client that sends the message.
- The supplier object is a part of the client object.
- The supplier object is locally declared in some operation.

Links between objects capture the client/server type of relationship. Another type of relationship between objects is aggregation, which reflects the whole/part-of relationship. Though not necessary, aggregation generally implies containment. That is, if an object A is an aggregation of objects B and C, then objects B and C will generally be within object A (though there are situations where the conceptual relationship of aggregation may not get reflected as actual containment of objects). The main implication of this is that a contained object cannot survive without its containing object. With links, that is not the case.

11.2.3 Inheritance and Polymorphism

Inheritance is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes. When a class B inherits from another class A, B is referred to as the subclass or the derived class and A is referred to as the superclass or the base class. In general, a subclass B will have two parts: a derived part and an incremental part. The derived part is the part inherited from A and the incremental part is the new code and definitions that have been specifically added for B. This is shown in Figure 11.1 Objects of type B have the derived part as well as the incremental part. Hence, by defining only the incremental part and inheriting the derived part from an existing class, we can define objects that contain both. Inheritance is often called an "is a" relation, implying that an object of type B is also an instance of type A. That is, an instance of a subclass, though more than an instance of the superclass, is also an instance of the superclass. The inheritance relation between classes forms a hierarchy. As discussed earlier, it is important that the hierarchy represent a structure present in the application domain and is not created simply to reuse some parts of an existing class. And the hierarchy should be such that an object of a class is also an object of all its super classes in the problem domain. The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the superclass. In other words, a feature is placed in the higher level of abstractions.

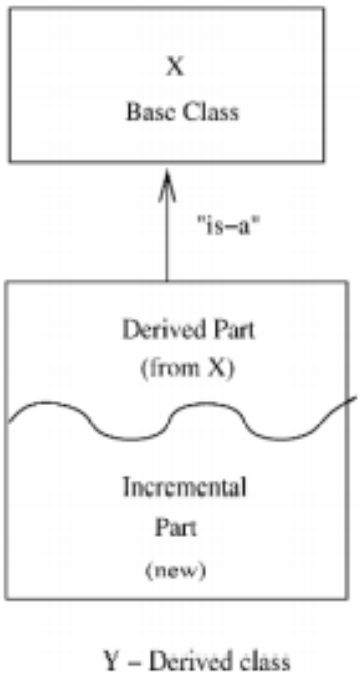


Figure 11.1 Inheritance

Once this is done, such features can be inherited from the parent class and used in the subclass directly. This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes.

In that case, the existing class can be tailored through inheritance to define the new class. Inheritance can be broadly classified as being of two types: strict inheritance and nonstrict inheritance. In strict inheritance a subclass takes all the features from the parent class and adds additional features to specialize it. That is, all data members and operations available in the base class are also available in the derived class. This form supports the "is-a" relation and is the easiest form of inheritance.

Nonstrict inheritance occurs when the subclass does not have all the features of the parent class or some features have been redefined. These forms do not satisfy Liskov's substitution principle. A class hierarchy need not be a simple tree structure. It may be a graph, which implies that a class may inherit from multiple classes. This type of in heritance, when a subclass inherits from many super classes, is called multiple inheritance.

Multiple inheritance complicates matters and its use is generally discouraged. We will assume that multiple inheritance is not to be used. Inheritance brings in polymorphism, a general concept widely used in type theory that deals with the ability of an object to be of different types. In OO programming, polymorphism comes in the form that a reference can refer to objects of different types at different times.

In object-oriented systems, with inheritance, polymorphism cannot be avoided—it must be supported. The reason is the "is a" relation supported by inheritance—an object x declared to be of class B is also an object of any class A that is the superclass of B. Hence, anywhere an instance of A is expected, x can be used.

With polymorphism, an entity has a static type and a dynamic type. The static type of an object is the type of which the object is declared in the program text, and it remains unchanged. The dynamic type of an entity, on the other hand, can change from time to time and is known only at reference time. Once an entity is declared at compile time, the set of types that this entity belongs to, can be determined from the inheritance hierarchy that has been defined. The dynamic type of the object will be one of this set, but the actual dynamic type will be defined at the time of reference of the object. This type of polymorphism requires dynamic binding of operations.

Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call. Let us illustrate with an example. Suppose x is a polymorphic reference whose static type is B but whose dynamic type could be either A or B. Suppose that an operation O() is defined in the class A, which is redefined in the class B. Now when the operation O() is invoked on x, it is not known statically what code will be executed. That is, the code to be executed for the statement x.O() is decided at runtime, depending on the dynamic type of x—if the dynamic type is A, the code for the operation O() in class A will be executed; if the dynamic type is B, the code for operation O() in class B will be executed. This dynamic binding can be used quite effectively during application development to reduce the size of the code. This feature polymorphism, which is essentially overloading of the feature (i.e., a feature can mean different things in different contexts and its exact meaning is determined only at runtime) causes no problem in strict inheritance because all features of a superclass are available in the subclasses. But in nonstrict inheritance, it can cause problems, because a child may lose a feature. Because the binding of the feature is determined at runtime, this can cause a runtime error as a situation may arise where the object is bound to the superclass in which the feature is not present.

11.3 An Object-Oriented Design Process

A methodology basically uses the concepts (of OO in this case) to provide guidelines for the design activity. Though methodologies are useful, they do not reduce the activity of design to a sequence of steps that can be followed mechanically. We briefly discuss one methodology here. Even though it is one of the earlier methodologies, its basic concepts are still applicable. We assume that during architecture design the system has been divided into high-level subsystems or components. The problem we address is how to produce an object-oriented design for a subsystem.

The OO design consists of specification of all the classes and objects that will exist in the system implementation. A complete OO design should be such that in the implementation phase, only further details about methods or attributes need to be added. A few low-level objects may be added later, but most of the classes and objects and their relationships are identified during design. An approach for creating an OO design consists of the following sequence of steps:

- Identify classes and relationships between them.
- Develop the dynamic model and use it to define operations on classes.
- Develop the functional model and use it to define operations on classes.

- Identify internal classes and operations.
- Optimize and package.

11.3.1 Identifying Classes and Relationships

Identifying the classes and their relationships requires identification of object types in the problem domain, the structures between classes (both inheritance and aggregation), attributes of the different classes, associations between the different classes, and the services each class needs to provide to support the system. Basically, in this step we are trying to define the initial class diagram of the design.

To identify analysis objects, start by looking at the problem and its description. In the descriptions consider the phrases that represent entities. Include an entity as an object if the system needs to remember something about it, the system needs some services from it to perform its own services, or it has multiple attributes. If the system does not need to keep information about some real-world entity or does not need any services from the entity, it need not be considered as an object for design. Carefully consider objects that have only one attribute; such objects can frequently be included as attributes in other objects. Though in the analysis we focus on identifying objects, in modeling, classes for these objects are represented.

Classes have attributes. Attributes add detail about the class and are the repositories of data for an object. For example, for an object of class Person, the attributes could be the name, sex, and address. The data stored in forms of values of attributes are hidden from outside the objects and are accessed and manipulated only by the service functions for that object. Which attributes should be used to define the class of an object depends on the problem and what needs to be done. For example, while modeling a hospital system, for the class Person attributes of height, weight, and date of birth may be needed, although these may not be needed for a database for a county that keeps track of populations in various neighborhoods.

To identify attributes, consider each class and see which attributes are needed by the problem domain. This is frequently a simple task. Then position each attribute properly using the structures; if the attribute is a common attribute, it should be placed in the superclass, while if it is specific to a specialized object, it should be placed with the subclass. While identifying attributes, new classes may also get defined or old classes may disappear (e.g., if you find that a class really is an attribute of another).

For a class diagram, we also need to identify the structures and associations between classes. To identify the classification structure, consider the classes that have been identified as a generalization and see if there are other classes that can be considered as specializations of this. The specializations should be meaningful for the problem domain. For example, if the problem domain does not care about the material used to make some objects, there is no point in specializing the classes based on the material they are made of. Similarly, consider classes as specializations and see if there are other classes that have similar attributes. If so, see if a generalized class can be identified of which these are specializations. Once again, the structure obtained must naturally reflect the hierarchy in the problem domain; it should not be "extracted" simply because some classes have some attributes with the same names.

To identify assembly structure, a similar approach is taken. Consider each object of a class as an assembly and identify its parts. See if the system needs to keep track of the parts. If it does, then the parts must be reflected as objects; if not, then the parts should not be modeled as separate objects. Then, consider an object of a class as a part and see to which class's object it can be considered as belonging. Once again, this separation is maintained only if the system needs it. As before, the structures identified should naturally reflect the hierarchy in the problem domain and should not be "forced."

For associations we need to identify the relationship between instances of various classes. For example, an instance of the class Company may be related to an instance of the class Person by an "employs" relationship. This is similar to what is done in ER modeling. And like in ER modeling, an instance connection may be of 1:1 type representing that one instance of this type is related to exactly one instance of another class. Or it could be 1:M, indicating that one instance of this class may be related to many instances of the other class. There are M:M connections, and there are sometimes multiway connections, but these are not very common. The associations between objects are derived from the problem domain directly once the objects have been identified. An association may have attributes of its own; these are typically attributes that do not naturally belong to either object. Although in many situations they can be "forced" to belong to one of the two objects without losing any information, it should not be done unless the attribute naturally belongs to the object.

11.3.2 Dynamic Modeling

The class diagram obtained gives the initial module-level design. This design will be further shaped by the events in the system, as the design has to ensure that the

expected behavior for the events can be supported. Modeling the dynamic behavior of the system will help in further refining the design. The dynamic model of a system aims to specify how the state of various objects changes when events occur. An event is something that happens at some time instance. For an object, an event is essentially a request for an operation. An event typically is an occurrence of something and has no time duration associated with it. Each event has an initiator and a responder. Events can be internal to the system, in which case the event initiator and the event responder are both within the system. An event can be an external event, in which case the event initiator is outside the system (e.g., the user or a sensor).

A scenario is a sequence of events that occur in a particular execution of the system. From the scenarios, the different events being performed on different objects can be identified, which are then used to identify services on objects. The different scenarios together can completely characterize the behavior of the system. If the design is such that it can support all the scenarios, we can be sure that the desired dynamic behavior of the system can be supported by the design. This is the basic reason for performing dynamic modeling. With use cases, dynamic modeling involves preparing interaction diagrams for the important scenarios. It is best to start by modeling scenarios being triggered by external events. The scenarios should not necessarily cover all possibilities, but the major ones should be considered. First the main success scenarios should be modeled, then scenarios for “exceptional” cases should be modeled. For example, in a restaurant, the main success scenario for placing an order could be the following sequence of actions: customer reads the menu; customer places the order; order is sent to the kitchen for preparation; ordered items are served; customer requests a bill for the order; bill is prepared for this order; customer is given the bill; customer pays the bill. An “exception” scenario could be if the ordered item was not available or if the customer cancels his order.

From each scenario, events have to be identified. Events are interactions with the outside world and object-to-object interactions. All the events that have the same effect on the flow of control in the system are grouped as a single event type. Each event type is then allocated to the object classes that initiate it and that service the event. With this done, a scenario can be represented as a sequence (or collaboration) diagram showing the events that will take place on the different objects in the execution corresponding to the scenario. A possible sequence diagram of the main success scenario of the restaurant is given in Figure 11.2.

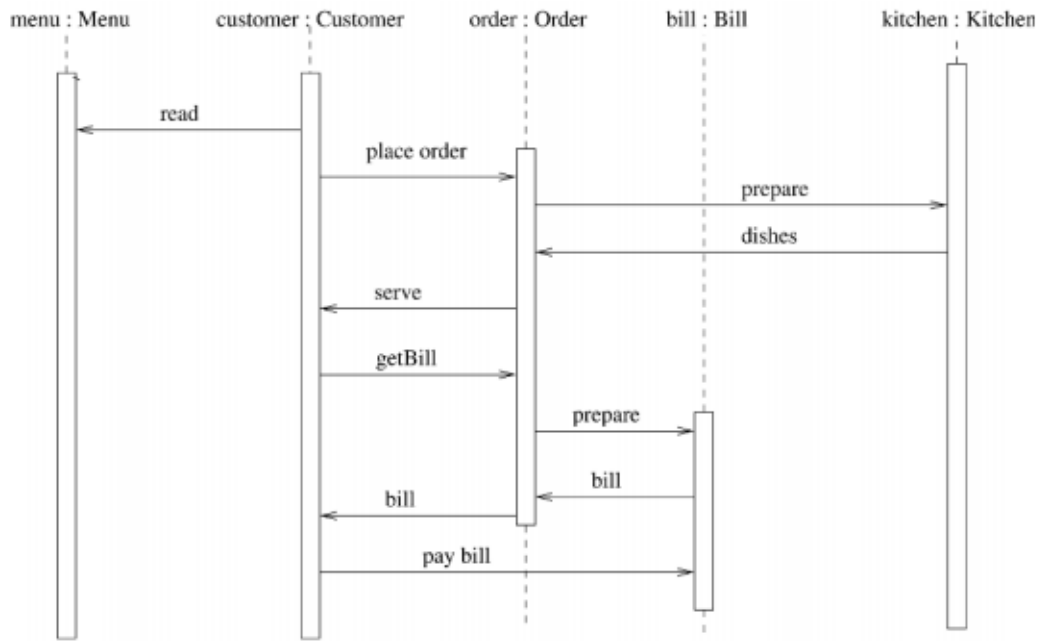


Figure 11.2 A sequence diagram for the restaurant.

Once the main scenarios are modeled, various events on objects that are needed to support executions corresponding to the various scenarios are known. This information is then used to expand our view of the classes in the design. Generally speaking, for each event in the sequence diagrams, there will be an operation on the object on which the event is invoked. So, by using the scenarios and sequence diagrams we can further refine our view of the objects and add operations that are needed to support some scenarios but may not have been identified during initial modeling. For example, from the event trace diagram in Figure 6.18, we can see that “place order” and “getBill” will be two operations required on the object of type Order if this interaction is to be supported. The effect of these different events on a class itself can be modeled using the state diagrams. We believe that the state transition diagram is of limited use during system design but may be more useful during detailed design.

11.3.3 Functional Modeling

A functional model of a system specifies how the output values are computed in the system from the input values, without considering the control aspects of the computation. This represents the functional view of the system—the mapping from inputs to outputs and the various steps involved in the mapping. Generally, when the transformation from the inputs to outputs is complex, consisting of many steps, the functional modeling is likely to be useful. In systems where the transformation of inputs to outputs is not complex, functional model is likely to

be straightforward. As we have seen, the functional model of a system can be represented by a data flow diagram (DFD). We have used DFDs in problem modeling, and the structured design methodology, discussed earlier. Just as with dynamic modeling, the basic purpose of doing functional modeling is to use the model to make sure that the object model can perform the transformations required from the system. As processes represent operations and in an object-oriented system, most of the processing is done by operations on classes, all processes should show up as operations on classes. Some operations might appear as single operations on an object; others might appear as multiple operations on different classes, depending on the level of abstraction of the DFD. If the DFD is sufficiently detailed, most processes will occur as operations on classes. The DFD also specifies the abstract signature of the operations by identifying the inputs and outputs.

11.3.4 Defining Internal Classes and Operations

The classes identified so far are the ones that come from the problem domain. The methods identified on the objects are the ones needed to satisfy all the interactions with the environment and the user and to support the desired functionality. However, the final design is a blueprint for implementation. Hence, implementation issues have to be considered. While considering implementation issues, algorithm and optimization issues arise. These issues are handled in this step.

First, each class is critically evaluated to see if it is needed in its present form in the final implementation. Some of the classes might be discarded if the designer feels they are not needed during implementation. Then the implementation of operations on the classes is considered. For this, rough algorithms for implementation might be considered. While doing this, a complex operation may get defined in terms of lower-level operations on simpler classes.

In other words, effective implementation of operations may require heavy interaction with some data structures and the data structure to be considered an object in its own right. These classes that are identified while considering implementation concerns are largely support classes that may be needed to store intermediate results or to model some aspects of the object whose operation is to be implemented. Once the implementation of each class and each operation on the class has been considered and it has been satisfied that they can be implemented, the system design is complete. The detailed design might also uncover some very low-level objects, but most such objects should be identified during system design.

11.3.5 Optimize and Package

During design, some inefficiencies may have crept in. In this final step, the issue of efficiency is considered, keeping in mind that the final structures should not deviate too much from the logical structure produced. Various optimizations are possible and a designer can exercise his judgment keeping in mind the modularity aspects also.

11.4 Summary:

- The clear idea of the OOPs concepts.
- The structured design methodology gives guidelines on how to create a design (represented as a structure chart) such that the modules have minimum dependence on each other (low coupling) and a high level of cohesion. For this, the methodology partitions the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. These cleanly partitions the system into parts each independently dealing with different concerns.
- The OO design methodology focuses on identifying classes and relationships between them, and validating the class definitions using dynamic and functional modelling.

11.5 Self-Assessment Questions:

- What are the OOPs concepts?
- Write the steps of object oriented design process.
- Explain the following with example:
    a. Inheritance
    b. Polymorphism
    c. Classes and objects
- What is dynamic modeling?
- Write a short note on functional modeling.

11.6 List of References:

- Software Engineering, Ian Somerville, 8th edition, Pearson Education
- Software Engineering, Pankaj Jalote
- Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw-Hill

❖ ❖ ❖

UNIT 4

12

USER INTERFACE DESIGN – RAPID SOFTWARE DEVELOPMENT

Unit Structure:

12.0 Objectives

The objective of this chapter is to introduce some aspects of user interface design that are important for software engineers. When you have read this chapter, you will:

- understand a number of user interface design principles;
- be introduced to several interaction styles and understand when these are most appropriate;
- understand when to use graphical and textual presentation of information;
- come to know what is involved in the principal activities in the use interface design process;
- Understand usability attributes and have been introduced to different approaches to interface evaluation.
- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan driven development;
- know the key practices in extreme programming and how these relate to the general principles of agile methods;
- understand the importance of RAD
- know the concept of Software prototyping

12.1 Introduction

Computer system design encompasses a spectrum of activities from hardware design to user interface design. While specialists are often employed for hardware design and for the graphic design of web pages, only large organizations normally employ specialist interface designers for their application software. Therefore, software engineers must often take responsibility for user interface design as well as for the design of the software to implement that interface. Even when software designers and programmers are competent users of interface implementation technologies, such as Java's Swing classes (Elliott et al., 2(02) or XHTML (Musdano and Kennedy, 2002), the user interfaces they develop are often unattractive and inappropriate for their target users.

The focus here is, therefore, on the design products for user interfaces rather than

the software that implements these facilities. Because of space limitations, it considered only graphical user interfaces. it is not discussed about interfaces that require special (perhaps very simple) displays such as cell phones, DVD players, televisions, copiers and fax machines.

Careful user interface design is an essential part of the overall software design process. If a software system is to achieve its full potential, it is essential that its user interface should be designed to match the skills, experience and expectations of its anticipated users. Good user interface design is critical for system dependability. Many so-called user errors are caused by the fact that user interfaces do not consider the capabilities of real users and their working environment. A poorly designed user interface means that users will probably be unable to access some of the system features, will make mistakes and will feel that the system hinders rather than helps them in achieving whatever they are using the system for.

12.2 Need of UI Design

When making user interface design decisions, you should take into account the physical and mental capabilities of the people who use software. Human issues in detail discussed here but important factors that you should consider are:

1. People have a limited short-term memory-we can instantaneously remember about seven items of information (Miller, 1957). Therefore, if you present users with too much information at the same time, they may not be able to take it all ill.

2. We all make mistakes, especially when we have to handle too much information or are under stress. When systems go wrong and issue warning messages and alarms, this often puts more stress on users, thus increasing the chances that they will make operational errors.

3. We have a diverse range of physical capabilities. Some people see and hear better than others, some people are color-blind, and some are better than others at physical manipulation. You should not design for your own capabilities and assume that all other users will be able to cope.

4. We have different interaction preferences. Some people like to work with pictures, others with text. Direct manipulation is natural for some people, but others prefer a style of interaction that is based on issuing commands to the system.

Principle	Description
User familiarity	The interface should use terms and concepts drawn from the experience of the people who will make most use of the system.
Consistency	The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.
Minimal surprise	Users should never be surprised by the behaviour of a system.
Recoverability	The interface should include mechanisms to allow users to ecover from errors.
User guidance	The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.
User diversity	The interface should provide appropriate interaction facilities for different types of system users.

Figure 12.1

These human factors are the basis for the design principles shown in Figure 12.1. These general principles are applicable to all user interface designs and should normally be instantiated as more detailed design guidelines for specific organizations or types of system. User interface design principles are covered in more detail by Dix, et al. (Dix, et al., 2004). Shneiderman (Shneiderman, 1998) gives a longer list of more specific user interface design guidelines.

The principle of user familiarity suggests that users should not be forced to adapt to an interface because it is convenient to implement. The interface should use terms that are familiar to the user, and the objects manipulated by the system should be directly related to the user s working environment. For example, if a system is designed for use by air traffic controllers, the objects manipulated should be aircraft, flight paths, beacons, and so on. Associated operations might be to increase or reduce aircraft speed, adjust heading and change height. The underlying implementation of the interface in terms of files and data structures should be hidden from the end user.

The principle of user interface consistency means that system commands and menus should have the same format, parameters should be passed to all commands in the same way, and command punctuation should be similar. Consistent interfaces reduce user learning time. Knowledge learned in one command or application is therefore applicable in other parts of the system or in related applications. Interface consistency across applications is also important. As far as possible, commands with similar meanings in different applications should be expressed in the same way.

Errors are often caused when the same keyboard command, such as 'Control-b' means different things in different systems. For example, in the word processor that is normally used, 'Control-b' means embolden text, but in the graphics program that is used to draw diagrams, 'Control-b' means move the selected object behind another object. The mistakes are made when using them together and sometimes try to embolden text in a diagram using the key combination. Then to get confused when the text disappears behind the enclosing object. You can normally avoid this kind of error if you follow the command key shortcuts defined by the operating system that you use.

This level of consistency is low-level. Interface designers should always try to achieve this in a user interface. Consistency at a higher level is also sometimes desirable. For Example, it may be appropriate to support the same operations (print, copy, etc.) on all types of system entities. However, Grodin (Grodin, 1989) points out that complete consistency is neither possible nor desirable. It may be sensible to implement deletion from a desktop by dragging entities into a trash can. It would be awkward to delete text in a word processor in this way. Unfortunately, the principles of user familiarity and user consistency are sometimes conflicting. Ideally, applications with common features should always use the same commands to access these features. However, this can conflict with user practice when systems are designed to support a particular type of user, such as graphic designers. These users may have evolved their own styles of interactions, terminology and operating conventions. These may clash with the interaction 'standards' that are appropriate to more general applications such as word processors.

The principle of minimal surprise is appropriate because people get very irritated when a system behaves in an unexpected way. As a system is used, users build a mental model of how the system works. If an action in one context causes a particular type of change, it is reasonable to expect that the same action in a different context it will cause a comparable change. If something completely different happens, the user is both surprised and confused.

Interface designers should therefore try to ensure that comparable actions have comparable effects. Surprises in user interfaces are often the result of the fact that many interfaces are moded. This means that there are several modes of working (e.g., viewing mode and editing mode), and the effect of a command is different depending on the mode. It is very important that, when designing an interface, you include a visual indicator showing the user the current mode.

The principle of recoverability is important because users inevitably make mistakes when using a system. The interface design can minimize these mistakes

(e.g., using menus means avoids typing mistakes), but mistakes can never be completely eliminated. Consequently, you should include interface facilities that allow users to recover Lom their mistakes. These can be of three kinds:

1. Confirmation of destructive actions

   If a user specifies an action that is potentially destructive, the system should ask the user to confirm that this is really what is wanted before destroying any information.

2. The provision of an undo facility

   Undo restores the system to a state before the action occurred. Multiple levels of undo are useful because users don't always recognize immediately that a mistake has been made.

3. Checkpointing

   Checkpointing involves saving the state of a system at periodic intervals and allowing the system to restart from the last checkpoint. Then, when mistakes occur, users can go back to a previous state and start again. Many systems now include checkpointing to cope with system failures but, paradoxically, they don't allow system users to use them to recover from their own mistakes.

A related principle is the principle of user assistance. Interfaces should have built in user assistance or help facilities. These should be integrated with the system and should provide different levels of help and advice. Levels should range from basic information on getting started to a full description of system facilities. Help systems should be structured so that users are not overwhelmed with information when they ask for help.

The principle of user diversity recognizes that, for many interactive systems, there may be different types of users. Some will be casual users who interact occasionally with the system while others may be power users who use the system for several hours each day.

Casual users need interfaces that provide guidance, but power users require shortcuts so that they can interact as quickly as possible. Furthermore, users may suffer from disabilities of various types and, if possible, the interface should be

adaptable to cope with these. Therefore, you might include facilities to display enlarged text, to replace sound with text, to produce very large buttons and so on. This reflects the notion of Universal Design (UD) (Preiser and Ostoff, 2001), a design philosophy whose goal is to avoid excluding users because of thoughtless design choices.

The principle of recognizing user diversity can conflict with the other interface design principles, since some users may prefer very rapid interaction over, for example, user interface consistency. Similarly, the level of user guidance required can be radically different for different users, and it may be impossible to develop support that is suitable for all types of users. You therefore have to make compromises to reconcile the needs of these users.

12.3 Design issues

A coherent user interface must integrate user interaction and information presentation. This can be difficult because the designer has to find a compromise between the most appropriate styles of interaction and presentation for the application, the background and experience of the system users, and the equipment that is available.

12.3.1 User interaction

User interaction means issuing commands and associated data to the computer system. On early computers, the only way to do this was through a command-line interface, and a special-purpose language was used to communicate with the machine. However, this was geared to expert users and a number of approaches have now evolved that are easier to use. Shneiderman (Shneiderman, 1998) has classified these forms of interaction into five primary styles:

- Direct manipulation

  The user interacts directly with objects on the screen. Direct manipulation usually involves a pointing device (a mouse, a stylus, a trackball or, on touch screens, a finger) that indicates the object to be manipulated and the action, which specifies what should be done with that object. For example, to delete at file, you may click on an icon representing that file and drag it to a trash can icon.

- Menu selection

  The user selects a command from a list of possibilities (a menu). The user may also select another screen object by direct manipulation, and the command operates on that object. In this approach, to delete a file, you would select the file icon then select the delete command.

- Form fill-in

  The user fills in the fields of a form. Some fields may have associated menus, and the form may have action 'buttons' that, when pressed, cause some action to be initiated. You would not normally use this approach to implement the interface to operations such as file deletion. Doing so would involve filling in the name of the file on the form then 'pressing' a delete button.

- Command language

  The user issues a special command and associated parameters to instinct the system what to do. To delete a file, you would type a delete command with the filename as a parameter.

- Natural language

  The user issues a command in natural language. This is usually a front end to a command language; the natural language is parsed and translated to system commands. To delete a file, you might type 'delete the file named xxx.

  Each of these styles of interaction has advantages and disadvantages and is best suited to a particular type of application and user (Shneiderman, 1998). Figure 12.2 shows the main advantages and disadvantages of these styles and suggests types of applications where they might be used. Of course, these interaction styles may be mixed, with several styles used in the same application.

Interaction style	Main advantages	Main disadvantages	Application examples
Direct manipulation	Fast and intuitive interaction Easy to learn	May be hard to implement Only suitable where there is a visual metaphor for tasks and objects	Video games CAD systems
Menu selection	Avoids user error Little typing required	Slow for experienced users Can become complex if many menu options	Most general-purpose systems
Form fill-in Easy to learn Checkable	Simple data entry	Takes up a lot of screen space Causes problems where user options do not match the form fields	Stock control Personal loan processing
Command language	Powerful and flexible	Hard to learn Poor error management	Operating systems Command and control systems
Natural language	Accessible to casual users Easily extended	Requires more typing Natural language understanding systems are unreliable	Information retrieval systems

Figure 12.2 Advantages and disadvantages of interaction styles

For example, Microsoft Windows supports direct manipulation of the iconic representation of files and directories, menu-based command selection, and for commands such as configuration commands, the user must fill in a special-purpose form that is presented to them. In principle, it should be possible to separate the interaction style from the underlying entities that are manipulated through the user interface. This was the basis of the Seeheim model (Pfaff and ten Hagen, 1985) of user interface management. In this model, the presentation of information, the dialogue management and the application are separate.

In reality, this model is more of an ideal than practical, but it is certainly possible to have separate interfaces for different classes of users (casual users and experienced users, say) that interact with the same underlying system.

This is illustrated in Figure 12.3, which shows a command language interface and a graphical interface to an underlying operating system such as Linux. Web-based user interfaces are based on the support provided by HTML or XHTML (the page description languages used for web pages) along with languages such as Java, which can associate programs with components on a page. Because these web-based interlaces are usually designed for casual users, they mostly use forms-based interfaces.

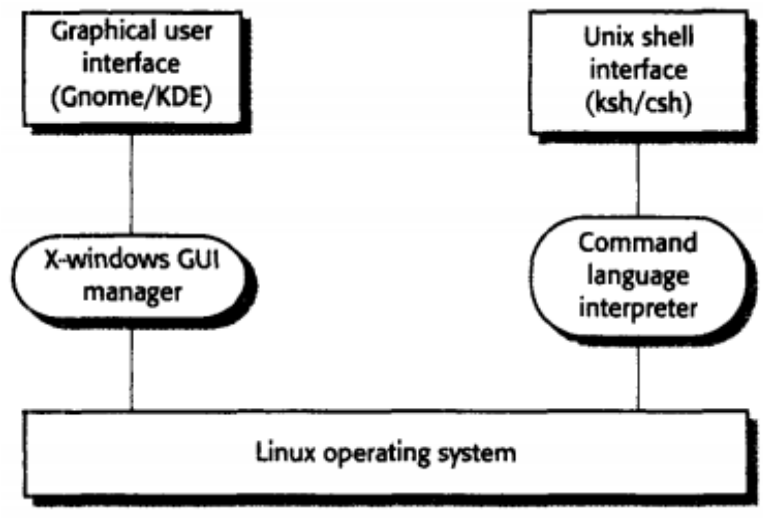


Figure 12.3

It is possible to construct direct manipulation interlaces on the web, but this is a complex programming task. Furthermore, because of the range of experience of web users and the fact that they come from many different cultures, it is difficult to establish a user interface metaphor for direct interaction that is universally acceptable.

To illustrate the design of web-based user interaction, it is discussed the approach used in the LIBSYS system where users can access documents from other libraries. There are two, fundamental operations that need to be supported:

1. Document search where users use the search facilities to find the documents that they need.

2. Document request where users request that the document be delivered to their local machine or server for printing The LIBSYS user interface is implemented using a web browser, so, given that users must supply information to the system such as the document identifier, their name and their authorization details, it makes sense to use a forms-based interlace. Figure 12.4 shows a possible interlace design for the search component of the system. In form-based interfaces, the user supplies all of the information required then initiates, the action by pressing a button. Forms fields can be menus, free-text input fields or radio buttons. In the LIBSYS example, a user chooses the collection to search from a menu of collections that can be accessed ('All' is the default, meaning search all collections) and types the search phrase into a free-text input field. The user chooses the field of the library record from a menu ('Title' is the default) and selects a radio button to indicate whether the search terms should be adjacent in the record.

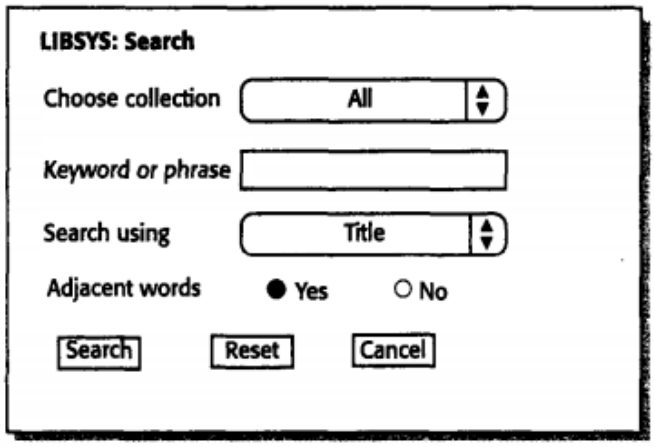


Figure 12.4 A forms-based interface to the L1BSYS system

12.3.2 Information presentation

All interactive systems have to provide some way of presenting information to users. The information presentation may simply be a direct representation of the input information (e.g., text in a word processor) or it may present the information graphically. A good design guideline is to keep the software required for information presentation separate from the information itself. Separating the presentation system from the data allows us to change the representation on the user s screen without having to change the underlying computational system. This is illustrated in Figure 12.5. The MVC approach (Figure 12.6), first made widely available in Smalltalk (Goldberg and Robson, 1983), is an effective way to support multiple presentations of data. Users can interact with each presentation in a style that is appropriate to the presentation. The data to be displayed is encapsulated in a model object. Each model object may have a number of separate view objects associated with it where each view is a different display representation of the model. Each view has an associated controller object that handles user input and device interaction. Therefore, a model that represents numeric data may have a view that represents the data as a histogram and a view that presents the data as a table. The model may be edited by changing the values in the table or by lengthening or shortening the bars in the histogram.

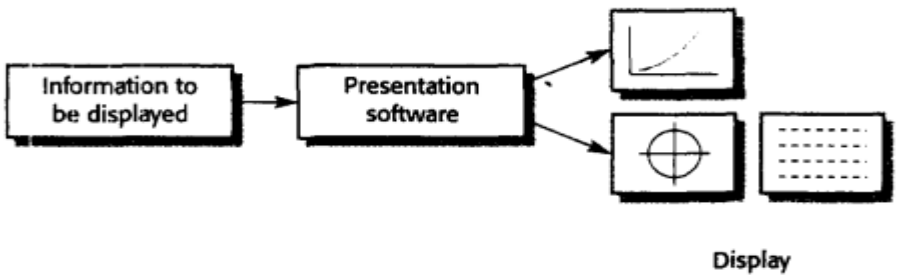


Figure 12.5

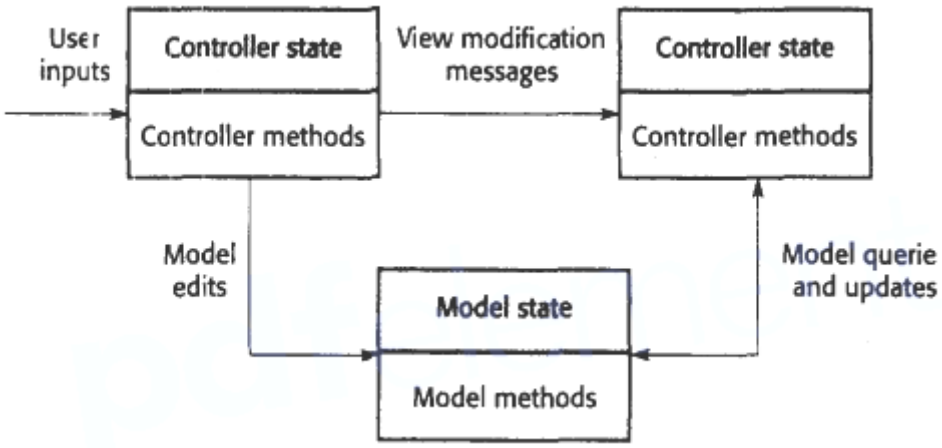


Figure 12.6

You should not assume that using graphics makes your display more interesting. Graphics take up valuable screen space (a major issue with portable devices) and can take a long time to download if the user is working over a slow, dial-up connection. Information that does not change during a session may be presented either graphically or as text depending on the application. Textual presentation takes up less screen space but cannot be read at a glance. You should distinguish information that does not change from dynamic information by using a different presentation style. For example, you could present all static information in a particular font or color, or you could associate a 'static information' icon with it. You should use text to present information when precise information is required and the information changes relatively slowly. If the data changes quickly or if the relationships between data rather than the precise data values are significant, then you should present the information graphically.

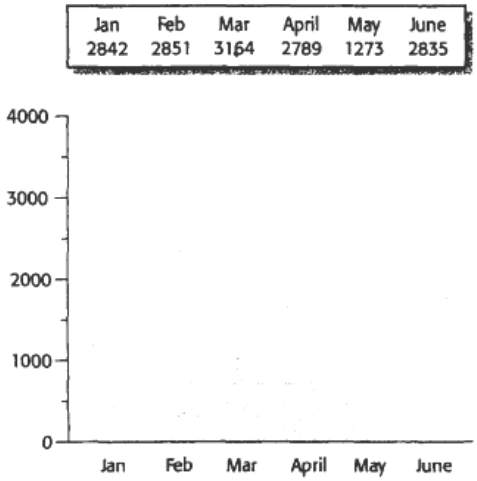


Figure 12.7 Alternative information presentations

For example, consider a system that records and summarizes the sales figures for a company on a monthly basis. Figure 12.7 illustrates how the same information can be presented as text or in a graphical form. Managers studying sales figures are usually more interested in trends or anomalous figures rather than precise values. Graphical presentation of this information, as a histogram, makes the anomalous figures in March and May stand out from the others. Figure 12.7 also illustrates how textual presentation takes less space than a graphical representation of the same information.

In control rooms or instrument panels such as those on a car dashboard, the information that is to be presented represents the state of some other system (e.g., the altitude of an aircraft) and is changing all the time. A constantly changing digital display can be confusing and irritating as readers can't read and assimilate the information before it changes. Such dynamically varying numeric information is therefore best presented graphically using an analogue representation. The graphical display can be supplemented if necessary with a precise digital display. Different ways of presenting dynamic numeric information are shown in Figure 12.8. Continuous analogue displays give the viewer some sense of relative value. In Figure 12.9, the values of temperature and pressure are approximately the same. However, the graphical display shows that temperature is close to its maximum value whereas pressure has not reached 25% of its maximum. With only a digital value, the viewer must know the maximum values and mentally compute the relative state of the reading. The extra thinking time required can lead to human errors in stressful situations when problems occur and operator displays may be showing abnormal readings.

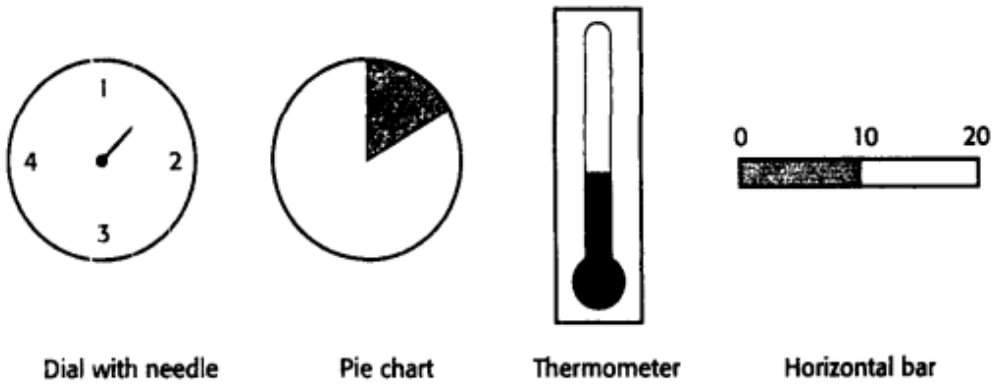


Figure 12.8 Methods of presenting dynamically varying numeric information

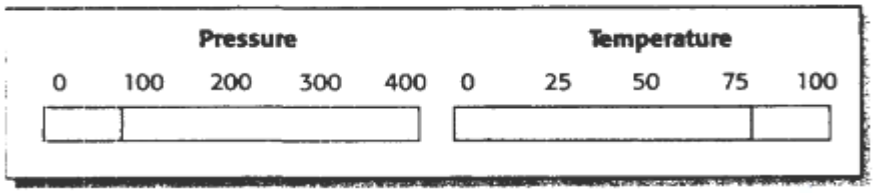


Figure 12.9 Graphical information display showing relative values

When large amounts of information have to be presented, abstract visualizations that link related data items may be used. This can expose relationships that are not obvious from the raw data. You should be aware of the possibilities of visualization, especially when the system user interface must represent physical entities. Examples of data visualizations are:

1. Weather information, gathered from a number of sources, is shown as a weather map with isobars, weather fronts, and so on.

2. The state of a telephone network is displayed graphically as a linked set of nodes in a network management centre.

3. The state of a chemical plant is visualized by showing pressures and temperatures in a linked set of tanks and pipes.

4. A model of a molecule is displayed and manipulated in three dimensions using a virtual reality system.

5. A set of web pages is displayed as a hyperbolic tree (Lamping et al., 1995).

Shneiderman (Shneiderman, 1998) offers a good overview of approaches to visualization as well as identifies classes of visualization that may be used. These include visualizing data. Using two- and three-dimensional presentations and as trees or networks. Most of these are concerned with the display of large amounts of information managed on a computer. However, the most common use of visualization in user interfaces is to represent some physical structure such as the molecular structure of a new drug, the links in a telecommunications network and so on. Three- dimensional presentations that may use special virtual reality equipment are particularly effective in product visualizations. Direct manipulation of these visualizations is a very effective way to interact with the data. In addition to the style of information presentation, you should think carefully about how color is used in the interface. Color can improve user interfaces by helping users understand and manage complexity. However, it is easy to misuse color and to create user interfaces that are visually unattractive and error-prone. Shneiderman

gives 14 key guidelines for the effective use of color in user interfaces. The most important of these are:

1. Limit the number of colors employed and be conservative how these are used

   You should not use more than four or five separate colors in a window and no more than seven in a system interface. If you use too many, or if they are too bright, the display may be confusing. Some users may find masses of color disturbing and visually tiring. User confusion is also possible if colors are used inconsistently.

2. Use color change to show a change in system status

   If a display changes color, this should mean that a significant event has occurred. Thus, in a fuel gauge, you could use a change of color to indicate that fuel is running low. Color highlighting is particularly important in complex displays where hundreds of distinct entities may be displayed.

3. Use color coding to support the task users are trying to perform

   If they have to identify anomalous instances, highlight these instances; if similarities are also to be discovered, highlight these using a different color.

4. Use color coding in a thoughtful and consistent way

   If one part of a system displays error messages in red (say), all other parts should do likewise. Red should not be used for anything else. If it is, the user may interpret the red display as an error message.

5. Be careful about color pairings

Because of the physiology of the eye, people cannot focus on red and blue simultaneously. Eyestrain is a likely consequence of a red on blue display. Other color combinations may also be visually disturbing or difficult to read.

In general, you should use color for highlighting, but you should not associate meanings with particular colors. About 10% of men are color-blind and may misinterpret the meaning. Human color perceptions are different, and there are different conventions in different professions about the meaning of particular colors. Users with different backgrounds may unconsciously interpret the same color in different ways. For example, to a driver, red usually means danger. However, to a chemist, red means hot. As well as presenting application

information, systems also communicate with users through messages that give information about errors and the system state. A user's first experience of a software system may be when the system presents an error message. Inexperienced users may start work, make an initial error and immediately have to understand the resulting error message. This can be difficult enough for skilled software engineers. It is often impossible for inexperienced or casual system users. Factors that you should take into account when designing system messages are shown in Figure 12.10.

Factor	Description
Context	Wherever possible, the messages generated by the system should reflect the current user context. As far as is possible, the system should be aware of what the user is doing and should generate messages that are relevant to their current activity
Experience	As users become familiar with a system they become irritated by long, 'meaningful' messages. However, beginners find it difficult to understand short, terse statements of a problem. You should provide both types of message and allow the user to control message conciseness.
Skill level	Messages should be tailored to the users' skills as well as their experience. Messages for the different classes of users may be expressed in different ways depending on the terminology that is familiar to the reader.
Style	Messages should be positive rather than negative. They should use the active rather than the passive mode of address. They should never be insulting or try to be funny.
Culture	Wherever possible, the designer of messages should be familiar with the culture of the country where the system is sold. There are distinct cultural differences between Europe, Asia and America. A suitable message for one culture might be unacceptable in another.

Figure 12.10 Design factors in message wording

You should anticipate the background and experience of users when designing error messages.. For example, say a system user is a nurse in an intensive-care ward in a hospital. Patient monitoring is carried out by a computer system. To view a patient's current state (heart rate, temperature, etc.), the nurse selects 'display' from a menu and inputs the patient's name in the box, as shown in Figure 12.11. In this case, let's assume that the nurse has misspelled the patient's name and has typed 'MacDonald' instead of 'McDonald'. The system generates an error message. Error messages should always be polite, concise, consistent and constructive. They must not be abusive and should not have associated beeps or other noises that might embarrass the user. Wherever possible, the message

should suggest how the error might be corrected. The error message should be linked to a context-sensitive online help system. Figure: 12.12 shows examples of good and bad error messages. The left-hand message is badly designed. It is negative (it accuses the user of making an error), it is not tailored to the user's skill and experience level, and it does not take context information into account.

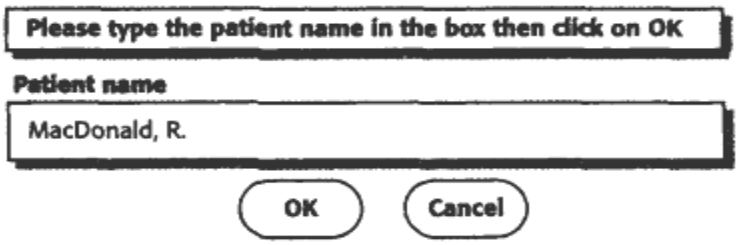


Figure 12.11 An input text box used by a nurse

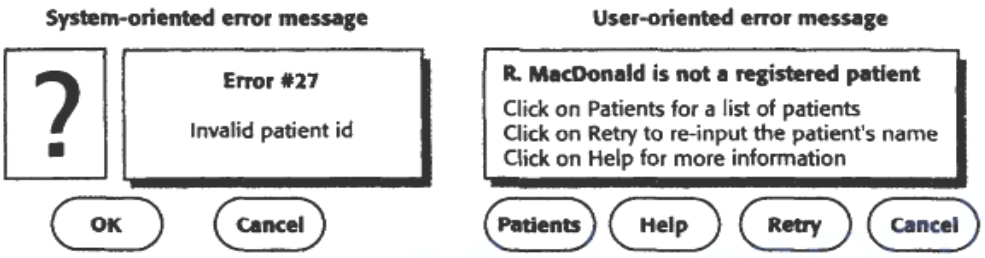


Figure 12.12 System and user-oriented error messages

It does not suggest how the situation might be rectified. It uses system-specific terms (patient id) rather than user-oriented language. Th right-hand message is better. It is positive, implying that the problem is a system rather than a user problem. It identifies the problem in the nurse's terms and offers an easy way to correct the mistake by pressing a single button. The help system is available if required.

12.4 The UI design process

User interface (UI) design is an iterative process where users interact with designers and interface prototypes to decide on the features, organisation and the look and feel of the system user interface. Sometimes, the interface is separately prototyped in parallel with other software engineering activities. More commonly, especially where iterative development is used, the user interface design proceeds incrementally as the software is developed. In both cases, however, before you start programming, you should have developed and, ideally, tested some paper-based designs. The overall UI design process is illustrated in Figure 12.13. There are three core activities in this process:

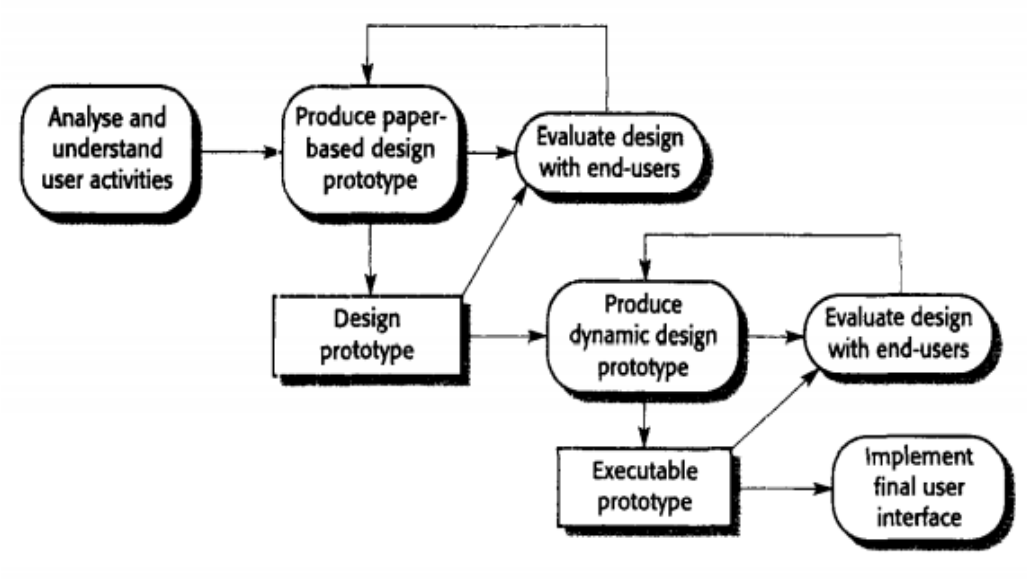


Figure 12.13 The UI design process

1. User analysis

In the user analysis process, you develop an understanding of the tasks that users do, their working environment, the other systems that they use, how they interact with other people in their work and so on. For products with a diverse range of users, you have to try to develop this understanding through focus groups, trials with potential users and similar exercises.

2. System-prototyping

User interface design and development is an iterative process. Although users may talk about the facilities they need from an interface, it is very difficult for them to be specific until they see something tangible. Therefore, you have to develop prototype systems and expose them to users, who call. then guide the evolution of the interface.

3. Interface evaluation

Although you will obviously have discussions with users during the prototyping process, you should also have a more formalized evaluation activity where you collect information about the users actual experience with the interface.

Radhika is a religious studies student writing an essay on Indian architecture and how it has been influenced by religious practices. To help her understand this, she would like to access pictures of details on notable buildings but cannot find anything in her local library. She approaches the subject librarian to discuss her needs and he suggests search terms that she might use. He also suggests libraries in Mumbai and London that might have this material, so he and Radhika log on to the library catalogues & search using these terms. They find some source material and place a request for photocopies of the pictures with architectural details, to be posted directly to Radhika

Figure 12.14 A library interaction scenario

12.5 User analysis

A critical UI design activity is the analyses of the user activities that are to be supported by the computer system. If you don't understand what users want to do with a system, then you have no realistic prospect of designing an effective user interface. To develop this understanding, you may use techniques such as task analysis, ethnographic studies, user interviews and observations or, commonly, a mixture of all of these.

A challenge for engineers involved in user analysis is to find a way to describe user analyses so that they communicate the essence of the tasks to other designers and to the users themselves. Notations such as UML sequence charts may be able to describe user interactions and are ideal {or communicating with software engineers. However, other users may think of these charts as too technical and will not try to understand them. Because it is very important to engage users in the design process, you therefore usually have to develop natural language scenarios to describe user activities. Figure 12.14 is an example of a natural language scenario that might have been developed during the specification and design process for the LIBSYS system. It describes a situation where LIBSYS does not exist and where a student needs to retrieve information from another library. From this scenario, the designer can see a number of requirements:

1. Users might not be aware of appropriate search terms. They may need to access ways of helping them choose search terms.

2. Users have to be able to select collections to search.

3. Users need to be able to carry out searches and request copies of relevant material.

You should not expect user analysis to generate very specific user interface requirements. Normally, the analysis helps you understand the needs and concerns of the system users.

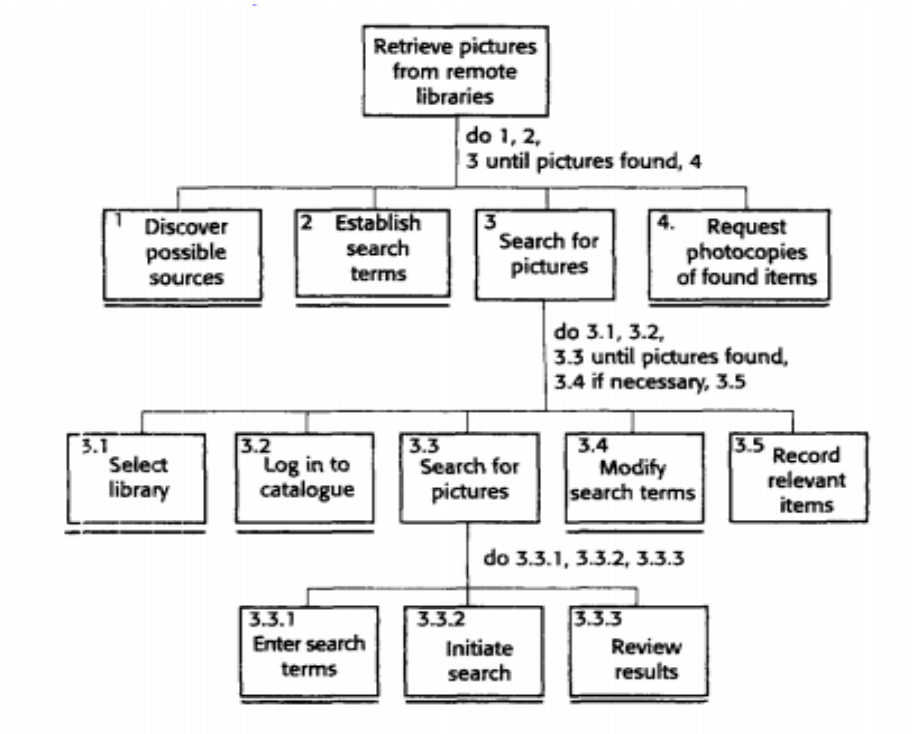


Figure 12.15

As you become more aware of how they work, their concerns and their constraints, your design can take these into account. This means that your initial designs (which you will refine through prototyping anyway) are more likely to be acceptable to users and so convince them to become engaged in the process of design refinement.

12.5.1 Analysis Techniques

There are various forms of task analysis (Diaper, 1989), but the most commonly used is Hierarchical Task Analysis (HTA). HTA was originally developed to help with writing user manuals, but it can also be used to identify what users do to achieve some goal. In HTA, a high-level task is broken down into subtasks, and plans are identified that specify what might happen in a specific situation. Starting with a user goal, you draw a hierarchy showing what has to be done to achieve that goal. Figure 12.15 illustrates this approach using the library scenario introduced in Figure 12.14. In the HTA notation, a line under a box normally indicates that it will not be decomposed into more detailed subtasks.

The advantage of HTA over natural language scenarios is that it forces you to consider each of the tasks and to decide whether these should be decomposed. With natural language scenarios, it is easy to miss important tasks. Scenarios also become long and boring to read if you want to add a lot of detail to them. The problem with this approach to describing user tasks is that it is best suited to tasks that are sequential processes. The notation becomes awkward when you try to model tasks that involve interleaved or concurrent activities or that involve a very large number of subtasks.

Furthermore, HTA does not record why tasks are done in a particular way or constraints on the user processes. You can get a partial view of user activities from HTA, but you need additional information to develop a fuller understanding of the UI design requirements. Normally, you collect information for HTA through observing and interviewing users. In this interviewing process, you can collect some of this additional information and record it alongside the task analyses. When interviewing to discover what users actually do, you should design interviews so that users can provide any information that they (rather than you) feel is relevant. This means you should not stick rigidly to prepared list of questions. Rather, your questions should be open ended and should encourage users to tell you why they do things as well as what they actually do. Interviewing, of course, is not just a way of gathering information for task analysis it is a general information-gathering technique.

You may decide to supplement individual interviews with group interviews or focus groups. The advantage of using focus groups is that users stimulate each other to provide information and may end up discussing different ways that they have developed of using systems. Task analysis focuses on how individuals work but, of course, most work is actually cooperative. People work together to achieve a goal, and users find it difficult to discuss how this cooperation actually takes place. Therefore, direct observation of how users work and use computer-based systems is an important additional technique of user analysis.

Air traffic control involves a number of control 'suites' where the suites controlling adjacent sectors of airspace are physically located next to each other. Flights in a sector are represented by paper strips that are fitted into wooden racks in an order that reflects their position in the sector. If there are not enough slots in the rack (i.e. when the airspace is very busy), controllers spread the strips out on the desk in front of the rack. When we were observing controllers, we noticed that controllers regularly glanced at the strip racks in the adjacent sector. We pointed this out to them and asked them why they did this. They replied that, when the adjacent controller has strips on his or her desk, then this means that a lot of flights will be entering their sector. They therefore tried to increase the speed of aircraft in the sector to 'clear space' for the incoming aircraft.

Figure 12.16 A report of observations of air traffic control

1. Controllers had to be able to see all flights in a sector (this was why they spread strip:; out on the desk). Therefore, we should avoid using scrolling displays where flights disappeared off the top or bottom of the display.

2. The interface should have some way of telling controllers how many flights are in adjacent sectors so that controllers can plan their work load. Checking adjacent sectors was an automatic controller action and it is very likely that they would not have mentioned this in discussions of the ATC process. It was only through direct observation that we discovered these important requirements.

None of these user analysis techniques, on their own, give you a complete picture of what users actually do. They are complementary approaches that you should use together to help you understand what users do and get insights into what might be an appropriate user interface design.

12.6 User interface prototyping

Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good! enough for expressing user interface requirements. Evolutionary or exploratory prototyping with end-user involvement is the only practical way to design and develop graphical user interfaces for software systems. Involving the user in the design and development process is an essential aspect of user-centered design (Norman and Draper, 1986), a design philosophy for interactive systems. The aim of prototyping is to allow users to gain direct experience with the interface. Most of us find it difficult to think abstractly about a user interface and to explain exactly what we want. However, when we are presented with examples, it is easy to identify the characteristics that we like and dislike.

Ideally, when you are prototyping a user interface, you should adopt a two-stage prototyping process:

1. Very early in the process, you should develop paper prototypes-mock-ups of screen designs-and walk through these with end-users.

2. You then refine your design and develop increasingly sophisticated automated prototypes, then make them available to users for testing and activity simulation.

Paper prototyping is a cheap and surprisingly effective approach to prototype

development (Snyder, 2003). You don't need to develop any executable software and the designs don't have to be drawn to professional standards. You can draw paper versions of the system screens that user interact with and design a set of scenarios describing how the system might be used. As a scenario progresses, you sketch the information that would be displayed and the options available to users. You then work through these scenarios with users to simulate how the system might be used. This is an effective way to get users' initial reactions to an interface design, the information they need from the system and how they would normally interact with the system.

Alternatively, you can use a storyboarding technique to present the interface design. A storyboard is a series of sketches that illustrate a sequence of interactions. This is less hands-on but can be more convenient when presenting the interface proposals to groups rather than individuals. After initial experiments with a paper prototype, you should implement a software prototype of the interface design.

The problem, of course, is that you need to have some system functionality with which the users can interact. If you are prototyping the ill very early in the system development process, this may not be available. To get around this problem, you can use 'Wizard of Oz' prototyping (see the web page for an explanation if you haven't seen the film). In this approach, users interact with what appears to be a computer system, but their inputs are actually channeled to a hidden person who simulates the system's responses. They can do this directly or by using some other system to compute the required responses.

In this case, you don't need to have any executable software apart from the proposed user interface. There are three approaches that you can use for user interface prototyping:

1. Script-driven approach

   If you simply need to explore ideas with users, you can use a script-driven approach such as you'd find in Macromedia Director. In this approach, you create screens with visual elements, such as buttons and menus, and associate a script with these elements. When the user interacts with these screens, the script is executed and the next screen is presented, showing them the Jesuits of their actions. There is no application logic involved.

2. Visual programming languages

3. Visual programming languages, such as Visual Basic, incorporate a powerful development environment, access to a range of reusable objects and a user-interface development system that allows interfaces to be created quickly, with components and scripts associated with interface objects.

These solutions, based on web browsers and languages such as Java, offer a ready-made user interface. You add functionality by associating segments of Java programs with the information to be displayed. These segments (called applets) are executed automatically when the page is loaded into the browser. This approach is a fast way to develop user interface prototypes, but there are inherent restrictions imposed by the browser and the Java security model.

Prototyping is obviously closely associated with interface evaluation. Formal evaluation is unlikely to be cost-effective for early prototypes, so what you are trying to achieve at this stage is a 'formative evaluation' where you look for ways in which the interface can be improved.

12.7 Interface evaluation

Interface evaluation is the process of assessing the usability of an interface and checking that it meets user requirements. Therefore, it should be part of the normal verification and validation process for software systems. For example, in a learnability specification, you might state that m operator who is familiar with the work supported should be able to use 80% of the system functionality after a three-hour training session. However, it is more common to specify usability (if it is specified at all) qualitatively rather than using metrics. You therefore usually have to use your judgement and experience in interface evaluation.

Systematic evaluation of a user interface design can be an expensive process involving cognitive scientists and graphics designers. You may have to design and carry out a statistically significant number of experiments with typical users. You may need to use specially constructed laboratories fitted with monitoring equipment. A user interface evaluation of this kind is economically unrealistic for systems developed by small organizations with limited resources.

Attribute	Description
Learnability	How long does it take a new user to become productive with the system?
Speed of operation	How well does the system response match the user's work practice?
Robustness	How tolerant is the system of user error?
Recoverability	How good is the system at recovering from user errors?
Adaptability	How closely is the system tied to a single model of work?

Figure 12.17 Usability attributes

There are a number of simpler, less expensive techniques of user interface evaluation that can identify particular user interface design deficiencies:

1. Questionnaires that collect information about what users thought of the interface;

2. Observation of users at work with the system and 'thinking aloud' about how they are trying to use the system to accomplish some task;

3. Video 'snapshots' of typical system use;

4. The inclusion in the software of code which collects information about the most used facilities and the most common errors.

Surveying users by questionnaire is a relatively cheap way to evaluate an interface. The questions should be precise rather than general. It is no use asking questions such as 'Please comment on the usability of the interface' as the responses will probably vary so much that you won't see any common trend. Rather, specific questions such as 'Please rate the understandability of the error messages on a scale from 1 to 5. A rating of 1 means very clear and 5 means incomprehensible' are better. They are both easier to answer and more likely to provide useful information to improve the interface. Users should be asked to rate their own experience and background when filling in the questionnaire. This allows the designer to find out whether users from any particular background have problems with the interface.

Questionnaires can even be used before any executable system is available if a paper mock-up of the interface is constructed and evaluated. Observation-based evaluation simply involves watching users as they use a system, looking at the facilities used, the errors made and so on. This can be supplemented by 'think aloud' sessions where users talk about what they are trying to achieve, how they understand the system and how they are trying to use the system to accomplish their objectives.

Relatively low-cost video equipment means that you can record user sessions for later analysis. Complete video analysis is expensive and requires a specially equipped evaluation suite with several cameras focused on the user and on the screen. However, video recording of selected user operations can be helpful in detecting problems. Other evaluation methods must be used to find out which operations cause user difficulties.

Analysis of recordings allows the designer to find out whether the interface requires too much hand movement (a problem with some systems is that users must regularly move their hand from keyboard to mouse) and to see whether unnatural eye movements are necessary. An interface that requires many shifts of focus may mean that the user makes more errors and misses parts of the display. Instrumenting code to collect usage statistics allows interfaces to be improved in a number of ways.

The most common operations can be detected. The interface can be reorganised so that these are the fastest to select. For example, if pop-up or pull-down menus are used, the most frequent operations should be at the top of the menu and destructive operations towards the bottom. Code instrumentation also allows error-prone commands to be detected and modified. Finally, it is easy to give users a 'gripe' command that they can use to pass messages to the tool designer. This makes users feel that their views are being considered.

The interface designer and other engineers can gain rapid feedback about individual problems. None of these relatively simple approaches to user interface evaluation is foolproof and they are unlikely to detect all user interface problems. However, the techniques can be used with a group of volunteers before a system is released without a large outlay of resources. Many of the worst problems of the user interface design can then be discovered and corrected.

RAPID SOFTWARE DEVELOPMENT

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions, and the emergence of competing products and services. Software is part of almost all business operations so new software is developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid development and delivery is therefore now often the most critical requirement for software systems. In fact, many businesses are willing to trade off software quality and compromise on requirements to achieve faster deployment of the software that they need.

Because these businesses are operating in a changing environment, it is often practically impossible to derive a complete set of stable software requirements. The initial requirements inevitably change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, the requirements are likely to change

quickly and unpredictably due to external factors. The software may then be out of date when it is delivered. Software development processes that plan on completely specifying the requirements and then designing, building, and testing the system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually prolonged and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, a plan-driven approach is the right one. However, in a fast-moving business environment, this can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential. The need for rapid system development and processes that can handle changing requirements has been recognized for some time. IBM introduced incremental development in the 1980s (Mills et al., 1980). The introduction of so-called fourth generation languages, also in the 1980s, supported the idea of quickly developing and delivering software (Martin, 1981).

However, the notion really took off in the late 1990s with the development of the notion of agile approaches such as DSDM (Stapleton, 1997), Scrum (Schwaber and Beedle, 2001), and extreme programming (Beck, 1999; Beck, 2000). Rapid software development processes are designed to produce useful software quickly. The software is not developed as a single unit but as a series of increments, with each increment including new system functionality. Although there are many approaches to rapid software development, they share some fundamental characteristics:

1. The processes of specification, design, and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document only defines the most important characteristics of the system.

2. The system is developed in a series of versions. End-users and other system stakeholders are involved in specifying and evaluating each version. They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icons on the interface. The system may then generate a web-based interface for a browser or an interface for a specific platform such as Microsoft Windows.

12.8 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, the use of analysis and design methods supported by CASE tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long lived software systems such as aerospace and government systems. This software was developed by large teams working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. These plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design has to change with the program. Dissatisfaction with these heavyweight approaches to software engineering led a number of software developers in the 1990s to propose new ‘agile methods’.

These allowed the development team to focus on the software itself rather than on its design and documentation. Agile methods universally rely on an incremental approach to software specification, development, and delivery. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut

down on process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used. The philosophy behind agile methods is reflected in the agile manifesto that was agreed on by many of the leading developers of these methods. This manifesto states:

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value: Individuals and interactions over processes and tools Working software over comprehensive documentation Customer collaboration over contract negotiation Responding to change over following a plan That is, while there is value in the items on the right, we value the items on the left more.

Although these agile methods are all based around the notion of incremental development and delivery, they propose different processes to achieve this. However, they share a set of principles, based on the agile manifesto, and so have much in common. These principles are shown in Figure 12.18.

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and so design the system to accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.

Figure 12.18 The principles of agile methods

Agile methods have been very successful for some types of system development:

- Product development where a software company is developing a small or medium-sized product for sale.
- Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.

In practice, the principles underlying agile methods are sometimes difficult to realize:

1. Although the idea of customer involvement in the development process is an attractive one, its success depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Frequently, the customer representatives are subject to other pressures and cannot take full part in the software development.

2. Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods, and therefore not interact well with other team members.

3. Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priori ties to different changes.

4. Maintaining simplicity requires extra work. Under pressure from delivery schedules, the team members may not have time to carry out desirable system simplifications.

5. Many organizations, especially large companies, have spent years changing their culture so that processes are defined and followed. It is difficult for them to move to a working model in which processes are informal and defined by development teams.

Another non-technical problem—that is a general problem with incremental development and delivery—occurs when the system customer uses an outside organization for system development. The software requirements document is usually part of the contract between the customer and the supplier. Because incremental specification is inherent in agile methods, writing contracts for this type of development may be difficult. Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. So long as all goes well, this benefits both the customer and the developer. However, if problems arise then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

There are only a small number of experience reports on using agile methods for software maintenance (Poole and Huisman, 2001). There are two questions that should be considered when considering agile methods and maintenance:

1. Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?

2. Can agile methods be used effectively for evolving a system in response to cus tomer change requests?

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is often not kept up to date and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. Agile practices therefore emphasize the importance of writing well-structured code and investing effort in code improvement. Therefore, the lack of documentation should not be a problem in maintaining systems developed using an agile approach. However, my experience of system maintenance suggests that the key document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. In this respect, the use of agile methods is likely to make subsequent system maintenance more difficult and expensive. Agile practices, used in the maintenance process itself, are likely to be effective, whether or not an agile approach has been used for system development. Incremental delivery, design for change and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process of software evolution.

12.9 Extreme programming

Extreme programming (XP) is perhaps the best known and most widely used of the agile methods. The name was coined by Beck (2000) because the approach was developed by pushing recognized good practice, such as iterative development, to ‘extreme’ levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated and tested in a day.

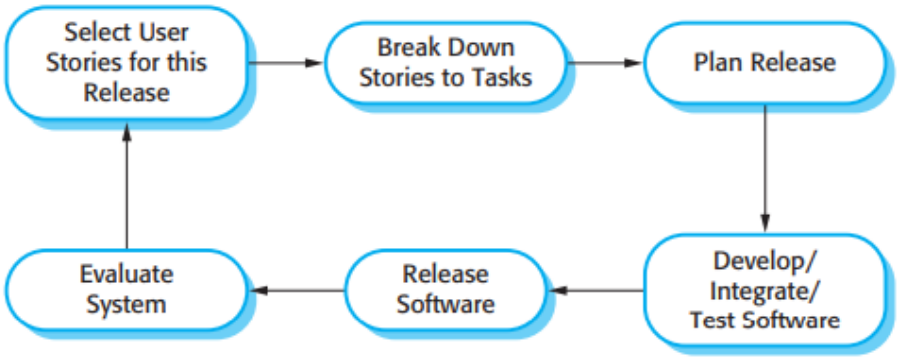


Figure 12.19 The extreme programming release cycle

In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system. Figure 12.19 illustrates the XP process to produce an increment of the system that is being developed.

Extreme programming involves a number of practices, summarized in Figure 12.20, which reflect the principles of agile methods:

Principle or practice	Description
Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development 'Tasks'.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

Figure 12.20 Extreme programming practices

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.
4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In an XP process, customers are intimately involved in specifying and prioritizing system requirements. The requirements are not specified as lists of required system functions. Rather, the system customer is part of the development team and discusses scenarios with other team members. Together, they develop a 'story card' that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the mental health care patient management system is shown in Figure 12.21.

Prescribing Medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose. If she wants to change the dose, she enters the dose and then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose and then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose and then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 12.21 A 'prescribing medication' story.

This is a short description of a scenario for prescribing medication for a patient. The story cards are the main inputs to the XP planning process or the ‘planning game’. Once the story cards have been developed, the development team breaks these down into tasks (Figure 12.22) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer. Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

Sometimes, during the planning game, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. In XP terms, this is a ‘spike’, an increment where no programming is done. There may also be ‘spikes’ to design the system architecture or to develop system documentation. Extreme programming takes an ‘extreme’ approach to incremental development. New versions of the software may be built several times per day and releases are delivered to customers roughly every two weeks. Release deadlines are never slipped; if there are development problems, the customer is consulted and functionality is removed from the planned release.

When a programmer builds the system to create a new version, he or she must run all existing automated tests as well as the tests for the new functionality. The new build of the software is accepted only if all tests execute successfully. This then becomes the basis for the next iteration of the system. A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn’t worth taking time to add generality to a program to cope with change. The changes anticipated often never materialize and completely different change requests may actually be made. Therefore, the XP approach accepts that changes will happen and reorganize the software when these changes actually occur.

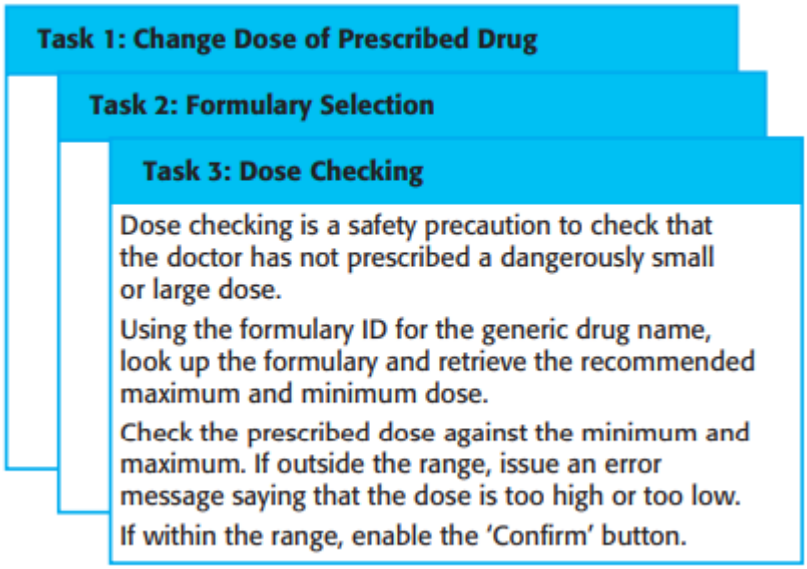


Figure 12.22 Examples of task cards for prescribing medication

A general problem with incremental development is that it tends to degrade the software structure, so changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Extreme programming tackles this problem by suggesting that the software should be constantly refactored. This means that the programming team look for possible improvements to the software and implement them immediately. When a team member sees code that can be improved, they make these improvements even in situations where there is no immediate need for them. Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of code with calls to methods defined in a program library. Program development environments, such as Eclipse (Carlson, 2005), include tools for refactoring which simplify the process of finding dependencies between code sections and making global code modifications. In principle then, the software should always be easy to understand and change as new stories are implemented.

In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require the architecture of the system to be modified. In practice, many companies that have adopted XP do not use all of the extreme programming practices listed in Figure 12.20. They pick

and choose according to their local ways of working. For example, some companies find pair programming helpful; others prefer to use individual programming and reviews. To accommodate different levels of skill, some programmers don't do refactoring in parts of the system they did not develop, and conventional requirements may be used rather than user stories. However, most companies who have adopted an XP variant use small releases, test-first development, and continuous integration.

12.9.1 Testing in XP

To avoid some of the problems of testing and system validation, XP emphasizes the importance of program testing. XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system. The key features of testing in XP are:

1. Test-first development,
2. Incremental test development from scenarios,
3. User involvement in the test development and validation, and
4. Use of automated testing frameworks.

Test-first development is one of the most important innovations in XP. Instead of writing some code and then writing tests for that code, you write the tests before you write the code. This means that you can run the test as the code is being written and discover problems during development. Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. This approach can be adopted in any process in which there is a clear relationship between a system requirement and the code implementing that requirement. In XP, you can always see this link because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation. The adoption of test-first development in XP has led to more general test-driven approaches to development (Astels, 2003). I discuss these in Chapter 8. In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of 'test-lag'. This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

User requirements in XP are expressed as scenarios or stories and the user prioritizes these for development. The development team assesses each scenario and breaks it down into tasks. For example, some of the task cards developed from the story card for prescribing medication (Figure 12.21) are shown in Figure 12.22. Each task generates one or more unit tests that check the implementation described in that task. Figure 12.23 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

Test 4: Dose Checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose × frequency is too high and too low.
4. Test for inputs where single dose × frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 12.23 Test case description for dose checking

In XP, acceptance testing, like development, is incremental. The customer who is part of the team writes tests as development proceeds. All new code is therefore validated to ensure that it is what the customer needs. For the story in Figure 12.21, the acceptance test would involve scenarios where

a) the dose of a drug was changed,

b) a new drug was selected, and

c) the formulary was used to find a drug. In practice, a series of acceptance tests rather than a single test are normally required.

Relying on the customer to support acceptance test development is sometimes a major difficulty in the XP testing process. People adopting the customer role have very limited available time and may not be able to work full-time with the development team. The customer may feel that providing the requirements was enough of a contribution and so may be reluctant to get involved in the testing process. Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing

components should be stand alone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Massol and Husted, 2003) is a widely used example of an automated testing framework. As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually results in a large number of tests being written and executed. However, this approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.

2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.

3. It difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests written after development, then undetected bugs may be delivered in the system release.

12.9.2 Pair programming

Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software. They actually sit together at the same workstation to develop the software. However, the same pairs do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process. The use of pair programming has a number of advantages:

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's (1971) idea of egoless programming where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has

collective responsibility for resolving these problems.

2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (covered in Chapter 24) are very successful in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Although pair programming is a less formal process that probably doesn't find as many errors as code inspections, it is a much cheaper inspection process than formal program inspections.

3. It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. There have been various studies of the productivity of paid programmers with mixed results. Using student volunteers,

However, more experienced programmers found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

12.10 Rapid Application Development

Although agile methods as an approach to iterative development have received a great deal of attention in the last few years, business systems have been developed iteratively fer many years using rapid application development techniques. Rapid application development (RAD) techniques evolved from so-called fourth-generation languages in the I980s and are used for developing applications that are data-intensive. Consequently, they are usually organized as a set of tools that allow data to be created, searched, displayed and presented in reports. Figure 12.24 illustrates a typical organization for a RAD system.

The tools that are included in a RAD environment are:

1. A database programming language that embeds knowledge of the database structure; and includes fundamental database manipulation operations. SQL is the standard database programming language. The SQL commands may be input directly or generated automatically from forms filled in by 2n end-user.

2. An interface generator, which is used to create forms for data input and display.

3. links to office applications such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.

4. A report generator, which is used to define and create reports from information in the database.

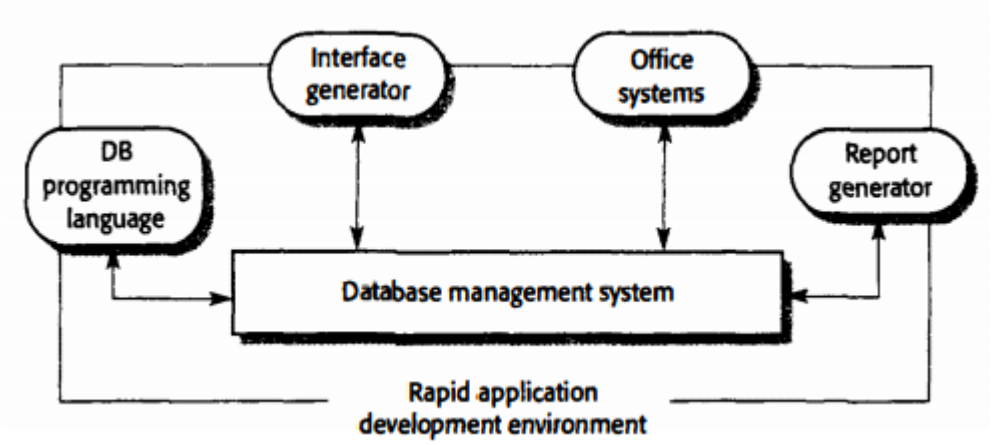


Figure 12.24 A rapid application development environment

Standard forms are used for input and output. RAD systems are geared towards producing interactive applications that rely on abstracting information from an organizational database, presenting it to end-users on their terminal or workstation, and updating the database with changes made by users. Many business applications rely on structured forms for input and output, so RAD environments provide powerful facilities for screen definition and report generation. Screens are often defined as a series of linked forms (in one application we studied, there were 137 form definitions) so the screen generation system must provide for:

1. Interactive Jann definition where the developer defines the fields to be displayed and how these are to be organized.

2. Form linking where the developer can specify that particular inputs cause further forms to be displayed.

3. Field verification where the developer defines allowed ranges for values input to form fields.

All RAD environments now support the development of database interfaces based on web browsers. These allow the database to be accessed from anywhere with a valid Internet connection. This reduces training and software costs and allows external users to have access to a database. However, the inherent limitations of web browsers and Internet protocols mean that this approach may be unsuitable for systems where very fast, interactive responses are required. Most RAD systems now also include visual programming tools that allow the system to be developed interactively. Rather than write a sequential program, the system developer manipulates graphical icons representing functions, data or user interface components, and associates processing scripts with these icons. An executable program is generated automatically from the visual representation of the system.

12.11 Software Prototyping

Software prototyping is becoming very popular as a software development model, as it enables to understand customer requirements at an early stage of development. It helps get valuable feedback from the customer and helps software designers and developers understand about what exactly is expected from the product under development.

What is Software Prototyping?

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation. Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Following is the stepwise approach to design a software prototype:

- Basic Requirement Identification: This step involves understanding the very basics product requirements especially in terms of user interface. The more intricate details of the internal design and external aspects like performance and security can be ignored at this stage.
- Developing the initial Prototype: The initial Prototype is developed in this stage, where the very basic requirements are showcased and user interfaces are provided. These features may not exactly work in the same manner internally in the actual software developed and the workarounds are used to give the same look and feel to the customer in the prototype developed.
- Review of the Prototype: The prototype developed is then presented to the customer and the other important stakeholders in the project. The feedback is collected in an organized manner and used for further enhancements in the product under development.
- Revise and enhance the Prototype: The feedback and the review comments are discussed during this stage and some negotiations happen with the customer based on factors like , time and budget constraints and technical feasibility of actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until customer expectations are met.

Prototypes can have horizontal or vertical dimensions. Horizontal prototype displays the user interface for the product and gives a broader view of the entire system, without concentrating on internal functions. A vertical prototype on the other side is a detailed elaboration of a specific function or a sub system in the product. The purpose of both horizontal and vertical prototype is different. Horizontal prototypes are used to get more information on the user interface level and the business requirements. It can even be presented in the sales demos to get business in the market. Vertical prototypes are technical in nature and are used to get details of the exact functioning of the sub systems. For example, database requirements, interaction and data processing loads in a given sub system.

Software Prototyping Types

There are different types of software prototypes used in the industry. Following are the major software prototyping types used widely:

1. Throwaway/Rapid Prototyping: Throwaway prototyping is also called as rapid or close ended prototyping. This type of prototyping uses very little efforts with minimum requirement analysis to build a prototype. Once the actual requirements are understood, the prototype is discarded and the actual system is developed with a much clear understanding of user requirements.

2. Evolutionary Prototyping: Evolutionary prototyping also called as breadboard prototyping is based on building actual functional prototypes with minimal functionality in the beginning. The prototype developed forms the heart of the future prototypes on top of which the entire system is built. Using evolutionary prototyping only well understood requirements are included in the prototype and the requirements are added as and when they are understood.

3. Incremental Prototyping: Incremental prototyping refers to building multiple functional prototypes of the various sub systems and then integrating all the available prototypes to form a complete system.

4. Extreme Prototyping : Extreme prototyping is used in the web development domain. It consists of three sequential phases. First, a basic prototype with all the existing pages is presented in the html format. Then the data processing is simulated using a prototype services layer. Finally the services are implemented and integrated to the final prototype. This process is called Extreme Prototyping used to draw attention to the second phase of the process, where a fully functional UI is developed with very little regard to the actual services.

Software Prototyping Application

Software Prototyping is most useful in development of systems having high level of user interactions such as online systems. Systems which need users to fill out forms or go through various screens before data is processed can use prototyping very effectively to give the exact look and feel even before the actual software is developed. Software that involves too much of data processing and most of the functionality is internal with very little user interface does not usually benefit from prototyping. Prototype development could be an extra overhead in such projects and may need lot of extra efforts.

12.12 Summary:

- User interface principles covering user familiarity, consistency, minimal surprise, recoverability, user guidance and user diversity help guide the design of user interfaces. Styles of interaction with a software system include direct manipulation, menu systems, form fill-in, command languages and natural language.
- Graphical information display should be used when it is intended to present trends and approximate values. Digital display should only be used when precision is required. Color should be used sparingly and consistently in user interfaces. Designers should take account of the fact that a significant number of people are color-blind. The user interface design process includes sub-processes concerned with user analysis, interface prototyping and interface evaluation.
- The aim of user analysis is to sensitive designers to the ways in which users actually work. You should use different techniques-task analysis, interviewing and observation-during user analysis.
- User interface prototype development should be a staged process with early prototypes based on paper versions of the interface that, after initial evaluation and feedback, are used as a basis for automated prototypes. The goals of user interface evaluation are to obtain feedback on how a UI design can be improved and to a user whether an interface meets its usability requirements.
- Agile methods are incremental development methods that focus on rapid development, frequent releases of the software, reducing process overheads, and producing high-quality code. They involve the customer directly in the development process.
- Extreme programming is a well-known agile method that integrates a range of good programming practices such as frequent releases of the software, continuous software improvement, and customer participation in the development team.
- The Software Prototyping refers to building software application prototypes which display the functionality of the product under development but may not actually hold the exact logic of the original software.

12.13 Self-Assessment Questions:

- Why do we need UI design?
- Explain the design issues
- What is the design process?
- Write a short note on:
  a) User Analysis
  b) RAD
  c) Testing in XP
  d) Interface Evaluation
- What is Software Prototyping?

12.14 List of References:

- Software Engineering, Ian Somerville, 8th edition, Pearson Education
- Software Engineering, Pankaj Jalote
- Software Engineering, A practitioner's approach, Roger Pressman, Tata McGraw-Hill

❖ ❖ ❖

13

COMPONENT BASED SOFTWARE ENGINEERING

Unit Structure:

13.0 Objective

The objective of this chapter is to describe a software development process based on the composition of reusable, standardised components.

When you have read this chapter, you will:

- know that component-based software engineering is concerned with developing standardised components based on a component model and composing these into application systems;
- understand what is meant by a component and a component model;
- know the principal activities in the CBSE process and understand why you have to make requirements compromises so that components can be reused;
- understand some of the difficulties and problems that arise during the process of component composition.

13.1 Components

Different people have proposed definitions of a software component.

1. Councill and Heineman (Councill and Heineman, 2001) define a component as:

   "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."

   This definition is essentially based on standards,a software unit that conforms

   to these standards is a component.

2. Szyperski (Szyperski, 2002) does not mention standards in his definition of a component but focuses instead on the key characteristics of components:

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Szyperski also states that a component has no externally observable state. This means that copies of components are indistinguishable.

All these definitions have in common is that they agree that components are independent and that they are the fundamental unit of composition in a system. A complete definition of a component can be derived from both of these proposals.

Component characteristics are as follow:

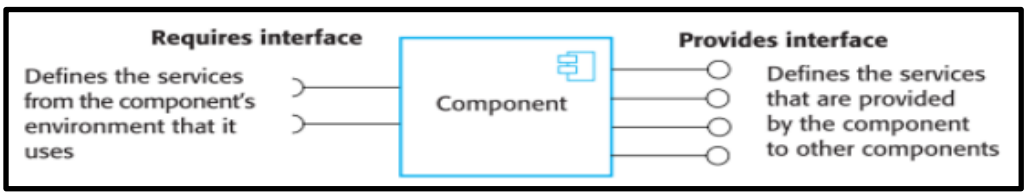
- **Standardised**
  - Component standardisation means that a component used in a CBSE process has to conform to some standardised component model.
  - This model may define component interfaces, component metadata, documentation, composition and deployment.
- **Independent**
  - A component should be independent—it should be possible to compose and deploy it without having to use other specific components.
  - In situations where the component needs externally provided services, these should be explicitly set out in a 'requires' interface specification.

- **Composable**
  - For a component to be composable, all external interactions must take place through publicly defined interfaces.
  - In addition, it must provide external access to information about itself, such as its methods and attributes.
- **Deployable**
  - To be deployable, a component has to be self-contained and must be able to operate as a standalone entity on a component platform that implements the component model.
  - This usually means that the component is binary and does not have to be compiled before it is deployed.
- **Documented**
  - Components have to be fully documented so that potential users can decide whether or not the components meet their needs.
  - The syntax and, ideally, the semantics of all component interfaces have to be specified.

Viewing a component as a service provider emphasises two critical characteristics of a reusable component:

1. The component is an independent executable entity. Source code is not available, so the component does not have to be compiled before it is used with other system components.
2. The services offered by a component are made available through an interface, and all interactions are through that interface. The component interface is expressed in terms of parameterised operations and its internal state is never exposed.

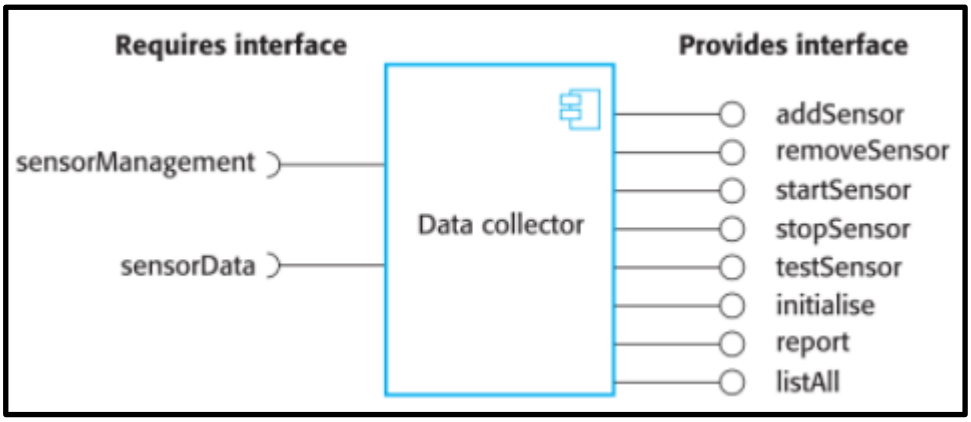
Components are defined by their interfaces and, in the most general cases, can be thought of as having two related interfaces, as shown in following Figure:



1. A *provides* interface defines the services provided by the component. The provides interface, essentially, is the component API. It defines the methods that can be called by a user of the component. Provides interfaces are indicated by a circle at the end of a line from the component icon.

2. A *requires* interface specifies what services must be provided by other components in the system. If these are not available, then the component will not work. This does not compromise the independence or deployability of the component because it is not required that a specific component should be used to provide the services. Requires interfaces are indicated by a semi-circle at the end of a line from the component icon. Notice that provides and required interface icons can fit together like a ball and socket.

A model of a data collector component



Components are usually developed using an object-oriented approach, but they differ from objects in a number of important ways:

1. **Components are deployable entities**

   - ❑ That is, they are not compiled into an application program but are installed directly on an execution platform.
   - ❑ The methods and attributes defined in their interfaces can then be accessed by other components.

2. **Components do not define types**

   - ❑ A class definition defines an abstract data type and objects are instances of that type.
   - ❑ A component is an instance, not a template that is used to define an instance.

3. **Component implementations are opaque**

   - ❑ Components are, in principle at least, completely defined by their interface specification.
   - ❑ The implementation is hidden from component users.
   - ❑ Components are often delivered as binary units so the buyer of the component does not have access to the implementation.

4. **Components are language-independent**

   - ❑ Object classes have to follow the rules of a particular object-oriented programming language and, generally, can only interoperate with other classes in that language.
   - ❑ Although components are usually implemented using object-oriented languages such as Java, you can implement them in non-object-oriented programming languages.

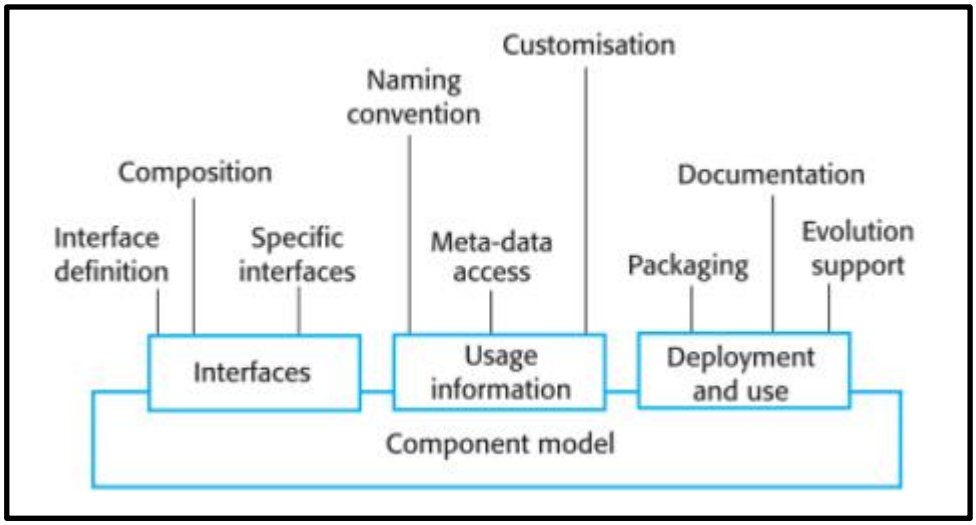
5. **Components are standardised**

   - ❑ Unlike object classes that you can implement in any way, components must conform to some component model that constrains their implementation.

13.2 Component models

- A component model is a definition of standards for component implementation, documentation and deployment.
- These standards are for component developers to ensure that components can interoperate.
- They are also for providers of component execution infrastructures who provide middleware to support component operation.

Basic elements of a component model



- The basic elements of an ideal component model are discussed by Weinreich and Sametinger (Weinreich and Sametinger, 2001).
- This diagram shows that the elements in a component model can be classified as elements relating to the component interfaces, elements relating to information that you need to use the component in a program and elements concerned with component deployment.
- The defining elements of a component are its interfaces.
- The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, that should be included in an interface definition.
- The model should also specify the language used to define the interfaces.
- In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them.
- Component model implementations normally include specific ways (such as the use of a reflection interface in Java) to access this component metadata.
- An important part of a component model is a definition of how components should be packaged for deployment as independent, executable entities.
- The services provided by a component model implementation fall into two categories:

1. **Platform services**

   - ❑ These fundamental services enable components to communicate with each other.
   - ❑ CORBA is an example of a component model platform.

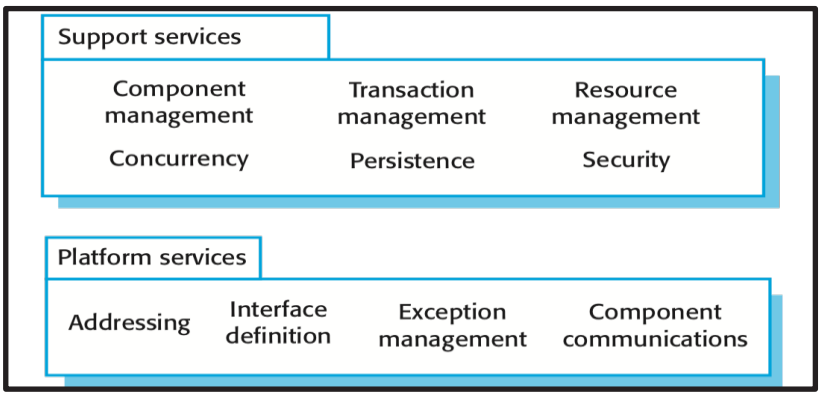
2. **Horizontal services**

   - ❑ These application-independent services are likely to be used by many different components.
   - ❑ The availability of these services reduces the costs of component development and means that potential component incompatibilities can be avoided.

Elements of a component model:

- **Interfaces**
  - Components are defined by specifying their interfaces.
  - The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.
- **Usage**
  - In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them.
  - This has to be globally unique.
- **Deployment**
  - The component model includes a specification of how components should be packaged for deployment as independent, executable entities.
- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide: **platform services** that allow components written according to the model to communicate, and **support services** that are application-independent services used by different components.

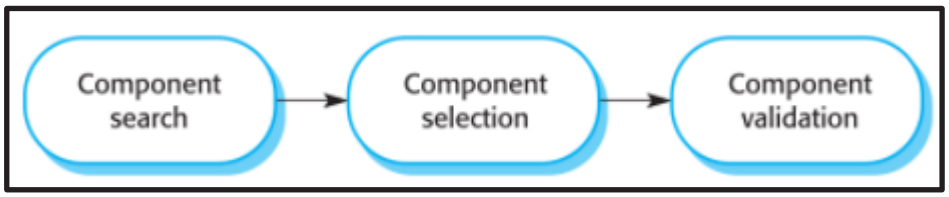
- To use services provided by a model, components are deployed in a **container**. This is a set of interfaces used to access the service implementations.



13.3 The CBSE Process

- Following Figure shows the principal sub-activities within a CBSE process.
- Some of the activities within this process, such as the initial discovery of user requirements, are carried out in the same way as in other software processes.
- The essential differences between this process and software processes based on original software development are:

The CBSE process



- CBSE processes are software processes that support component-based software engineering. They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.
- There are two types of CBSE processes:
  - **CBSE for reuse**

  ➔ is concerned with developing components or services that will be reused in other applications.

➔ It usually involves generalizing existing components.

○ **CBSE with reuse**

➔ is the process of developing new applications using existing components and services.

- **CBSE for reuse** focuses on component and service development.
- Components developed for a specific application usually have to be generalised to make them reusable.
- A component is most likely to be reusable if it associated with a stable domain abstraction (business object).
- For example, in a hospital stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

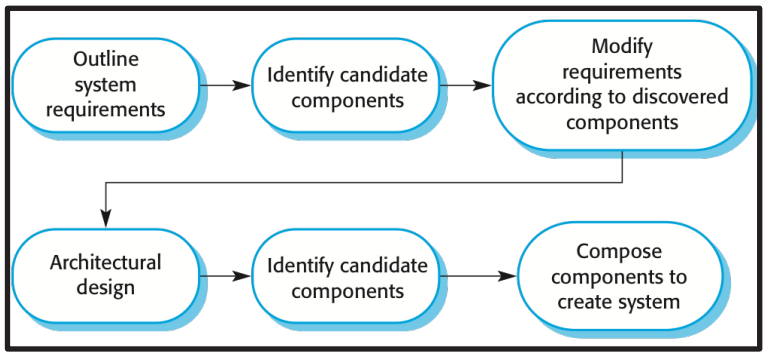
Component reusability:

- Should reflect stable domain abstractions;
- Should hide state representation;
- Should be as independent as possible;
- Should publish exceptions through the component interface.
- There is a **trade-off between reusability and usability**.
- The more general the interface, the greater the reusability but it is then more complex and hence less usable.

To make an existing component reusable:

- Remove application-specific methods.
- Change names to make them general.
- Add methods to broaden coverage.
- Make exception handling consistent.
- Add a configuration interface for component adaptation.
- Integrate required components to reduce dependencies.

- Existing **legacy systems** that fulfill a useful business function can be re-packaged as components for reuse.
- This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.
- Although costly, this can be much less expensive than rewriting the legacy system.
- The **development cost** of reusable components may be higher than the cost of specific equivalents.
- This extra reusability enhancement cost should be an organization rather than a project cost.
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.
- **Component management** involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions.
- A company with a reuse program may carry out some form of component certification before the component is made available for reuse.
- Certification means that someone apart from the developer checks the quality of the component.
- **CBSE with reuse** process has to find and integrate reusable components.
- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
- This involves:
    - ➔ Developing outline requirements;
    - ➔ Searching for components then modifying requirements according to available functionality;
    - ➔ Searching again to find if there are better components that meet the revised requirements;
    - ➔ Composing components to create the system.



Component identification issues:

➔ You need to be able to **trust** the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.

➔ Different groups of components will satisfy different **requirements**.

➔ **Validation.** The component specification may not be detailed enough to allow comprehensive tests to be developed. Components may have unwanted functionality. How can you test this will not interfere with your application?

- Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests.
- The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests.
- As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need.

13.4 Component Composition

- Component composition is the process of **assembling components to create a system**.
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write 'glue code' to integrate components.

- **Three types of component composition:**

1. **Sequential composition**

   ❑ where the composed components are executed in sequence.

   ❑ This involves composing the provides interfaces of each component.

2. **Hierarchical composition**

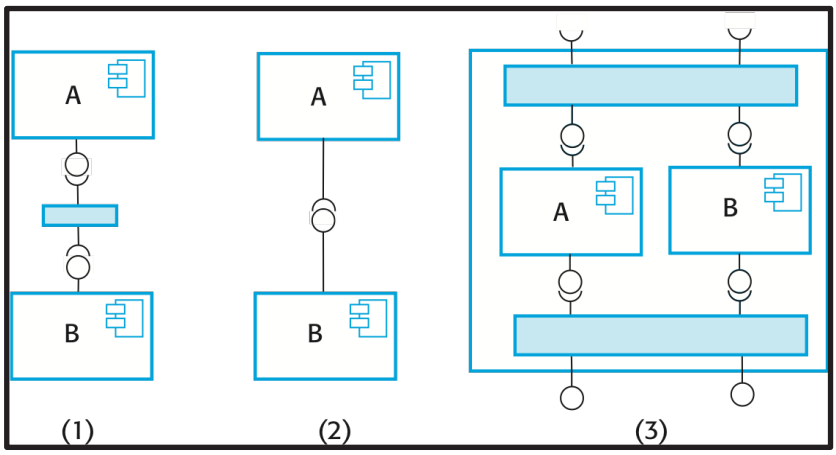
   ❑ where one component calls on the services of another.

   ❑ The provides interface of one component is composed with the requires interface of another.

3. **Additive composition**

   ❑ where the interfaces of two components are put together to create a new component.

   ❑ Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.



- When components are developed independently for reuse, their interfaces are often incompatible.
- **Three types of incompatibility can occur:**
  - **Parameter incompatibility**

    where operations have the same name but are of different types.
  - **Operation incompatibility**

    where the names of operations in the composed interfaces are different.

- **Operation incompleteness**

  where the provides interface of one component is a subset of the requires interface of another.

- Adaptor components address the problem of component incompatibility by reconciling the interfaces of the components that are composed.
- Different types of adaptor are required depending on the type of composition.
- When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.
- You need to make decisions such as:

  a. What composition of components is effective for delivering the functional requirements?

  b. What composition of components allows for future change?

  c. What will be the emergent properties of the composed system?

13.5 Summary:

- Component-based software engineering is a reuse-based approach to defining, implementing and composing loosely coupled independent components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by a set of public interfaces. Components can be combined with other components without reference to their implementation and can be deployed as an executable unit.
- A component model defines a set of standards for components, including interface standards, usage standards and deployment standards. The implementation of the component model provides a set of horizontal services that may be used by all components.
- During the CBSE process, you have to interleave the processes of requirements engineering and system design. You have to trade-off desirable requirements against the services that are available from existing reusable components.

- Component composition is the process of 'wiring' components together to create a system. Types of composition include sequential composition, hierarchical composition and additive composition.
- When composing reusable components that have not been written for your application, you normally need to write adaptors or 'glue code' to reconcile the different component interfaces.
- When choosing compositions, you have to consider the required functionality of the system, the non-functional requirements and the ease with which one component can be replaced by another when the system is changed.

13.6 Exercise

Answer the following:

1. Why is it important that all component interactions are defined through requires and provides interfaces?
2. Why is it important that components should be based on a standard component model?
3. Explain why it is very difficult to validate a reusable component without the component source code. In what ways would a formal component specification simplify the problems of validation?

❖ ❖ ❖

14

VERIFICATION AND VALIDATION

Unit Structure:

14.0 Objective

- The objective of this chapter is to introduce software verification and validation techniques. This chapter helps us to understand the distinctions between software verification and software validation.
- Software inspection is introduced to program inspections as a method of discovering defects in programs.
- We can understand what automated static analysis is and how it is used in verification and validation.
- Also understand how static verification is used in the development process. To describe the Cleanroom software development process

14.1 Verification and Validation

Verification and Validation is the process of investigating that a software system satisfies specifications and standards and it fulfills the required purpose.

Barry Boehm described verification and validation as the following:

Verification: *Are we building the product right?*

Validation: *Are we building the right product?*

Verification:

- ➔ Verification is the process of checking that a software achieves its goal without any bugs.
- ➔ It is the process to ensure whether the product that is developed is right or not.
- ➔ It verifies whether the developed product fulfills the requirements that we have.
- ➔ Verification is **Static Testing**.
- ➔ Activities involved in verification:

1. Inspections
2. Reviews
3. Walkthroughs
4. Desk-checking

Validation:

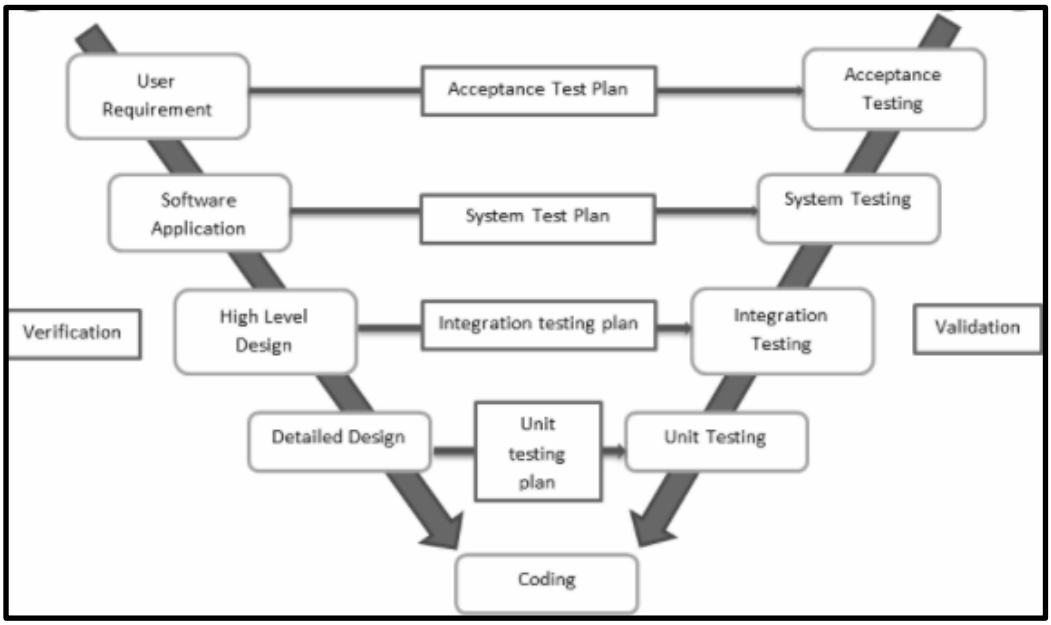
- ➔ Validation is the process of checking whether the software product is up to the mark.
- ➔ In other words the product has high level requirements.
- ➔ It is the process of checking the validation of product i.e. it checks what we are developing is the right product.

➔ It is validation of actual and expected products.

➔ Validation is the **Dynamic Testing**.

Verification is followed by Validation.

Diagramatic representation of Verification and validation model:



Difference between verification and validation testing:

Verification	**Validation**
We check whether we are developing the right product or not.	We check whether the developed product is right.
Verification is also known as **static testing**.	Validation is also known as **dynamic testing**.
Verification includes different methods like Inspections, Reviews, and Walkthroughs.	Validation includes testing like functional testing, system testing, integration, and User acceptance testing.

It is a process of checking the work-products (not the final product) of a development cycle to decide whether the product meets the specified requirements.	It is a process of checking the software during or at the end of the development cycle to decide whether the software follow the specified business requirements.
Quality assurance comes under verification testing.	**Quality control** comes under validation testing.
The execution of code does not happen in the verification testing.	In validation testing, the execution of code happens.
In verification testing, we can find the bugs early in the development phase of the product.	In the validation testing, we can find those bugs, which are not caught in the verification process.
Verification testing is executed by the Quality assurance team to make sure that the product is developed according to customers' requirements.	Validation testing is executed by the testing team to test the application.
Verification is done before the validation testing.	After verification testing, validation testing takes place.
In this type of testing, we can verify that the inputs follow the outputs or not.	In this type of testing, we can validate that the user accepts the product or not.

KEY DIFFERENCE

- Verification process includes checking of documents, design, code and program whereas the Validation process includes testing and validation of the actual product.
- Verification does not involve code execution while Validation involves code execution.

- Verification uses methods like reviews, walkthroughs, inspections and desk-checking whereas Validation uses methods like black box testing, white box testing and non-functional testing.
- Verification checks whether the software confirms a specification whereas Validation checks whether the software meets the requirements and expectations.
- Verification finds the bugs early in the development cycle whereas Validation finds the bugs that verification can not catch.
- Verification process targets software architecture, design, database, etc. while the Validation process targets the actual software product.
- Verification is done by the QA team while Validation is done by the involvement of testing team with QA team.
- Verification process comes before validation whereas the Validation process comes after verification.

14.2 Software Inspections:

- Software inspection is a static V & V process in which a software system is reviewed to find errors, omissions and anomalies.
- It focuses on source code, but any readable representation of the software such as its requirements or a design model can be inspected.
- While inspecting a system, you use knowledge of the system, its application domain and the programming language or design model to discover errors.
- There are three major advantages of inspection over testing:

1. *During testing, errors can mask (hide) other errors.*

   ❑ Once one error is discovered, you can never be sure if other output anomalies are due to a new error or are side effects of the original error.

   ❑ Because inspection is a static process, you don't have to be concerned with interactions between errors.

   ❑ Consequently, a single inspection session can discover many errors in a system.

2. *Incomplete versions of a system can be inspected without additional costs.*

   - ❑ If a program is incomplete, then you need to develop specialised test harnesses to test the parts that are available.
   - ❑ This obviously adds to the system development costs.

3. *Searching for program defects, an inspection can also consider broader quality attributes of a program*

   - ❑ such as compliance with standards, portability and maintainability.
   - ❑ You can look for inefficiencies, inappropriate algorithms and poor programming style that could make the system difficult to maintain and update.

- There have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing.
- Fagan (Fagan, 1986) reported that more than 60% of the errors in a program can be detected using informal program inspections. Mills et al. (Mills, et al., 1987) suggest that a more formal approach to inspection based on correctness arguments can detect more than 90% of the errors in a program.
- This technique is used in the Cleanroom process. Selby and Basili (Selby, et al., 1987) empirically compared the effectiveness of inspections and testing.
- They found that static code reviewing was more effective and less expensive than defect testing in discovering program faults.
- Gilb and Graham (Gilb and Graham, 1993) have also found this to be true.
- Reviews and testing each have advantages and disadvantages and should be used together in the verification and validation process.
- Gilb and Graham (Gilb and Graham, 1993) have also found this to be true that static code reviewing was more effective and less expensive.
- They suggest that one of the most effective uses of reviews is to review the test cases for a system. Reviews can discover problems with these tests and can help design more effective ways to test the system.
- You can start system V & V with inspections early in the development process, but once a system is integrated, you need testing to check its emergent properties and that the system's functionality is what the owner of the system really wants.

Roles in the inspection process with Description are as follow:

1. *Author or owner*

   - ❑ The programmer or designer responsible for producing the program or document.
   - ❑ Responsible for fixing defects discovered during the inspection process.

2. *Inspector*

   - ❑ Finds errors, omissions and inconsistencies in programs and documents.
   - ❑ May also identify broader issues that are outside the scope of the inspection team.

3. *Reader*

   - ❑ Presents the code or document at an inspection meeting.

4. *Scribe*

   - ❑ Records the results of the inspection meeting.

5. *Chairman or moderator*

   - ❑ Manages the process and facilitates the inspection.
   - ❑ Reports process results to the chief moderator.

6. *Chief moderator*

   - ❑ Responsible for inspection process improvements, checklist updating, standards development, etc.

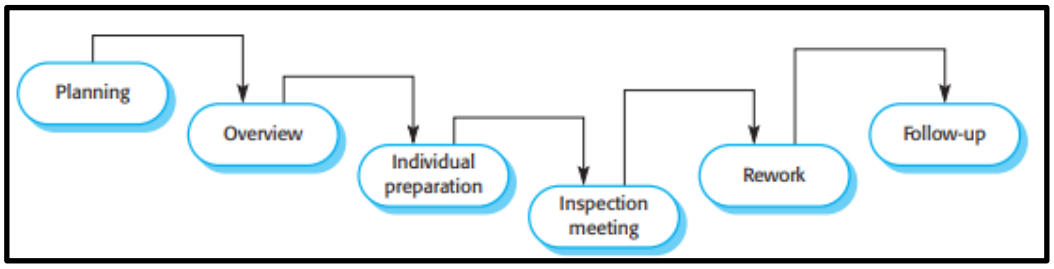
The activities in the inspection process are as follow:

Before a program inspection process begins, it is essential that,

1. You have a precise specification of the code to be inspected. It is impossible to inspect a component at the level of detail required to detect defects without a complete specification.
2. The inspection team members are familiar with the organisational standards.

3. An up-to-date, compilable version of the code has been distributed to all team members. There is no point in inspecting code that is 'almost complete' even if a delay causes schedule disruption.

The inspection process:



- During an inspection, a checklist of common programmer errors is often used to focus the discussion.
- You need different checklists for different programming languages because each language has its own characteristic errors.
- Humphrey (Humphrey, 1989), in a comprehensive discussion of inspections, gives a number of examples of inspection checklists.
- This checklist varies according to programming language because of the different levels of checking provided by the language compiler.
- For example, a Java compiler checks that functions have the correct number of parameters, a C compiler does not.
- Gilb and Graham (Gilb and Graham, 1993) emphasise that each organisation should develop its own inspection checklist based on local standards and practices.

Inspection checks with different Fault class are as follow:

1. *Data faults:*

   ❑ Are all program variables initialised before their values are used?

   ❑ Have all constants been named?

   ❑ Should the upper bound of arrays be equal to the size of the array or Size -1?

   ❑ If character strings are used, is a delimiter explicitly assigned?

   ❑ Is there any possibility of buffer overflow?

2. *Control faults:*

   - ❑ For each conditional statement, is the condition correct?
   - ❑ Is each loop certain to terminate?
   - ❑ Are compound statements correctly bracketed?
   - ❑ In case statements, are all possible cases accounted for?
   - ❑ If a break is required after each case in case statements, has it been included?

3. *Input/output faults:*

   - ❑ Are all input variables used?
   - ❑ Are all output variables assigned a value before they are output?
   - ❑ Can unexpected inputs cause corruption?

4. *Interface faults:*

   - ❑ Do all function and method calls have the correct number of parameters?
   - ❑ Do formal and actual parameter types match?
   - ❑ Are the parameters in the right order?
   - ❑ If components access shared memory, do they have the same model of the shared memory structure?

5. *Storage management faults:*

   - ❑ If a linked structure is modified, have all links been correctly reassigned?
   - ❑ If dynamic storage is used, has space been allocated correctly?
   - ❑ Is space explicitly de-allocated after it is no longer required?

6. *Exception management faults:*

   - ❑ Have all possible error conditions been taken into account?

14.3 Automated Static Analysis

- Inspections are one form of static analysis where you examine the program without executing it.
- Inspections are often driven by checklists of errors and heuristics that identify common errors in different programming languages.
- For some errors and heuristics(an approach to problem solving or self-discovery), it is possible to automate the process of checking programs against this list, which has resulted in the development of automated static analysers for different programming languages.
- Static analysers are software tools that scan the source text of a program and detect possible faults and anomalies.
- They parse the program text and thus recognise the types of statements in the program.
- They can then detect whether statements are well formed, make inferences about the control flow in the program and, in many cases, compute the set of all possible values for program data.
- They complement the error detection facilities provided by the language compiler.
- They can be used as part of the inspection process or as a separate V & V process activity.
- The intention of automatic static analysis is to draw an inspector's attention to anomalies in the program, such as variables that are used without initialisation, variables that are unused or data whose value could go out of range.

The stages involved in static analysis include:

1. *Control flow analysis*

    - ❑ This stage identifies and highlights loops with multiple exit or entry points and unreachable code.
    - ❑ Unreachable code is code that is surrounded by unconditional goto statements or that is in a branch of a conditional statement where the guarding condition can never be true.

2. *Data use analysis*

   - ❑ This stage highlights how variables in the program are used.
   - ❑ It detects variables that are used without previous initialisation, variables that are written twice without an intervening assignment and variables that are declared but never used.
   - ❑ Data use analysis also discovers ineffective tests where the test condition is redundant. Redundant conditions are conditions that are either always true or always false.

3. *Interface analysis*

   - ❑ This analysis checks the consistency of routine and procedure declarations and their use.
   - ❑ It is unnecessary if a strongly typed language such as Java is used for implementation as the compiler carries out these checks.
   - ❑ Interface analysis can detect type errors in weakly typed languages like FORTRAN and C.
   - ❑ Interface analysis can also detect functions and procedures that are declared and never called or function results that are never used.

4. *Information flow analysis*

   - ❑ This phase of the analysis identifies the dependencies between input and output variables.
   - ❑ While it does not detect anomalies, it shows how the value of each program variable is derived from other variable values.
   - ❑ With this information, a code inspection should be able to find values that have been wrongly computed.
   - ❑ Information flow analysis can also show the conditions that affect a variable's value.

5. *Path analysis*

   - ❑ This phase of semantic analysis identifies all possible paths through the program and sets out the statements executed in that path.

- ❑ It essentially unravels the program's control and allows each possible predicate to be analysed individually.

Automated static analysis checks with Fault class are as follow:

1. *Data faults*
   - ❑ Variables used before initialisation
   - ❑ Variables declared but never used
   - ❑ Variables assigned twice but never used between assignments
   - ❑ Possible array bound violations
   - ❑ Undeclared variables
2. *Control faults*
   - ❑ Unreachable code
   - ❑ Unconditional branches into loops
3. *Input/output faults*
   - ❑ Variables output twice with no intervening assignment
4. *Interface faults*
   - ❑ Parameter type mismatches
   - ❑ Parameter number mismatches
   - ❑ Non-usage of the results of functions
   - ❑ Uncalled functions and procedures
5. *Storage management faults*
   - ❑ Unassigned pointers
   - ❑ Pointer arithmetic

14.4 Verification and Formal Methods

- Formal methods of software development are based on mathematical representations of the software, usually as a formal specification.
- These formal methods are mainly concerned with a mathematical analysis of the specification
- with transforming the specification to a more detailed, semantically equivalent representation; or
- with formally verifying that one representation of the system is semantically equivalent to another representation.
- We can use formal methods as the ultimate static verification technique.
- They require very detailed analyses of the system specification and the program, and their use is often time consuming and expensive.
- Consequently, the use of formal methods is mostly confined to safety- and security-critical software development processes.
- The use of formal mathematical specification and associated verification was mandated in UK defence standards for safety-critical software (MOD, 1995).

Formal methods may be used at different stages in the V & V process as shown below:

1. A formal specification of the system may be developed and mathematically analysed for inconsistency. This technique is effective in discovering specification errors and omissions.
2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification and is effective in discovering programming and some design errors. A transformational development process where a formal specification is transformed through a series of more detailed representations or a Cleanroom process may be used to support the formal verification process.
    - Formal verification demonstrates that the developed program meets its specification so implementation errors do not compromise dependability.

- The argument against the use of formal specification is that it requires specialised notations.
- These can only be used by specially trained staff and cannot be understood by domain experts.
- Software engineers cannot recognise potential difficulties with the requirements because they don't understand the domain; domain experts cannot find these problems because they don't understand the specification.
- Although the specification may be mathematically consistent, it may not specify the system properties that are really required.
- Many people think that formal verification is not cost-effective.
- The same level of confidence in the system can be achieved more cheaply by using other validation techniques such as inspections and system testing.
- It is sometimes claimed that the use of formal methods for system development leads to more reliable and safer systems.
- There is no doubt that a formal system specification is less likely to contain anomalies that must be resolved by the system designer.

Formal specification and proof do not guarantee that the software will be reliable in practical use. The reasons for this are:

1. *The specification may not reflect the real requirements of system users.*
   - ❑ Lutz (Lutz, 1993) discovered that many failures experienced by users were a consequence of specification errors and omissions that could not be detected by formal system specification.
   - ❑ System users rarely understand formal notations so they cannot read the formal specification directly to find errors and omissions.
2. *The proof may contain errors.*
   - ❑ Program proofs are large and complex, so, like large and complex programs, they usually contain errors.

3. *The proof may assume a usage pattern which is incorrect.*

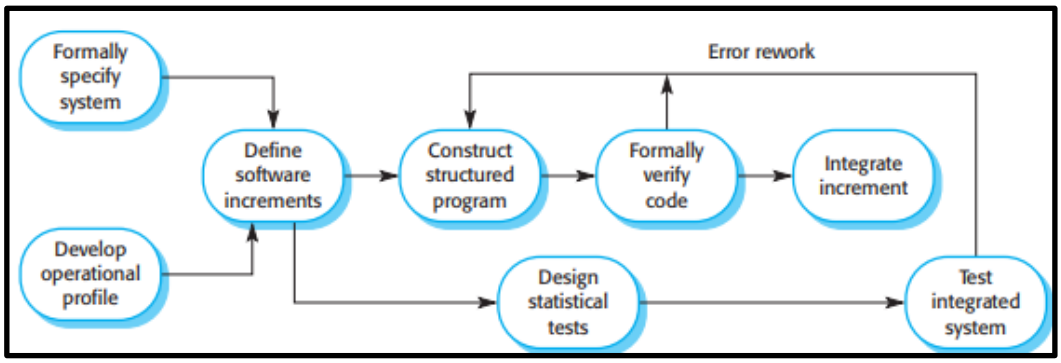
   - ❑ If the system is not used as anticipated, the proof may be invalid.
   - In spite of their disadvantages, formal methods have an important role to play in the development of critical software systems.
   - Formal specifications are very effective in discovering specification problems that are the most common causes of system failure.
   - Formal verification increases confidence in the most critical components of these systems.
   - The use of formal approaches is increasing as procurers demand it and as more and more engineers become familiar with these techniques.

14.5 Cleanroom software development:

- Another well-documented approach that uses formal methods is the Cleanroom development process.
- Cleanroom software development (Mills, et al., 1987; Cobb and Mills, 1990; Linger, 1994; Prowell, et al., 1999) is a software development philosophy that uses formal methods to support rigorous software inspection.
- The **objective** of this approach to software development **is zero-defect software**.
- The name ‘Cleanroom’ was derived by analogy with semiconductor fabrication units where defects are avoided by manufacturing in an ultra-clean atmosphere.
- Cleanroom development is particularly relevant to this chapter because it has replaced the unit testing of system components by inspections to check the consistency of these components with their specifications.

The Cleanroom approach to software development is based on five key strategies:

The Cleanroom development process



1. *Formal specification*
   - ❑ The software to be developed is formally specified.
   - ❑ A state transition model that shows system responses to stimuli is used to express the specification.
2. *Incremental development*
   - ❑ The software is partitioned into increments that are developed and validated separately using the Cleanroom process.
   - ❑ These increments are specified, with customer input, at an early stage in the process.
3. *Structured programming*
   - ❑ Only a limited number of control and data abstraction constructs are used.
   - ❑ The program development process is a process of stepwise refinement of the specification.
   - ❑ A limited number of constructs are used and the aim is to systematically transform the specification to create the program code.
4. *Static verification*
   - ❑ The developed software is statically verified using rigorous software inspections.
   - ❑ There is no unit or module testing process for code components.

5. *Statistical testing of the system*

   - ❑ The integrated software increment is tested statistically to determine its reliability.
   - ❑ These statistical tests are based on an operational profile, which is developed in parallel with the system specification as shown in above Figure.

There are three teams involved when the Cleanroom process is used for large system development:

1. *The specification team*

   - ❑ This group is responsible for developing and maintaining the system specification.
   - ❑ This team produces customer-oriented specifications (the user requirements definition) and mathematical specifications for verification.
   - ❑ In some cases, when the specification is complete, the specification team also takes responsibility for development.

2. *The development team*

   - ❑ This team has the responsibility of developing and verifying the software.
   - ❑ The software is not executed during the development process.
   - ❑ A structured, formal approach to verification based on inspection of code supplemented with correctness arguments is used.

3. *The certification team*

   - ❑ This team is responsible for developing a set of statistical tests to exercise the software after it has been developed.
   - ❑ These tests are based on the formal specification.
   - ❑ Test case development is carried out in parallel with software development.
   - ❑ The test cases are used to certify the software reliability. Reliability growth models used to decide when to stop testing.

14.6 Summary:

- Verification and validation are not the same thing.
- Verification is intended to show that a program meets its specification.
- Validation is intended to show that the program does what the user requires.
- Static verification techniques involve examination and analysis of the program source code to detect errors.
- They should be used with program testing as part of the V & V process.
- Program inspections are effective in finding program errors.
- The aim of an inspection is to locate faults.
- A fault checklist should drive the inspection process.
- In a program inspection, a small team systematically checks the code.
- Team members include a team leader or moderator, the author of the code, a reader who presents the code during the inspection and a tester who considers the code from a testing perspective.
- Static analysers are software tools that process a program source code and draw attention to anomalies such as unused code sections and uninitialised variables.
- These anomalies may be the result of faults in the code.
- Cleanroom software development relies on static techniques for program verification and statistical testing for system reliability certification.
- It has been successful in producing systems that have a high level of reliability.

14.7 Exercise

Answer the following:

1. Explain the differences between verification and validation, and explain why validation is a particularly difficult process.
2. Explain why program inspections are an effective technique for discovering errors in a program.
3. What types of errors are unlikely to be discovered through inspections?
4. Suggest why an organisation with a competitive, elitist culture would probably find it difficult to introduce program inspections as a V & V technique.
5. Explain why it may be cost-effective to use formal methods in the development of safety critical software systems.
6. Why do you think that some developers of this type of system are against the use of formal methods?

❖ ❖ ❖

15

SOFTWARE TESTING - SOFTWARE COST ESTIMATION

Unit Structure:

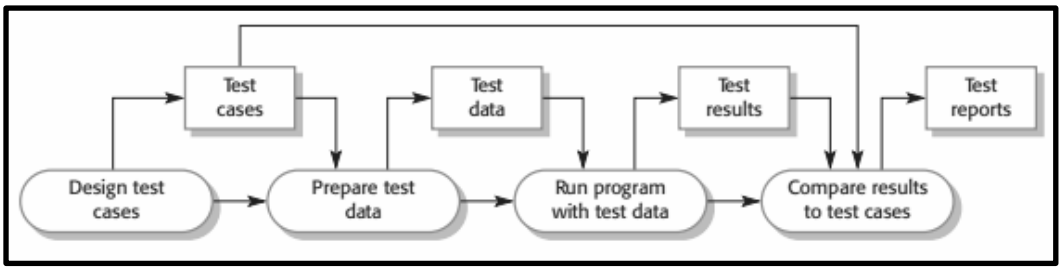
15.0 Objective

- The objective of this chapter is to describe the Processes of Software Testing and Introduce a Range of Testing Techniques.
- Understand the distinctions between Validation Testing and Defect Testing;
- Understand the Principles of System and Component Testing;
- Understand three Strategies that may be used to Generate System Test Cases;

- Understand the Essential Characteristics of Software Tools that Support Test Automation.
- The two fundamental testing activities are
  - component testing—testing the parts of the system and
  - system testing—testing the system as a whole.
- The Software Testing Process Has Two Distinct Goals:
  - To Demonstrate to The Developer and the Customer that the Software Meets Its Requirements
  - To Discover Faults or Defects in the Software where The Behaviour of the Software is incorrect, Undesirable or Does Not Conform to Its Specification.
- The Aim of the Component Testing Stage is to Discover Defects yy Testing Individual Program Components.

15.1 Software Testing:

A model of the software testing process



A general model of the testing process is shown in above Figure.

- Test cases are specifications of the inputs to the test and the expected output from the system plus a statement of what is being tested.
- Test data are the inputs that have been devised to test the system.
- Test data can sometimes be generated automatically. Automatic test case generation is impossible.
- The output of the tests can only be predicted by people who understand what the system should do.
- Exhaustive testing, where every possible program execution sequence is tested, is impossible.
- Testing, therefore, has to be based on a subset of possible test cases.

- Ideally, software companies should have policies for choosing this subset rather than leave this to the development team.
- These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once.
- Alternatively, the testing policies may be based on experience of system usage and may focus on testing the features of the operational system.

For example:

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

15.2 System Testing:

- System testing involves integrating two or more components that implement system functions or features and then testing this integrated system.
- In an iterative development process, system testing is concerned with testing an increment to be delivered to the customer; in a waterfall process, system testing is concerned with testing the entire system.

For most complex systems, there are two distinct phases to system testing:

1. *Integration testing,* where the test team has access to the source code of the system.
    - ❑ When a problem is discovered, the integration team tries to find the source of the problem and identify the components that have to be debugged.
    - ❑ Integration testing is mostly concerned with finding defects in the system.
2. *Release testing,* where a version of the system that could be released to users is tested.
    - ❑ Here, the test team is concerned with validating that the system meets

its requirements and with ensuring that the system is dependable.

- ❑ Release testing is usually 'black-box' testing where the test team is simply concerned with demonstrating that the system does or does not work properly.
- ❑ Problems are reported to the development team whose job is to debug the program.
- ❑ Where customers are involved in release testing, this is sometimes called acceptance testing. If the release is good enough, the customer may then accept it for use.

Integration testing

- The process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions.
- The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components.
- Integration testing checks that these components actually work together, are called correctly and transfer the right data at the right time across their interfaces.
- System integration involves identifying clusters of components that deliver some system functionality and integrating these by adding code that makes them work together.
- In *the top-down integration* skeleton of the system is developed first, and components are added to it.
- Alternatively, in *the bottom-up integration* first integrate infrastructure components that provide common services, such as network and database access, then add the functional components.
- In both top-down and bottom-up integration, usually we have to develop additional code to simulate other components and allow the system to execute.
- **Drawback**:

- A major problem that arises during integration testing is localising errors.
- There are complex interactions between the system components and, when an anomalous output is discovered, you may find it hard to identify where the error occurred.

- To make it easier to locate errors, you should always use an incremental approach to system integration and testing. Initially, you should integrate a minimal system

- **Regression testing:**
  - These problems mean that when a new increment is integrated, it is important to rerun the tests for previous increments as well as the new tests that are required to verify the new system functionality.
  - Rerunning an existing set of tests is called regression testing.
  - If regression testing exposes problems, then you have to check whether these are problems in the previous increment that the new increment has exposed or whether these are due to the added increment of functionality.
  - Regression testing is clearly an expensive process and is impractical without some automated support.

Release testing:

- Release testing is the process of testing a release of the system that will be distributed to customers.
- The primary goal of this process is to increase the supplier's confidence that the system meets its requirements. If so, it can be released as a product or delivered to the customer.
- Release testing is usually a black-box testing process where the tests are derived from the system specification.

Performance testing:

- Once a system has been completely integrated, it is possible to test the system for emergent properties such as performance and reliability.
- Performance tests have to be designed to ensure that the system can process

its intended load.

- This usually involves planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system.
- To test whether performance requirements are being achieved, you may have to construct an operational profile.

15.3 Component Testing :

- Component testing also known as unit testing is the process of testing individual components in the system.
- This is a defect testing process so its goal is to expose faults in these components.
- As I discussed in the introduction, for most systems, the developers of components are responsible for component testing.
- **There are different types of component that may be tested at this stage:**

  1. Individual functions or methods within an object
  2. Object classes that have several attributes and methods
  3. Composite components made up of several different objects or functions.

- These composite components have a defined interface that is used to access their functionality.
- Individual functions or methods are the simplest type of component and your tests are a set of calls to these routines with different input parameters.
- You can use the approaches to test case design, discussed in the next section, to design the function or method tests.
- When you are testing object classes, you should design your tests to provide coverage of all of the features of the object.
- Therefore, object class testing should include:

1. The testing in isolation of all operations associated with the object
2. The setting and interrogation of all attributes associated with the object
3. The exercise of the object in all possible states.

- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.
- Examples of state sequences that should be tested in the weather station include:
- Shutdown → Waiting → Shutdown

  Waiting → Calibrating → Testing → Transmitting → Waiting

  Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

15.4 Test Case Design :

- Test case design is a part of system and component testing where you design the test cases (inputs and predicted outputs) that test the system.
- The goal of the test case design process is to create a set of test cases that are effective in discovering program defects and showing that the system meets its requirements.
- To design a test case, you select a feature of the system or component that you are testing.
- You then select a set of inputs that execute that feature, document the expected outputs or output ranges and, where possible, design an automated check that tests that the actual and expected outputs are the same.
- There are various approaches that you can take to test case design:

1. Requirements-based testing where test cases are designed to test the system requirements. This is mostly used at the system-testing stage as system requirements are usually implemented by several components. For each requirement, you identify test cases that can demonstrate that the system meets that requirement.
2. Partition testing where you identify input and output partitions and design

tests so that the system executes inputs from all partitions and generates outputs in all partitions. Partitions are groups of data that have common characteristics such as all negative numbers, all names less than 30 characters, all events arising from choosing items on a menu, and so on.

3. Structural testing where you use knowledge of the program's structure to design tests that exercise all parts of the program. Essentially, when testing a program, you should try to execute each statement at least once. Structural testing helps identify test cases that can make this possible.

    - In general, when designing test cases, you should start with the highest-level tests from the requirements then progressively add more detailed tests using partition and structural testing.

15.4.1 Requirements-based testing:

A general principle of requirements engineering is that requirements should be testable.

That is, the requirement should be written in such a way that a test can be designed so that an observer can check that the requirement has been satisfied.

Requirements-based testing, therefore, is a systematic approach to test case design where you consider each requirement and derive a set of tests for it.

Requirements-based testing is validation rather than defect testing demonstrates that the system has properly implemented its requirements.

Process of Requirements based Testing:

- **Defining Test Completion Criteria** -

  Testing is completed only when all the functional and non-functional testing is complete.

- **Design Test Cases** -

  A Test case has five parameters namely the initial state or precondition, data setup, the inputs, expected outcomes and actual outcomes.

- **Execute Tests** -

  Execute the test cases against the system under test and document the results.

- **Verify Test Results** -

Verify if the expected and actual results match each other.

- **Verify Test Coverage** -

  Verify if the tests cover both functional and non-functional aspects of the requirement.

- **Track and Manage Defects** -

  Any defects detected during the testing process goes through the defect life cycle and are tracked to resolution. Defect Statistics are maintained which will give us the overall status of the project.

Requirements Testing process:

- Testing must be carried out in a timely manner.
- Testing process should add value to the software life cycle, hence it needs to be effective.
- Testing the system exhaustively is impossible hence the testing process needs to be efficient as well.
- Testing must provide the overall status of the project, hence it should be manageable.

  ➔ The Requirements based testing process starts at the very early phase of the software development, as correcting issues/errors is easier at this phase.

  ➔ It begins at the requirements phase as the chances of occurrence of bugs have its roots here.

  ➔ It aims at quality improvement of requirements. Insufficient requirements leads to failed projects.

15.4.2 Partition testing:

One systematic approach to test case design is based on identifying all partitions for a system or component.

Test cases are designed so that the inputs or outputs lie within these partitions.

Partition testing can be used to design test cases for both systems and components.

Input equivalence partitions are sets of data where all of the set members should be

processed in an equivalent way.

Output equivalence partitions are program outputs that have common characteristics, so they can be considered as a distinct class.

You also identify partitions where the inputs are outside the other partitions that you have chosen.

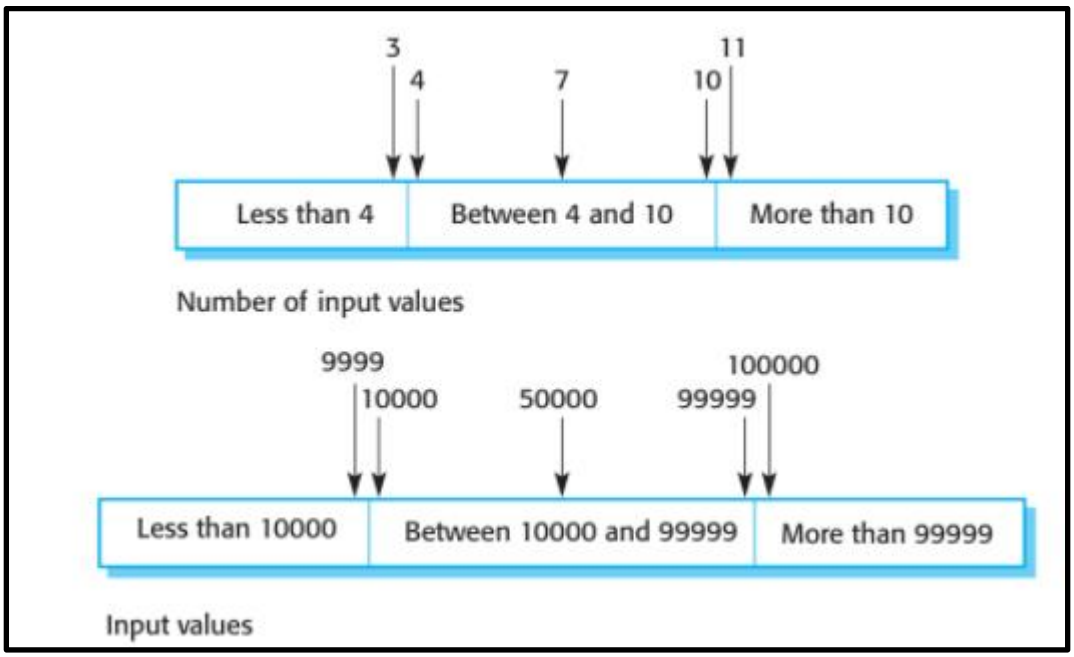
These test whether the program handles invalid input correctly.

Valid and invalid inputs also form equivalence partitions.

You identify partitions by using the program specification or user documentation and, from experience, where you predict the classes of input value that are likely to detect errors.

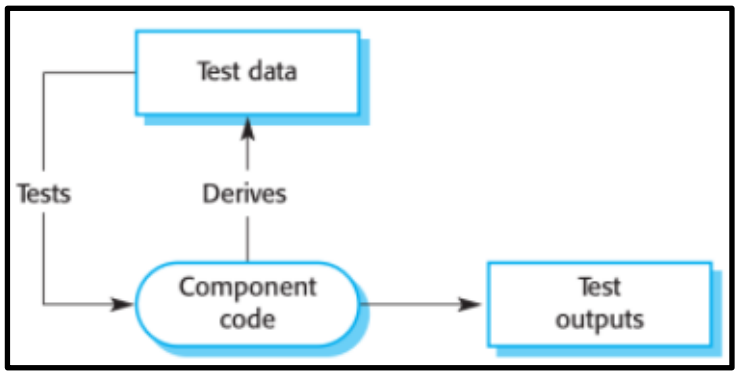
For example, say a program specification states that the program accepts 4 to 8 inputs that are five-digit integers greater than 10,000.

Equivalence partitions



15.4.3 Structural testing:

- Structural testing is an approach to test case design where the tests are derived from knowledge of the software's structure and implementation.
- This approach is sometimes called 'white-box', 'glass-box' testing, or ''clear-box' test-

- ing to distinguish it from black-box testing.
- Understanding the algorithm used in a component can help you identify further partitions and test cases.
- 

- Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.
- Structural Testing Techniques:
    - Statement Coverage - This technique is aimed at exercising all programming statements with minimal tests.
    - Branch Coverage - This technique is running a series of tests to ensure that all branches are tested at least once.
    - Path Coverage - This technique corresponds to testing all possible paths which means that each statement and branch are covered.

15.4.4 Path testing:

- Path Testing is a structural testing method based on the source code or algorithm and NOT based on the specifications.
- It can be applied at different levels of granularity.
- Path Testing Assumptions:
    - The Specifications are Accurate
    - The Data is defined and accessed properly
    - There are no defects that exist in the system other than those that affect control flow

- Path Testing Techniques:
    - Control Flow Graph (CFG) - The Program is converted into Flow graphs by representing the code into nodes, regions and edges.
    - Decision to Decision path (D-D) - The CFG can be broken into various Decision to Decision paths and then collapsed into individual nodes.
    - Independent (basis) paths - Independent path is a path through a DD-path graph which cannot be reproduced from other paths by other methods.

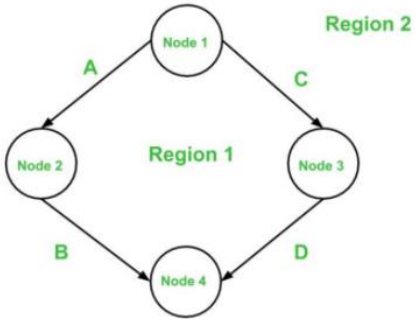
1. Control Flow Graph –

A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module. A control flow graph (V, E) has V number of nodes/vertices and E number of edges in it. A control graph can also have :

Junction Node – a node with more than one arrow entering it.

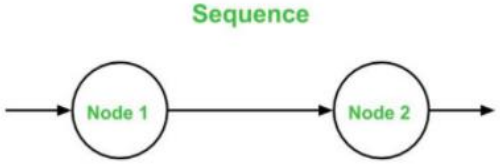
Decision Node – a node with more than one arrow leaving it.

Region – area bounded by edges and nodes (area outside the graph is also counted as a region.).



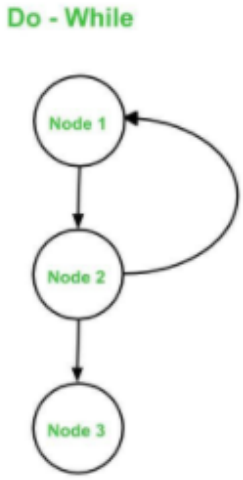
Below are the **notations** used while constructing a flow graph :

- **Sequential Statements –**

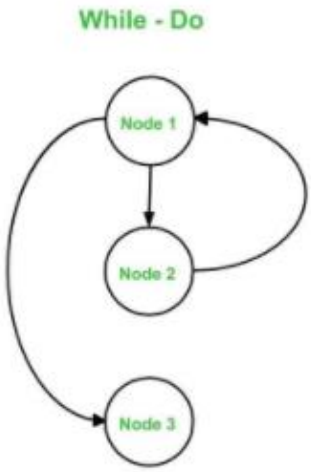


- **If – Then – Else –**

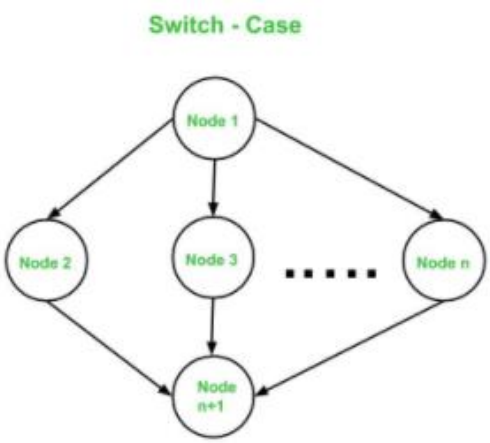
- **Do – While –**



- **While – Do –**



- **Switch – Case –**



Cyclomatic Complexity –

The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

Formula based on edges and nodes :
V(G) = e - n + 2*P
Where,
e is number of edges,
n is number of vertices,
P is number of connected components.
For example, consider first graph given above,
where, e = 4, n = 4 and p = 1

So,

Cyclomatic complexity V(G)

= 4 - 4 + 2 * 1

1. = 2

   Formula based on Decision Nodes :
   V(G) = d + P
   where,
   d is number of decision nodes,
   P is number of connected nodes.
   For example, consider first graph given above,
   where, d = 1 and p = 1

   So,

Cyclomatic Complexity V(G)

= 1 + 1

2. = 2

Formula based on Regions :
V(G) = number of regions in the graph
For example, consider first graph given above,
Cyclomatic complexity V(G)

= 1 (for Region 1) + 1 (for Region 2)

3. = 2

Hence, using all the three above formulae, the cyclomatic complexity obtained remains same. All these three formulae can be used to compute and verify the cyclomatic complexity of the flow graph.

Note –

1. For one function [e.g. Main( ) or Factorial( )], only one flow graph is constructed. If in a program, there are multiple functions, then a separate flow graph is constructed for each one of them. Also, in the cyclomatic complexity formula, the value of 'p' is set depending of the number of graphs present in total.

2. If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula :
d = k - 1
Here, k is number of arrows leaving the decision node.

Independent Paths :

An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined. The cyclomatic complexity gives the number of independent paths present in a flow graph. This is because the cyclomatic complexity is used as an upper-bound for the number of tests that should be executed in order to make sure that all the statements in the program have been executed at least once.

Consider first graph given above here the independent paths would be 2 because number of independent paths is equal to the cyclomatic complexity.

So, the independent paths in above first given graph :

- Path 1:

A -> B

- Path 2:

C -> D

Note –

Independent paths are not unique. In other words, if for a graph the cyclomatic complexity comes out be N, then there is a possibility of obtaining two different sets of paths which are independent in nature.

Design Test Cases :

Finally, after obtaining the independent paths, test cases can be designed where each test case represents one or more independent paths.

Advantages :

Basis Path Testing can be applicable in the following cases:

1. More Coverage –
   Basis path testing provides the best code coverage as it aims to achieve maximum logic coverage instead of maximum path coverage. This results in an overall thorough testing of the code.

2. Maintenance Testing –
   When a software is modified, it is still necessary to test the changes made in the software which as a result, requires path testing.

3. Unit Testing –
   When a developer writes the code, he or she tests the structure of the program or module themselves first. This is why basis path testing requires enough knowledge about the structure of the code.

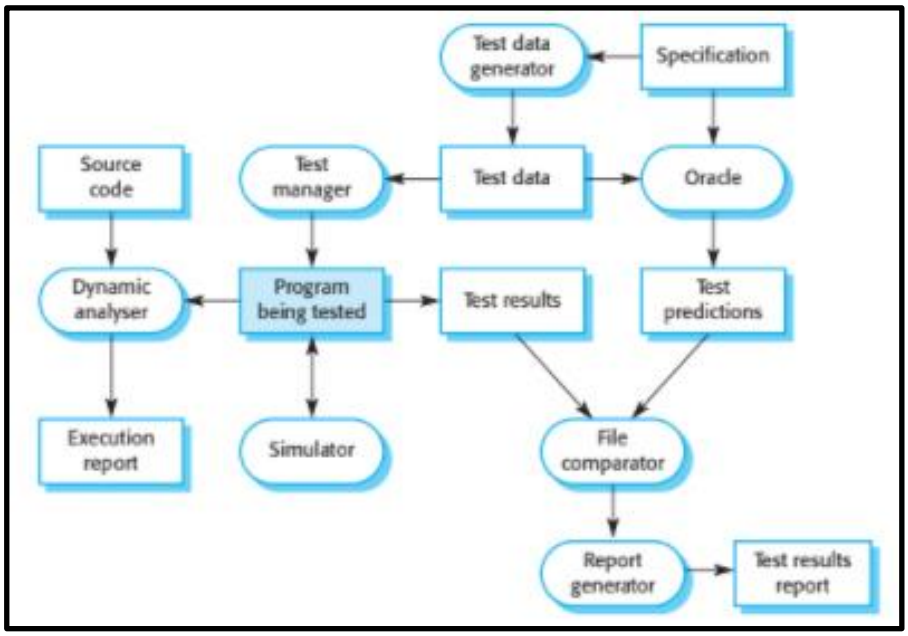
4. Integration Testing –
   When one module calls other modules, there are high chances of Interface errors. In order to avoid the case of such errors, path testing is performed to test all the paths on the interfaces of the modules.

5. Testing Effort –
   Since the basis path testing technique takes into account the complexity of

the software (i.e., program or module) while computing the cyclomatic complexity, therefore it is intuitive to note that testing effort in case of basis path testing is directly proportional to the complexity of the software or program.

15.5 Test Automation :

- Testing is an expensive and laborious phase of the software process.
- As a result,testing tools were among the first software tools to be developed.
- These tools offer a range of facilities and their use can significantly reduce the costs of testing.
- The tests themselves should be written in such a way that they indicate whether the tested system has behaved as expected.
- A software testing workbench is an integrated set of tools to support the testing process.
- In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.
- Following Figure shows some of the tools that might be included in such a testing workbench:



1. **Test manager** Manages the running of program tests. The test manager keeps

track of test data, expected results and program facilities tested. Test automation frameworks such as JUnit are examples of test managers.

2. **Test data** generator Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.

3. **Oracle** Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems. Back-to-back testing involves running the oracle and the program to be tested in parallel. Differences in their outputs are highlighted.

4. **File comparator** Compares the results of program tests with previous test results and reports differences between them. Comparators are used in regression testing where the results of executing different versions are compared. Where automated tests are used, this may be called from within the tests themselves.

5. **Report** generator Provides report definition and generation facilities for test results.

6. Dynamic analyser Adds code to a program to count the number of times each statement has been executed. After testing, an execution profile is generated showing how often each program statement has been executed.

7. Simulator Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute.

User interface simulators are script-driven programs that simulate multiple simultaneous user

interactions. Using simulators for I/O means that the timing of transaction sequences is repeatable.

15.6 Software Cost Estimation :

It is always necessary to know how much any new project will cost to develop and how much development time will it take.

These estimates are needed before development is initiated.

The objective of this section is to introduce techniques for estimating the cost and effort required for software production.

- understand the fundamentals of software costing and reasons why the price of the software may not be directly related to its development cost;
- have been introduced to three metrics that are used for software productivity assessment;
- appreciate why a range of techniques should be used when estimating software costs and schedule;
- understand the principles of the COCOMO II model for algorithmic cost estimation.

Several estimation procedures have been developed and are having the following attributes in common.

1. Project scope must be established in advanced.
2. Software metrics are used as a support from which evaluation is made.
3. The project is broken into small PCs which are estimated individually. To achieve true cost & schedule estimate, several option arise.
4. Delay estimation
5. Used symbol decomposition techniques to generate project cost and schedule estimates.
6. Acquire one or more automated estimation tools.

15.6.1 Software productivity:

Factors affecting software pricing are as follow:

- **Market opportunity**
    - A development organisation may quote a low price because it wishes to move into a new segment of the software market.
    - Accepting a low profit on one project may give the organisation the opportunity to make a greater profit later.
    - The experience gained may also help it develop new products.

- **Cost estimate uncertainty**
    - If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
- **Contractual terms**
    - A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects.
    - The price charged may then be less than if the software source code is handed over to the customer.
- **Requirements volatility**
    - If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
- **Financial health**
    - Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.
- Productivity estimates are usually based on measuring attributes of the software and dividing this by the total effort required for development.
- There are two types of metric that have been used:

1. *Size-related metrics*

➔ These are related to the size of some output from an activity.

➔ The most commonly used size-related metric is lines of delivered source code.

➔ Other metrics that may be used are the number of delivered object code instructions or the number of pages of system documentation.

2. *Function-related metrics*

➔ These are related to the overall functionality of the delivered software. Productivity is expressed in terms of the amount of useful functionality produced in some given time.

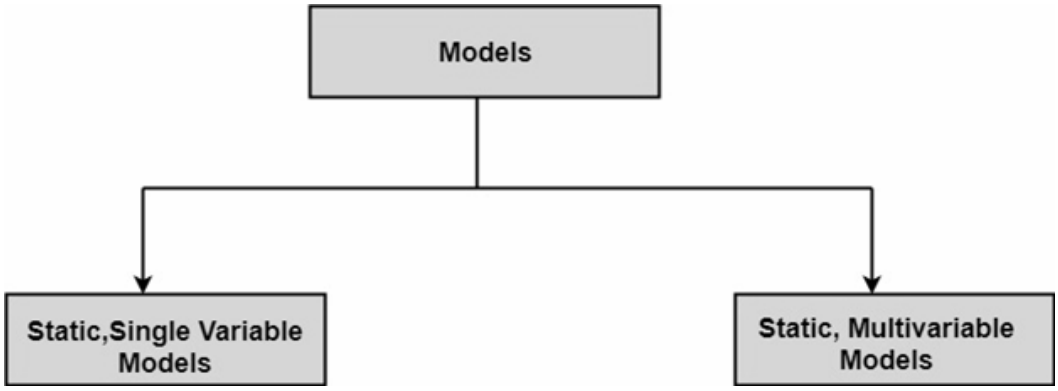
➔ Function points and object points are the best-known metrics of this type.

Cost Estimation Models:

A model may be static or dynamic.

In a static model, a single variable is taken as a key element for calculating cost and time.

In a dynamic model, all variable are interdependent, and there is no basic variable.



Static, Single Variable Models:

When a model makes use of single variables to calculate desired values such as cost, time, efforts, etc. is said to be a single variable model. The most common equation is:

$C=aL^b$

Where C = Costs

L= size

a and b are constants

The Software Engineering Laboratory established a model called SEL model, for estimating its software production. This model is an example of the static, single variable model.

$E=1.4L^{0.93}$

$DOC=30.4L^{0.90}$

$D=4.6L^{0.26}$

Where E= Efforts (Person Per Month)

DOC=Documentation (Number of Pages)

D = Duration (D, in months)

L = Number of Lines per code

Static, Multivariable Models:

These models are based on method (1), they depend on several variables describing various aspects of the software development environment.

In some models, several variables are needed to describe the software development process, and the selected equation combines these variables to give the estimate of time & cost.

These models are called multivariable models.

WALSTON and FELIX develop the models at IBM provide the following equation gives a relationship between lines of source code and effort:

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

In the same manner duration of development is given by

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

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

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

Where $\mathbf{W_i}$ is the weight factor for the $\mathbf{i^{th}}$variable and $\mathbf{X_i}$**={-1,0,+1}** the estimator gives $\mathbf{X_i}$one of the values **-1, 0 or +1** depending on the variable decreases, has no effect or increases the productivity.

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

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

b. Calculate the duration of the development.

c. Calculate the productivity in LOC/PY

d. Calculate the average manning

Solution:

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

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

$$L = \left(\frac{E}{a}\right) 1/b$$

Then

L (SEL) = (96/1.4)1⁄0.93=94264 LOC

L (SEL) = (96/5.2)1⁄0.91=24632 LOC

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

D (SEL) = 4.6 (L) 0.26

= 4.6 (94.264)0.26 = 15 months

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

= 4.1 (24.632)0.36 = 13 months

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

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

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

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

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

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

COCOMO Model :

- Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.
- COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.
- **The necessary steps in this model are:**
    - Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
    - Determine a set of 15 multiplying factors from various attributes of the project.
    - Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.
- The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size.
- To determine the initial effort E_i in person-months the equation used is of the type is shown below
- Ei=a*(KDLOC)b , where The value of the constant a and b are depends on the project type.

In COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

1. Organic:

- A development project can be treated of the organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar methods of projects.
- Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.

2. Semidetached:

- A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff.
- Team members may have finite experience in related systems but may be unfamiliar with some aspects of the order being developed.
- Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.

3. Embedded:

- A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist.
- For Example: ATM, Air Traffic control.

  For three product categories, Bohem provides a different set of expression to predict effort (in a unit of person month)and development time from the size of estimation in KLOC(Kilo Line of code) efforts estimation takes into account the productivity loss due to holidays, weekly off, coffee breaks, etc.

According to Boehm, software cost estimation should be done through three stages:

1. Basic Model
2. Intermediate Model
3. Detailed Model

1. Basic COCOMO Model:

The basic COCOMO model provide an accurate size of the project parameters.

The following expressions give the basic COCOMO estimation model:

Effort=a1*(KLOC) a2 PM

Tdev=b1*(efforts)b2 Months

Where

KLOC is the estimated size of the software product indicate in Kilo Lines of Code,

a1,a2,b1,b2 are constants for each group of software products,

Tdev is the estimated time to develop the software, expressed in months,

Effort is the total effort required to develop the software product, expressed in person months (PMs).

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: Effort = 2.4(KLOC) 1.05 PM

Semi-detached: Effort = 3.0(KLOC) 1.12 PM

Embedded: Effort = 3.6(KLOC) 1.20 PM

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic: Tdev = 2.5(Effort) 0.38 Months

Semi-detached: Tdev = 2.5(Effort) 0.35 Months

Embedded: Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes.

Following Fig shows a plot of estimated effort versus product size.

From fig, we can observe that the effort is somewhat superliner in the size of the software product.

Thus, the effort required to develop a product increases very rapidly with project size.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

Effort=a_1*(KLOC) a_2 PM

Tdev=b_1*(efforts)b_2 Months

Estimated Size of project= 400 KLOC

(i) Organic Mode

E = 2.4 * (400)1.05 = 1295.31 PM

D = 2.5 * (1295.31)0.38=38.07 PM

(ii) Semidetached Mode

E = 3.0 * (400)1.12=2462.79 PM

D = 2.5 * (2462.79)0.35=38.45 PM

(iii) Embedded Mode

E = 3.6 * (400)1.20 = 4772.81 PM

D = 2.5 * (4772.8)0.32 = 38 PM

Example2: A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: The semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time.

Hence E=3.0(200)1.12=1133.12PM

D=2.5(1133.12)0.35=29.3PM

$$\text{Average Staff Size (SS)} = \frac{E}{D} \text{ Persons}$$

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

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

P = 176 LOC/PM

2. Intermediate Model:

The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems.

The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

- Required software reliability extent
- Size of the application database
- The complexity of the product

Hardware attributes -

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

Personnel attributes -

- Analyst capability
- Software engineering capability
- Applications experience

- Programming language experience

Project attributes -

- Use of software tools
- Application of software engineering methods
- Required development schedule

Intermediate COCOMO equation:

$$E=a_i (KLOC) b_i*EAF$$

$$D=c_i (E)d_i$$

3. Detailed COCOMO Model:

Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost drivers' effect on each method of the software engineering process.

The detailed model uses various effort multipliers for each cost driver property.

In detailed cocomo, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

1. Planning and requirements
2. System structure
3. Complete structure
4. Module code and test
5. Integration and test
6. Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers

are given according to every phase of the software lifecycle.

15.6.2 Estimation techniques:

Organisations need to make software effort and cost estimates.

To do so, one or more of the techniques described below may be used.

All of these techniques rely on experience-based judgements by project managers who use their knowledge of previous projects to arrive at an estimate of the resources required for the project.

Some examples of the changes that may affect estimates based on experience include:

1. Distributed object systems rather than mainframe-based systems
2. Use of web services
3. Use of ERP or database-centred systems
4. Use of off-the-shelf software rather than original system development
5. Development for and with reuse rather than new development of all parts of a system
6. Development using scripting languages such as TCL or Perl (Ousterhout, 1998)
7. The use of CASE tools and program generators rather than unsupported software development.

Various Cost estimation techniques are as follows:

- **Algorithmic cost modelling**
    - A model is developed using historical cost information that relates some software metric (usually its size) to the project cost.
    - An estimate is made of that metric and the model predicts the effort required.
- **Expert judgement**
    - Several experts on the proposed software development techniques and the application domain are consulted.

- These estimates are compared and discussed.
- The estimation process iterates until an agreed estimate is reached.

- **Estimation by analogy**
  - This technique is applicable when other projects in the same application domain have been completed.
  - The cost of a new project is estimated by analogy with these completed projects.
  - Myers (Myers, 1989) gives a very clear description of this approach.
- **Parkinson's Law**
  - Parkinson's Law states that work expands to fill the time available.
  - The cost is determined by available resources rather than by objective assessment.
  - If the software has to be delivered in 12 months and 5 people are available, the effort required is estimated to be 60 person-months.
- **Pricing to win**
  - The software cost is estimated to be whatever the customer has available to spend on the project.
  - The estimated effort depends on the customer's budget and not on the software functionality.

15.6.3 Algorithmic cost modelling:

- Algorithmic cost modelling uses a mathematical expression to predict project costs based on estimates of the project size, the number of software engineers, and other process and product factors.
- An algorithmic cost model can be developed by analyzing the costs and attributes of completed projects and finding the closest fit mathematical expression to the actual project.
- In general, an algorithmic cost estimate for software cost can be expressed as:

$$EFFORT = A \times SIZE^{B} \times M$$

- In this equation A is a constant factor that depends on local organizational practices and the type of software that is developed.
- Variable SIZE may be either the code size or the functionality of software expressed in function or object points.
- M is a multiplier made by combining process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.
- The exponential component B associated with the size estimate expresses the non-linearity of costs with project size.
- As the size of the software increases, extra costs are emerged.
- The value of exponent B usually lies between 1 and 1.5.
- All algorithmic models have the same difficulties:
    - It is difficult to estimate SIZE in the early stage of development. Function or object point estimates can be produced easier than estimates of code size but are often inaccurate.
    - The estimates of the factors contributing to B and M are subjective. Estimates vary from one person to another person, depending on their background and experience with the type of system that is being developed.
- The number of lines of source code in software is the basic software metric used in many algorithmic cost models.
- The code size can be estimated by previous projects, by converting function or object points to code size, by using a reference component to estimate the component size, etc.
- The programming language used for system development also affects the number of lines of code to be implemented.
- Furthermore, it may be possible to reuse codes from previous projects and the size estimate has to be adjusted to take this into account.

15.6.4 Project Duration and Staffing:

- As well as estimating the effort required to develop a software system and the overall project costs, project managers must also estimate how long the software will take to develop and when staff will be needed to work on the project.
- The development time for the project is called the project schedule.
- Increasingly, organisations are demanding shorter development schedules so that their products can be brought to market before their competitor's.
- The relationship between the number of staff working on a project, the total effort required and the development time is not linear.
- As the number of staff increases, more effort may be needed. The reason for this is that people spend more time communicating and defining interfaces between the parts of the system developed by other people.
- Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved.
- The COCOMO model includes a formula to estimate the calendar time (TDEV) required to complete a project. The time computation formula is the same for all COCOMO levels:
- TDEV = 3 (PM)(0.33+0.2*(B-1.01))
- PM is the effort computation and B is the exponent computed, as discussed above (B is 1 for the early prototyping model).
- This computation predicts the nominal schedule for the project.
- However, the predicted project schedule and the schedule required by the project plan are not necessarily the same thing.
- The planned schedule may be shorter or longer than the nominal predicted schedule.
- However, there is obviously a limit to the extent of schedule changes, and the COCOMO II model predicts this:
- TDEV = 3 (PM)(0.33+0.2*(B-1.01)) SCEDPercentage/100
- SCEDPercentage is the percentage increase or decrease in the nominal schedule.
- If the predicted figure then differs significantly from the planned schedule, it suggests that there is a high risk of problems delivering the software as planned.

- To illustrate the COCOMO development schedule computation, assume that 60 months of effort are estimated to develop a software system (Option C in Figure Assume that the value of exponent B is 1.17. From the schedule equation, the time required to complete the project is:
- TDEV = 3 (60)0.36 = 13 months
- In this case, there is no schedule compression or expansion, so the last term in the formula has no effect on the computation.
- An interesting implication of the COCOMO model is that the time required to complete the project is a function of the total effort required for the project.
- It does not depend on the number of software engineers working on the project.

- This confirms the notion that adding more people to a project that is behind schedule is unlikely to help that schedule to be regained. Myers (Myers, 1989) discusses the problems of schedule acceleration.
- He suggests that projects are likely to run into significant problems if they try to develop software without allowing sufficient calendar time.

15.7 Summary:

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Component testing is the responsibility of the component developer. A separate testing team usually carries out system testing.
- Integration testing is the initial system testing activity where you test integrated components for defects. Release testing is concerned with testing customer releases and should validate that the system to be released meets its requirements.
- When testing systems, you should try to ‘break’ the system by using experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.
- Interface testing is intended to discover defects in the interfaces of composite components. Interface defects may arise because of errors made in reading the specification, specification misunderstandings or errors or invalid timing assumptions.

- Equivalence partitioning is a way of deriving test cases. It depends on finding partitions in the input and output data sets and exercising the program with values from these partitions. Often, the value that is most likely to lead to a successful test is a value at the boundary of a partition.
- Structural testing relies on analysing a program to determine paths through it and using this analysis to assist with the selection of test cases.
- Test automation reduces the costs of testing by supporting the testing process with a range of software tools.
- There is not necessarily a simple relationship between the price charged for a system and its development costs. Organisational factors may mean that the price charged is increased to compensate for increased risk or decreased to gain competitive advantage.
- Factors that affect software productivity include individual aptitude (the dominant factor), domain experience, the development process, the size of the project, tool support and the working environment.
- Software is often priced to gain a contract, and the functionality of the system is then adjusted to meet the estimated price.
- There are various techniques of software cost estimation. In preparing an estimate, several different techniques should be used. If the estimates diverge widely, this means that inadequate estimating information is available.
- The COCOMO II costing model is a well-developed algorithmic cost model that takes project, product, hardware and personnel attributes into account when formulating a cost estimate. It also includes a means of estimating development schedules.
- Algorithmic cost models can be used to support quantitative option analysis. They allow the cost of various options to be computed and, even with errors, the options can be compared on an objective basis.
- The time required to complete a project is not simply proportional to the number of people working on the project. Adding more people to a late project can increase rather than decrease the time required to finish the project.

15.8 Exercise:

Answer the following:

1. What is the objective of software cost estimation?

16

QUALITY MANAGEMENT

Unit Structure:

Objective:

Understand the software quality and measurement of the product/software.

Software planning and process is important. Software.

Software planning can decrease the quality of the product.

Understand how measurement is help full.

16.1 Quality Management

The quality is the capacity of commodity to satisfy human wants and needs better the quality higher the cost half the product which means as you satisfy all the requirement of the customer which are specified by individual customer and if those requirements are specifications providing output according to the

requirement and giving us high quality output with maximum consistency then we can called any product as a quality product

Software quality management for software systems has three principal concerns:

1. **The organizational level**:

   Quality management is concerned with establishing a framework of organizational processes and standards that will lead to high quality software. This means that the quality management team should take responsibility for defining the software development processes to be used and standards that should apply to the software and related documentation, including the system requirements, design, and code.

2. **The project level**:

   Quality management involves the application of specific quality processes, checking that these planned processes have been followed, and ensuring that the project outputs are conformant with the standards that are applicable to that project.

3. **Quality management**:

   at the project level is also concerned with establishing a quality plan for a project. The quality plan should set out the quality goals for the project and define what processes and standards are to be used.

16.1.1 There are several quality factors which define a quality product as follows

Portability:

Software must be platform independent. It must be easily installed in any of the operating system or it can be installed in any system environment, in any machine with other software. It can be stored in any of the device (CD, Pen-drive) while moving the software from one point to another point for installation

Usability:

Every product has good usability, but while using the product the working of the product needs to be understand by the end user or by the owner of that particular product. Understanding the product process is most important part in usability so that any expert or novice user can easily understand and use the product.

Re usability:

The software product must provide re-usability, as it can be used hey in other modules while developing a new module, such of software module hey need to be used all developed as it will save more time money on implementing different projects or modules.

Correct Ness:

The requirements which are provided by the customer must be implemented in the given project if all the requirements which are specified and working properly and giving quality output and consistency then we can measure the correct Ness of the product.

Maintainability:

Maintainability is nothing but we can do the changes into the system as per our requirement, whenever we want but it could not affect any part of the designing.

If any error occurs, then we can able to modify or remove error from the software. As per requirement we can add new functionalities in the system as well.

Software quality management provide independent check on software development process, for small system development, software quality management hey, it is done with the help of a proper communication between team members which include external customer as well who provide views on the project which is developed or in developing mode.

For large system we will do software quality management with respect to following points

Quality assurance:

Any product needs to assure the customer with its quality output, as it aims in building the organizational procedures and standards. Procedure/Output assure the quality of the product the market branding for the product as well As for organization may get increased.

Quality planning:

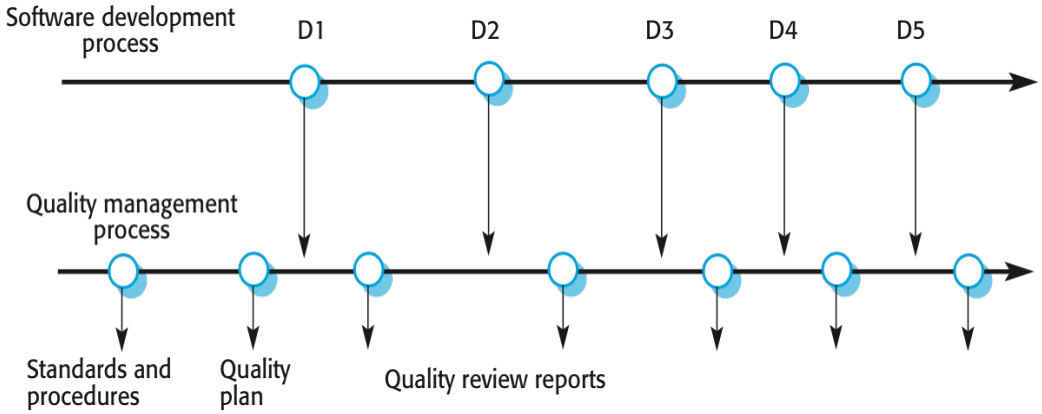
Better the plain best the product hey so that you have to select best planning procedure to build any product selecting appropriate procedure and proper planning increase the quality output of the product.

Quality control:

Quality of the product hey need to be controlled as it need to be consistent throughout the procedure. So, we need to implement a product or module by following quality procedures and standards.

16.1.2 Quality Management and Software development

- Quality management provides an independent check on the software development process. The quality management process checks the project deliverables to ensure that they are consistent with organizational standards and goals.
- The QA team should be independent from the development team so that they can take an objective view of the software.
- This allows them to report on software quality without being influenced by software development issues.



1. From the above diagram we can see that quality management provides an independent check on the software development process.
2. The quality management process checks the project deliverable to ensure that they are consistent with organizational standards and goals. This allows them to report on software quality without Bing influence by software development issue.
3. software development team is independent from quality management team.
    - Quality planning is the process of developing a quality plan for a project.

- The quality plan should set out the desired software qualities and describe how these are to be assessed.
- It therefore defines what 'high-quality software actually means for a particular system Without this definition, engineers may make different sometimes conflicting assumptions about which product attributes reflect the most important quality characteristics.
- Formalized quality planning is an integral part of plan-based development processes.
- Agile methods, however, adopt a less formal approach to quality management.

An out-line structure for a quality plan. This includes:

1. Product introduction A description of the product, its intended market, and the quality expectations for the product.
2. Product plans the critical release dates and responsibilities for the product. along with plans for distribution and product servicing.
3. Process descriptions the development and service processes and standards that should be used for product development and management.
4. Quality goals the quality goals and plans for the product, including an identification and justification of critical product quality attributes.
5. Risks and risk management the key risks that might affect product quality and the actions to be taken to address these risks.

16.2 Software Quality

The fundamentals of quality management were established by manufacturing industry in a drive to improve the quality of the products that were being made.

Software quality is not directly comparable with quality in manufacturing. The idea of tolerances is not applicable to digital systems and, for the following reasons, it may be impossible to come to an objective conclusion about whether or not a software system meets its specification.

1. Difficult to write the complete an unambiguous software specifications. hey Software developer and customer may interpret the requirements in different way.

2. specification usually integrate requirement from several classes of stakeholders. These requirements are inevitably a compromise and may not be included the requirement of all stakeholders group.

3. It is impossible to measure certain quality characteristics directly.

16.2.1 Software Quality attribute

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

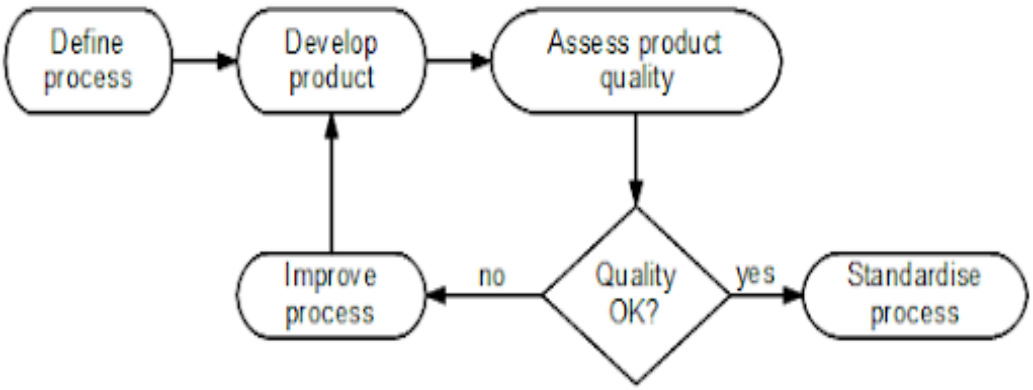


Fig: Process-based quality

- There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor.
- Once manufacturing systems are calibrated, they can be run again and again to output high-quality products.
- However, software is not manufactured-it is designed. In software development, therefore, the relationship between process quality and product quality is more complex.

- Software development is a creative rather than a mechanical process, so the influence of individual skills and experience is significant.
- External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.
- The development process used has a significant influence on the quality of the software and those good processes are more likely to lead to good quality software.
- Process quality management and improvement can lead to fewer defects in the software being developed. However, it is difficult to assess software quality attributes, such as maintainability, without using the software for a long period.
- Consequently, it is hard to tell how process characteristics influence these attributes.

16.3 Quality Assurance and software Standards

Quality assurance is a planned and systematic approach necessary to provide a high degree of confidence in the quality of a product. It provides quality assessment of the quality control activities and determines the validity of the data for identifying the quality.

The main function of quality assurance is to define the standards as:

1. Product standards

2. Process standards

1. Product standards

These apply to the software product being developed. They include document standards, such as the structure of requirements documents, documentation standards, such as a standard comment header for an object class definition, and coding standards, which define how a programming language should be used.

2. Process standards

These define the processes that should be followed during software development. They should encapsulate good development practice. Process standards may include definitions of specification, design and validation processes, process support tools, and a description of the documents that should be written during these processes.

Software standards are important for three reasons:

1. Standards capture wisdom that is of value to the organization. They are based on knowledge about the best or most appropriate practice for the company. Building it into a standard helps the company reuse this experience and avoid previous mistakes.
2. Standards provide a framework for defining what 'quality' means in a particular setting, software quality is subjective, and by using standards you establish a basis for deciding if a required level of quality has been achieved. Of course, this depends on setting standards that reflect user expectations for software dependability, usability, and performance.
3. Standards assist continuity when work carried out by one person is taken up and continued by another. Standards ensure that all engineers within an organization adopt the same practices. Consequently, the learning effort required when starting new work is reduced.

As there are two type pf standards there are many Product and Process standards which are depicted in table below.

Product standards	**Process standards**
Design review conduct	Design review form
Requirements document structure	Submission of new code for system building
Method header format	Version release process
Java programming style	Project plan approval process
Project plan format	Change control process
Change request form	Test recording process

To minimize dissatisfaction and to encourage buy-in to standards, quality managers who set the standards should therefore take the following steps:

1. Involve software engineers in the selection of product standards

If developers understand why standards have been selected, they are more likely to be com mitted to these standards. Ideally, the standards document should not just set out the standard to be followed but should also include commentary explaining why standardization decisions have been made.

2. Review and modify standards regularly to reflect changing technologies

Standards are expensive to develop and they tend to be enshrined in a company standards handbook. Because of the costs and discussion required, there is often a reluctance to change them. A standards handbook is essential, but it should evolve to reflect changing circumstances and technology.

3. Provide software tools to support standards

Developers often find standards to be a bugbear when conformance to them involves tedious manual work that could be done by a software tool. If tool support is available, very little effort is required to follow the software development standards. For example, document standards can be implemented using word processor styles.

16.3.1 The ISO 9001 standards framework

- There is an international set of standards that can be used in the development of quality management systems in all industries, called ISO 9000.
- ISO 9000 standards can be applied to a range of organizations from manufacturing through to service industries.
- ISO 9001, the most general of these standards, applies to organizations that design, develop, and maintain products, including software.
- The ISO 9001 standard is not itself a standard for software development but is a framework for developing software standards.
- It sets out general quality principles, describes quality processes in general, and lays out the organizational standards and procedures that should be defined.
- These should be documented in an organizational quality manual.

- The major revision of the ISO 9001 standard in 2000 reoriented the standard around nine core processes

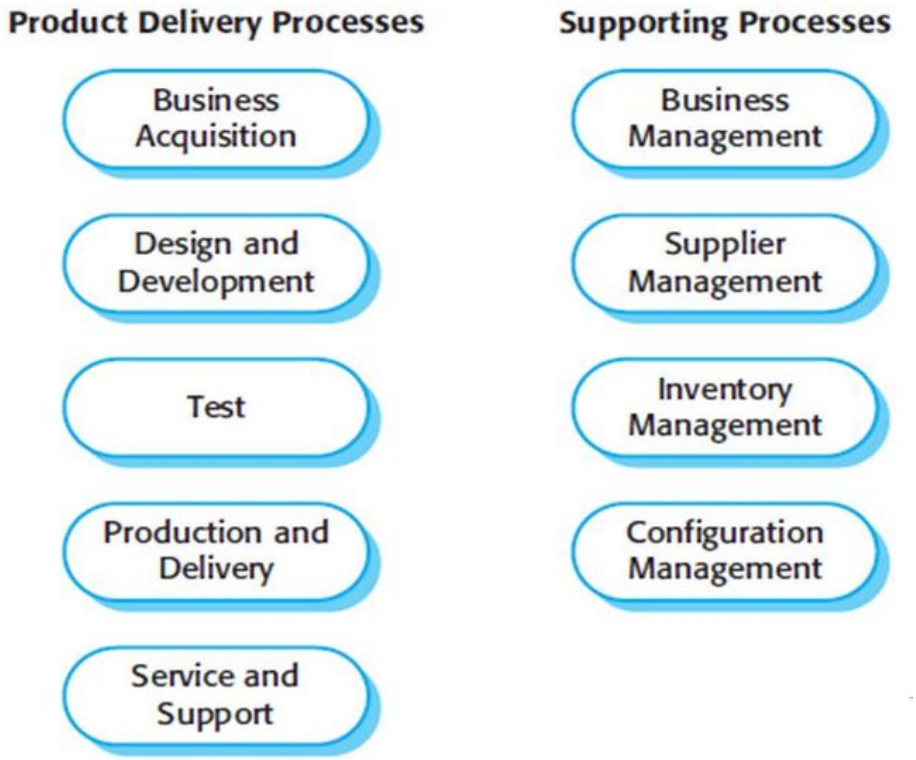


Fig: ISO 9001 core processes

- The relationships between ISO 9001, organizational quality manuals, and individual
- project quality plans are shown in ISO 9001 and quality management shown in following Figure.
- This diagram has been derived from a model given by Ince (1994), who explains how the general ISO 9001 standard can be used as a basis for software quality management processes.

The guideline steps are:

- Support the quality
- Satisfy the customer
- Establish quality policy

- Control quality system
- Provide quality resources
- Provide quality personnel
- Document the quality management system
- Perform management reviews.
- conduct quality system
- control customer processes
- control monitoring devices
- analyse quality information

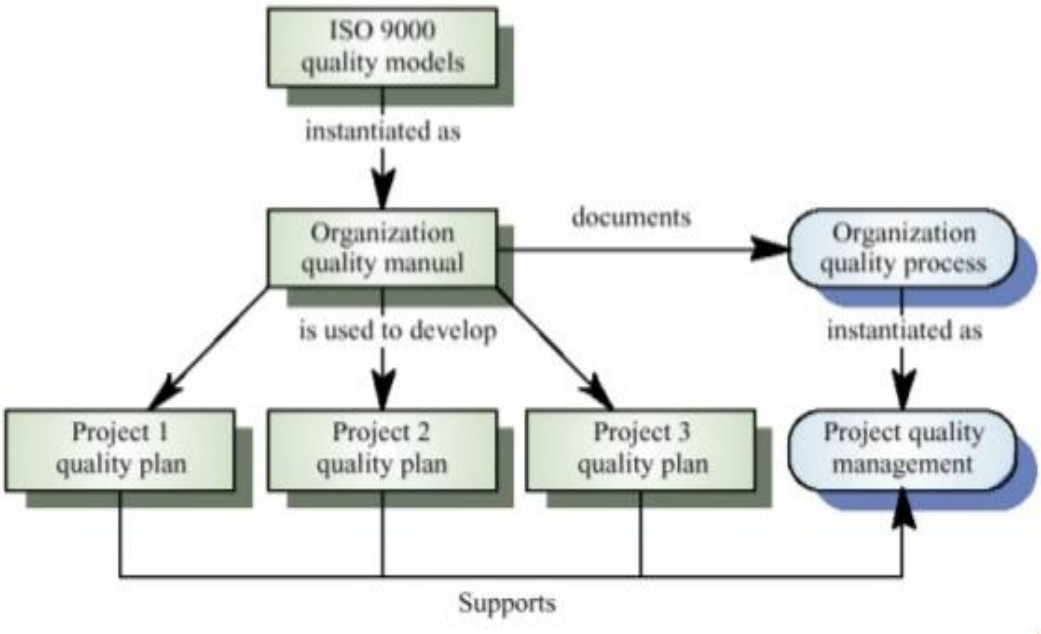


Fig: ISO 9001 and quality management

Review and inspection

1. Reviews and inspections are QA activities that check the quality of project deliverables. This involves examining the software, its documentation, and records of the process to discover errors and omissions and to see if quality standards have been followed.

2. During a review, a group of people examine the software and its associated documentation, looking for potential problems and non-conformance with standards.

3. The review team makes informed judgments about the level of quality of a system or project deliverable. Project managers may then use these assessments to make planning decisions and allocate resources to the development process.

The purpose of reviews and inspections is to improve software quality, not to assess the performance of people in the development team.

1. Reviewing is a public process of error detection, compared with the more private component-testing process.

2. Inevitably, mistakes that are made by individuals are revealed to the whole programming team.

3. To ensure that all developers engage constructively with the review process, project managers must be sensitive to individual concerns.

4. They must develop a working culture that provides support without blame when errors are discovered.

Although a quality review provides information for management about the software being developed, quality reviews are not the same as management progress reviews.

Progress reviews take external factors into account and changed circumstances may mean that software under development is no longer required or has to be radically changed.

16.3.2 The review processes

1. Pre-review activities

These are preparatory activities that are essential for the review to be effective. Typically, pre-review activities are concerned with review planning and review preparation. Review planning involves setting up a review team, arranging a time and place for the review, and distributing the documents to be reviewed. During review preparation, the team may meet to get an overview of the software to be reviewed. Individual review team members read and understand the software or documents and relevant

standards. They work independently to find errors, omissions, and departures from standards. Reviewers may supply written comments on the software if they cannot attend the review meeting.

2. The review meeting

During the review meeting, an author of the document or program being reviewed should walk through' the document with the review team. The review itself should be relatively short-two hours at most. One team member should chair the review, and another should formally record all review decisions and actions to be taken. During the review, the chair is responsible for ensuring that all written comments are considered. The review chair should sign a record of comments and actions agreed during the review.

3. Post-review activities

After a review meeting has finished, the issues and problems raised during the review must be addressed.

This may involve fixing software bugs, refactoring software so that it conforms to quality standards, or rewriting documents.

Sometimes, the problems discovered in a quality review are such that a management review is also necessary to decide if more resources should be made available to correct them.

After changes have been made, the review chair may check that the review comments have all been considered.

Sometimes, a further review will be required to check that the changes made cover all the previous review comments.

16.5 Software measurement and metrics

- Software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process.
- By comparing these values to each other and to the standards that apply across an organization, you may be able to draw conclusions about the quality of software, or assess the effectiveness of software processes, tools, and methods.

- The long-term goal of software measurement is to use measurement in place of reviews to make judgments about software quality.
- Using software measurement, a system could ideally be assessed using a range of metrics and, from these measurements, a value for the quality of the system could be inferred.
- If the software had reached a required quality threshold, then it could be approved without review.
- When appropriate, the measurement tools might also highlight areas of the software that could be improved. However, we are still quite a long way from this ideal situation and, there are no signs that automated quality assessment will become a reality in the foreseeable future.

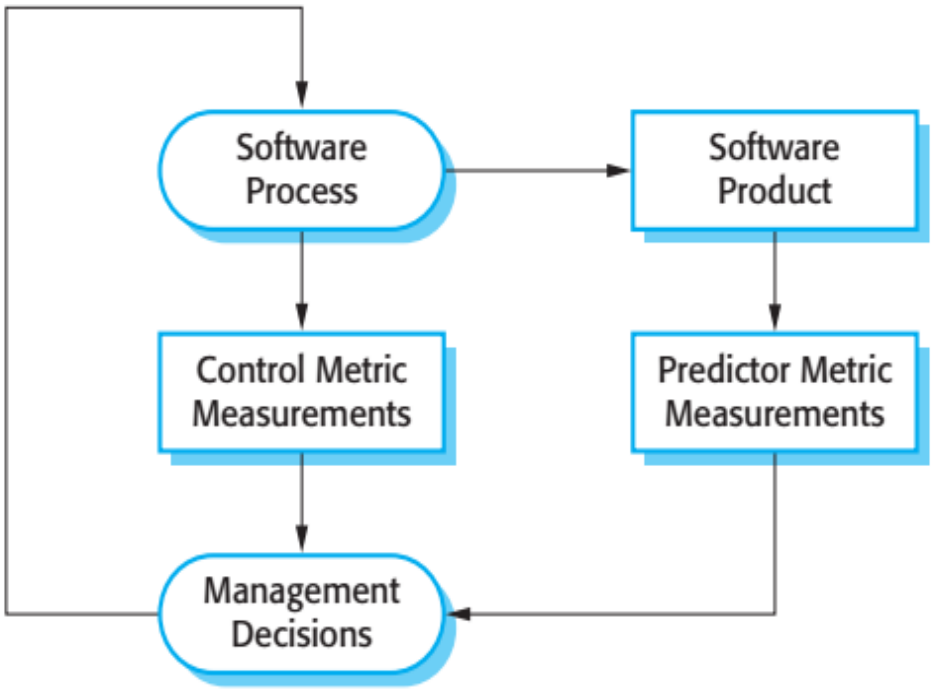


Fig: Predictor and control measurement

- A software metric is a characteristic of a software system, system documentation, or development process that can be objectively measured.
- Examples of metrics include the size of a product in lines of code.
- Software metrics may be either control metrics or predictor metrics. As the names imply, control metrics support process management, and predictor metrics help you predict characteristics of the software.

- Control metrics are usually associated with software processes.
- Examples of control or process metrics are the average effort and the time required to repair reported defects.

There are two ways in which measurements of a software system may be used:

1. To assign a value to system quality attributes

By measuring the characteristics of system components, such as their cyclomatic complexity, and then aggregating these measurements, you can assess system quality attributes, such as maintainability.

2. To identify the system components whose quality is substandard

Measurements can identify individual components with characteristics that deviate from the norm. For example, you can measure components to discover those with the highest complexity. These are most likely to contain bugs because the complexity makes them harder to understand.

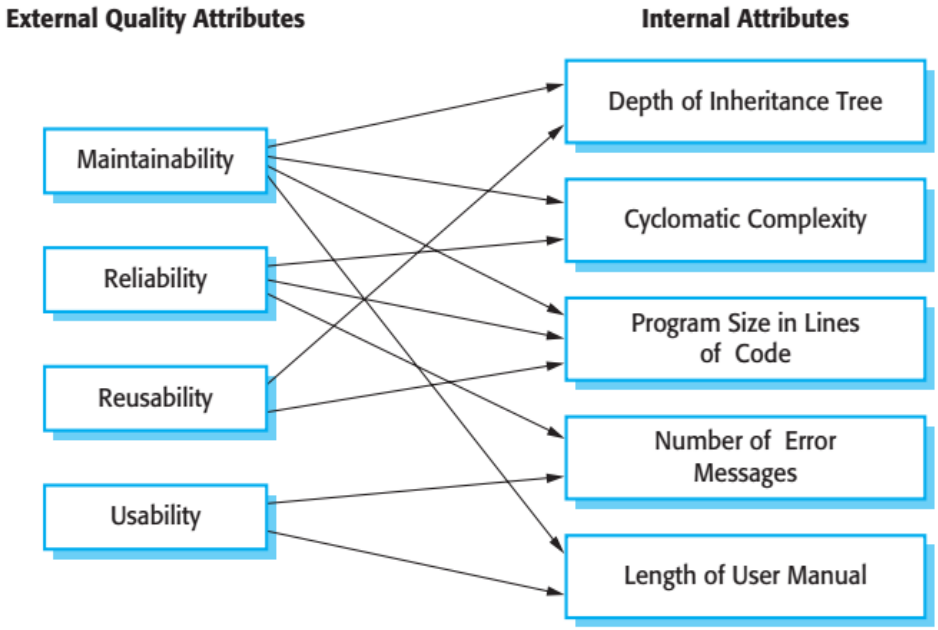


Fig: Relationships between internal and external software

The diagram suggests that there may be relationships between external and internal attributes, but it does not say how these attributes are related. If the measure of the internal attribute is to be a useful predictor of the external software characteristic, three conditions must hold.

1. **The internal attribute must be measured accurately.**

This is not always straightforward, and it may require special-purpose tools to make the measurements.

2. **A relationship must exist between the attribute**

that can be measured and the external quality attribute that is of interest. That is, the value of the quality attribute must be related, in some way, to the value of the attribute than can be measured.

3. **This relationship between the internal and external attributes**

must be understood, validated, and expressed in terms of a formula or model. Model formulation involves identifying the functional form of the model (linear, exponential, etc.) by analysis of collected data, identifying the parameters that are to be included in the model, and calibrating these parameters using existing data.

16.5.1 Product metrics

Product metrics are predictor metrics that are used to measure internal attributes of a software system. Examples of product metrics include the system size, measured in lines of code, or the number of methods associated with each object class.

Product metrics fall into two classes:

1. **Dynamic metrics**, which are collected by measurements made of a program in execution. These metrics can be collected during system testing or after the system has gone into use. An example might be the number of bug reports or the time taken to complete a computation.

2. **Static metrics,** which are collected by measurements made of representations of the system, such as the design, program, or documentation. Examples of static metrics are the code size and the average length of identifiers used.

- These types of metric are related to different quality attributes. Dynamic metrics help to assess the efficiency and reliability of a program. Static metrics help assess the complexity, understandability, and maintainability of a software system or system components.

- There is usually a clear relationship between dynamic metrics and software quality characteristics.

- It is easy to measure the execution time required for functions and to assess the time required to start up a system.
- These relate directly to the system's efficiency. Similarly, the number of system failures and the type of failure can be logged and related directly to the reliability of the software

16.5.2 Software component analysis

A measurement process that may be part of a software quality assessment process is shown in below Figure Each system component can be analysed separately using a range of metrics. The values of these metrics may then be compared for different components and, perhaps, with historical measurement data collected on previous projects. Anomalous measurements, which deviate significantly from the norm, may imply that there are problems with the quality of these components.

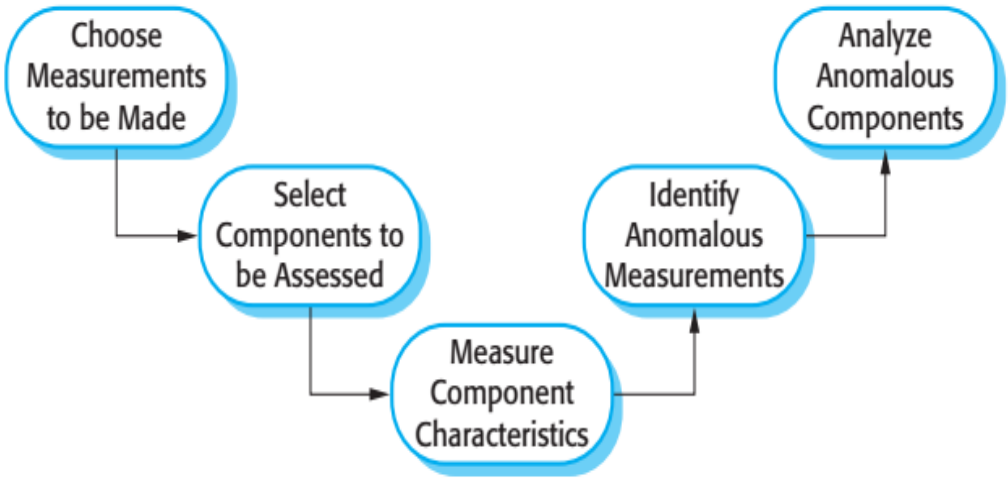


Fig: The process of product measurement

The key stages in this component measurement process are:

1. **Choose measurements to be made**

   The questions that the measurement is intended to answer should be formulated and the measurements required to answer these questions defined. Measurements that are not directly relevant to these questions need not be collected.

4. **Select components to be assessed**

You may not need to assess metric values for all the components in a software system. Sometimes, you can select a representative selection of components for measurement, allowing you to make an overall assessment of system quality. At other times, you may wish to focus on the core components of the system that are in almost constant use. The quality of these components is more important than the quality of components that are only rarely used.

5. Measure component characteristics

The selected components are measured, and the associated metric values computed. This normally involves processing the component representation (design, code, etc.) using an automated data collection tool. This tool may be specially written or may be a feature of design tools that are already in use.

6. Identify anomalous measurements

After the component measurements have been made, you then compare them with each other and to previous measurements that have been recorded in a measurement database. You should look for unusually high or low values for each metric, as these suggest that there could be problems with the component exhibiting these values.

7. Analyse anomalous components

When you have identified components that have anomalous values for your chosen metrics, you should examine them to decide whether these anomalous metric values mean that the quality of the component is compromised. An anomalous metric value for complexity (say) does not necessarily mean a poor-quality component. There may be some other reason for the high value, so may not mean that there are component quality problems.

Graded Question

1. Explain the process of Software Quality Management
2. Explain quality assurance and standards
3. Explain how product quality can be planned
4. Explain what is software metric and measurement?
5. What are the key component measurement process?

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition

2. Software Engineering, Pankaj Jalote Narosa Publication

3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

❖ ❖ ❖

17

PROCESS IMPROVEMENT

Unit Structure:

17.0 Objective:

In this chapter you will understand What is process and the Quality.

Understand the principles of software process improvement

Understand the cycle process improvement process

Process Improvement:

Process improvement is a cyclical activity. It involves three principal stages:

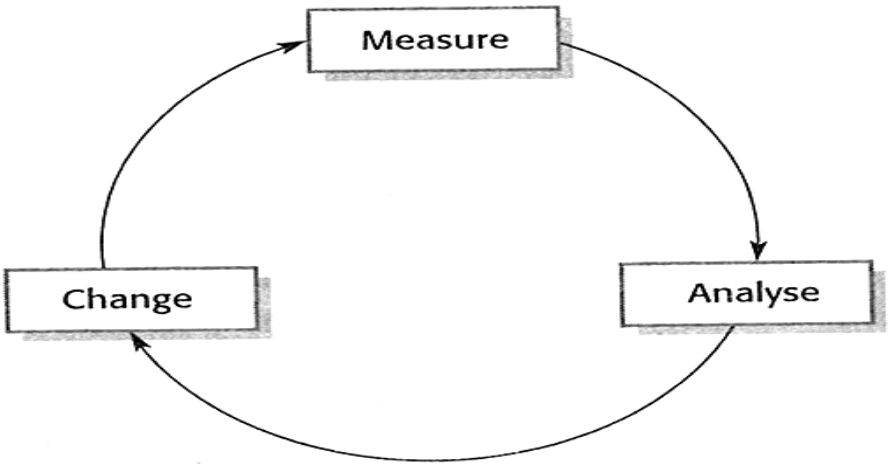


Fig.: The process improvement cycle

1) **Process measurement:** Attributes of the current project or the product are measured. The aim is to improve the measures according to the goals of the organization involved in process improvement.

2) **Process analysis:** The current process is assessed, and process weaknesses bottlenecks are identified. Process models that describe the process are usually developed during this stage.

3) **Process change:** Changes to the process that have been identified during analysis are introduced.

17.1 Process and Product Quality

1 . A fundamental assumption of quality management is that the quality of the development process directly affects the quality of delivered products.

2. This assumption comes from manufacturing systems where product quality is intimately related the production process.

3 . In an automated manufacturing system, the process involves configuring, setting up and operating the machines involved in process.

4. Once the machines are operating correctly, product quality naturally follows. You measure the quality of the product and change the process until you achieve the quality level that you need.

5. There is a clear link between process and product quality in manufacturing because the process is relatively easy to standardize and monitor.

6. Once manufacturing systems are calibrated, they can be run again and again to output high-quality products.

7. However, software is not manufactured but is designed.

8. Software development is a creative rather than a mechanical process, so the influence of Individual skills and experience is significant.

9. External factors, such as the novelty of an application or commercial pressure for an early product release, also affect product quality irrespective of the process used.

17.1.1 Process quality management involves:

1) Defining process standards such as how and when reviews should be conducted

2) Monitoring the development process to ensure that the standards are being followed

3) Reporting the software process to project management and to the buyer of the software

One problem with process-based quality assurance is that the quality assurance (QA) team may insist that standard processes should be used irrespective of the type of software that is being developed.

For example, process quality standards for critical systems may specify that specification must be complete and approved before implementation can begin.

However, some critical systems may require prototyping where programs are implemented without a complete specification.

I have experienced situations where the quality management team suggests that this prototyping should not be carried out because the prototype quality cannot be monitored.

In such situations, senior management must intervene to ensure that the quality process supports rather than hinders product development.

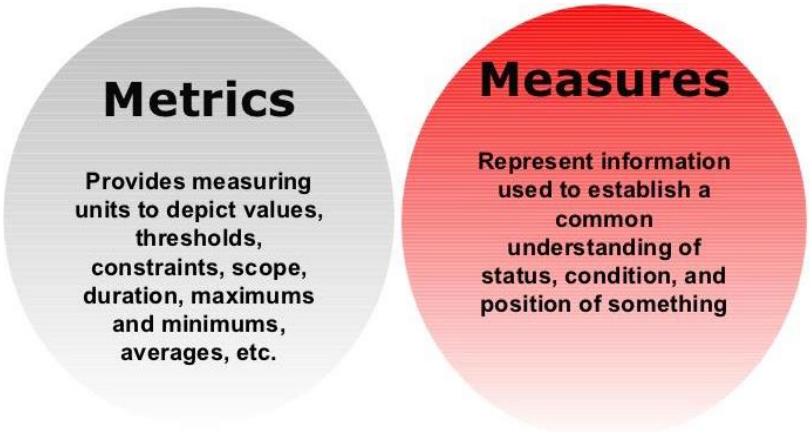


Fig: Metrics and Measures

Process characteristic	Description
Understandability	To what extent is the process explicitly defined and how easy is it to understand the process definition?
Visibility	Do the process activities culminate in clear results so that the progress of the process is externally visible?
Supportability	To what extent can CASE tools be used to support the process activities?
Acceptability	Is the defined process acceptable to and usable by the engineers responsible for producing the software product?
Reliability	Is the process designed in such a way that process errors are avoided or trapped before they result in product errors?
Robustness	Can the process continue in spite of unexpected problems?
Maintainability	Can the process evolve to reflect changing organizational requirements or identified process improvements?
Rapidity	How fast can the process of delivering a system from a given specification be completed?

Fig: Process characteristics

17.2 Process Classification

1)Informal processes. When there is no strictly defined process model, the development team chooses the process that they will use. Informal processes may use formal procedures such as configuration management, but the procedures and the relationships between procedures are defined as required by the development team.

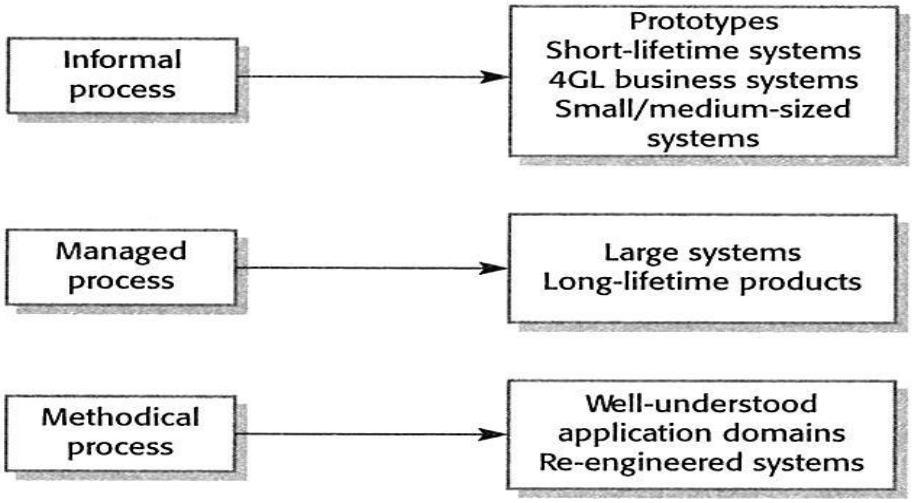


Fig.: Process applicability

2) **Managed processes**. A defined process model is used to drive the development process. The process model defines the procedures, their scheduling, and the relationships between the procedures.

3) **Methodical processes**. When some defined development method are used, these processes benefit from CASE tool support for design and analysis processes.

4) **Improving processes**. Processes that have inherent improvement objectives have a specific budget for improvements and procedures for introducing SUCH improvements. As part of this, quantitative process measurement may be introduced.

17.3 Process Measurement

Process measurements are quantitative data about the software process.

- The measurement of process and product attributes is essential for process improvement.

- Measurement has an important role to play in small-scale, personal process improvement.
- Process measurements can be used to assess whether the efficiency of a process has been improved.
- For example, the effort and time devoted to testing can be monitored.
- Effective improvements to the testing process should reduce the effort, testing time or both.
- However, process measurements on their own cannot be used to determine whether product quality has improved.
- Product quality data should also be collected and related to the process activities

Three classes of process metric can be collected:

1) **The time taken for a process to be completed**. This can be the total time devoted to the process, calendar time, the time spent on the process by engineers.

2) **The resources required for a process.** The resources might include total effort in person-days, travel costs and computer resources.

3) **The number of occurrences of an event**. Examples of events that might be monitored include the number of defects discovered during code inspection, the number of requirements changes requested and the average number of lines of code modified in response to a requirement change.

This approach relies on the identification of:

1) Goals. What the organization is trying to achieve. Examples of goals might be improved programmer productivity, shorter product development time and increased product reliability.

2) Questions. These are refinements of goals where specific areas of uncertainty related to the goals are identified. Normally, a goal will have several associated questions that need to be answered. Examples of questions related to the above goals are:

    - How can the number of debugged lines of code be increased?

- How can the time required to finalize product requirements be reduced?

How can more effective reliability assessments be made?

3) Metrics. These are the measurements that need to be collected to help answer the questions and to confirm whether process improvements have achieved the desired goal.

Advantages

1) Separate organizational concerns from specific process concerns.

2) focus on data collection and suggest the data to be analyse in different ways depending on question planned to answer.

17.4 Process Change

Process change involves making modifications to the existing process. You may do this by introducing new practices, methods, or tools, by changing the ordering of process activities, by introducing or removing deliverables from the process, or by introducing new roles and responsibilities.

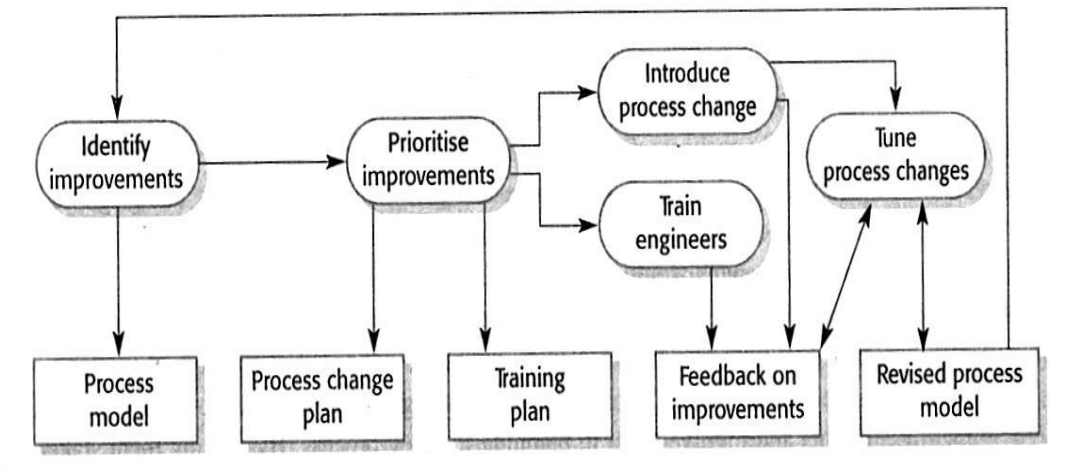


Fig.: The process change process

There are five key stages in the process change process:

1)Improvement identification. This stage is concerned with using the results of the process analysis to identify quality, schedule, or cost bottlenecks where process factors might adversely influence the product quality

2) **Improvement prioritizations.** This stage is concerned with assessing the possible changes and prioritizing them for implementation.

   Which are most important. You may make these decisions based on the need to improve specific process areas, the costs of introducing the change, the impact of the change on the organization and other factors.

3) **Process changes introduction.** Process change introduction means putting new procedures, methods, and tools into place, and integrating them with other process activities. You must allow enough time to introduce changes and to ensure that these changes are compatible with other process activities and with organizational procedures and standards.

4) **Process change training.** Without training, it is not possible to gain the full benefits from process changes. Process managers and software engineers may simply refuse to accept the new process.

5) **Change tuning.** Proposed process changes will never be completely effective as soon as they are introduced. You need a tuning phase where minor problems are discovered, and modifications to the process are proposed and are introduced.

17.5 The CMMI Process Improvement Framework

The CMMI model is intended to be a framework for process improvement that has broad applicability across a range of companies.

Its staged version is compatible with the Software CMM and allows an organization's system development and management processes to be assessed and assigned a maturity level from 1 to 5.

Category	Process area
Process management	Organizational process definition Organizational process focus Organizational training
Project management	Project planning Project monitoring and control Supplier agreement management Integrated project management Risk management

Engineering	Requirements management Requirements development Technical solution
Support	Configuration management Process and product quality management Measurement and analysis Decision analysis and resolution

Fig: Process areas in the CMMI

1) Process areas. The CMMI identifies 24 process areas that are relevant to software process capability and improvement. These are organized into four groups in the continuous CMMI model. These groups and related process areas are listed above.

2) Goals. Goals are abstract descriptions of a desirable state that should be attained by an organization. The CMMI has specific goals that are associated with each process area and that define the desirable state for that area. It also defines generic goals that are associated with the institutionalization of good practice.

3) Practices. Practices in the CMMI are descriptions of ways to achieving a goal. Up to seven specific and generic practices may be associated with each goal within each process area.

Goal	Process Area
Corrective actions are managed to closure when the project's performance or results deviate significantly from the	Specific goal in project monitoring and control
Actual performance and progress of the project is monitored against the project	Specific goal in project monitoring and control
The requirements are analyzed and validated, and a definition of the required functionality is developed	Specific goal in requirements development
Root causes of defects and other problems are systematically	Specific goal in causal analysis and resolution
The process is institutionalized as a defined process	Generic goal

Fig: Process area in the CMMI

A CMMI assessment involves examining the processes in an organization and rating these on a six-point scale that relates to the level of maturity in each process area.

The six-point scale assigns a level to a process as follows:

1. **Incomplete:** At least one of the specific goals associated with the process area is not satisfied. There are no generic goals at this level as institutionalization of an incomplete process does not make sense.

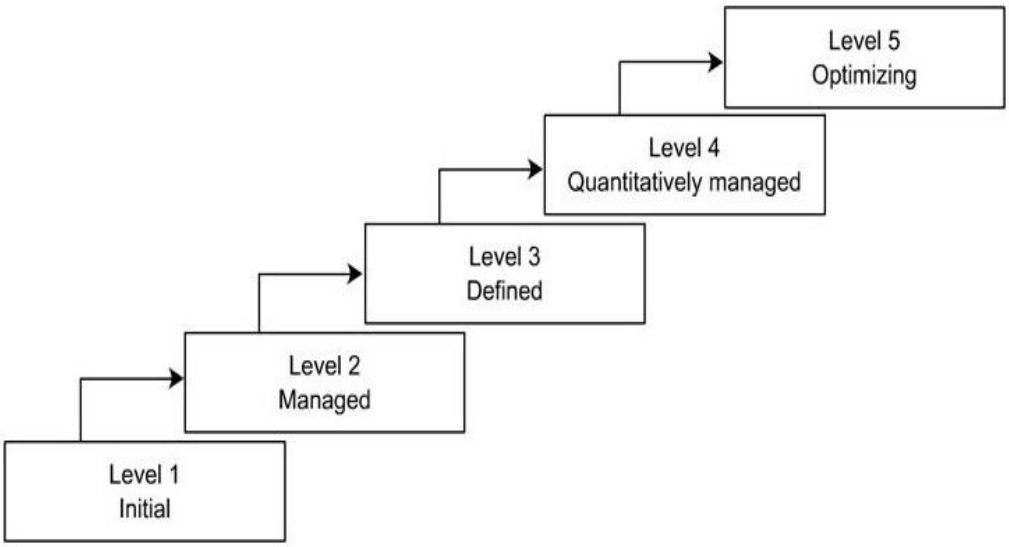
2. **Performed**: The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.

3. **Managed:** At this level, the goals associated with the process area are met and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.

4. **Defined**: This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.

5. **Quantitatively:** managed at this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses; that is, collected process and product measurements must be used in process management.

6. **Optimizing:** At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed, and the processes adapted to changing business needs.



17.5.1 The staged CMMI model

1. **Requirements management** Manage the requirements of the project's products and product components and identify inconsistencies between those requirements and the project's plans and work products.

2. **Project planning** Establish and maintain plans that define project activities.

3. **Project monitoring and control** Provide understanding into the project's progress so that appropriate corrective actions can be taken when the project's performance deviates significantly from the plan.

4. **Supplier agreement management** Manage the acquisition of products and services from suppliers external to the project for which a formal agreement exists.

5. **Measurement and analysis** Develop and sustain a measurement capability that is used to support management information needs.

6. **Process and product quality assurance** Provide staff and management with objective insight into the processes and associated work products.

7. **Configuration management** Establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

17.6 Service Oriented software engineering

Service-oriented architectures (SOA) are a way of developing distributed systems where the components of these systems are stand-alone services. These services may execute on geographically distributed computers. Standard protocols have been designed to support service communication and information exchange. Consequently, services are platform and implementation-language independent. Software systems can be constructed using services from different providers with seamless interaction between these services.

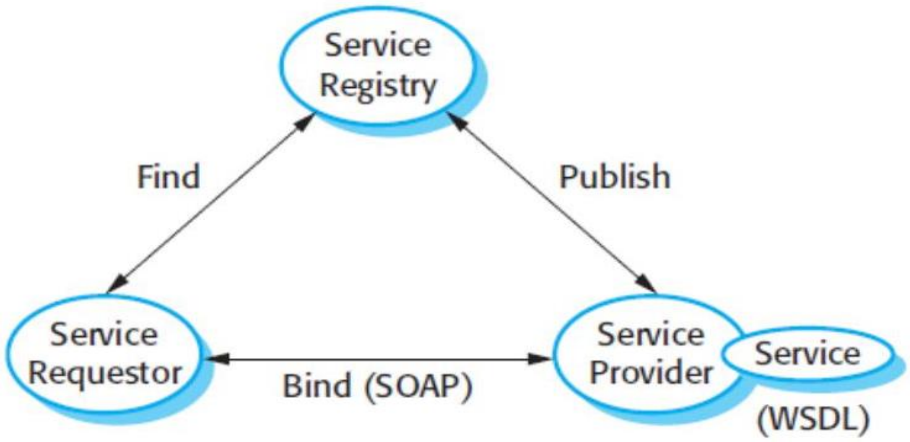


Fig: Service oriented architecture

- Service providers design and implement services and specify these services in a language called WSDL.
- They also publish information about these services in a generally accessible registry using a publication standard called UDDI.
- Service-oriented architecture is now generally recognized as a significant development, particularly for business application systems.
- It allows flexibility as services can be provided locally or outsourced to external providers.
- Services may be implemented in any programming language. By wrapping legacy systems as services, companies can preserve their investment in valuable software and make this available to a wider range of applications.

- SOA allows different platforms and implementation technologies that may be used in different parts of a company to inter-operate.

The key reason for the success of service-oriented architectures is the fact that,

1. From the outset, there has been an active standardization process working alongside technical developments.
2. All the major hardware and software companies are committed to these standards.
3. As a result, service-oriented architectures have not suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology.
4. Hence, problems, such as the multiple incompatible component models in CBSE have not arisen in service-oriented system development.

The key standards for web service-oriented architectures are:

1) SOAP: This is a message interchange standard that supports the communication between services. It defines the essential and optional components of messages passed between services.

2) WSDL: The Web Service Definition Language (WSDL) standard defines the way in which service providers should define the interface to these services. Essentially, it allows the interface of a service (the service operations, parameters, and their types) and its bindings to be defined in a standard way.

3) UDDI: The UDDI (Universal Description, Discovery, and Integration) standard defines the components of a service specification that may be used to discover the existence of a service. These include information about the service provider, the services provided, the location of the service description (usually expressed in WSDL) and information about business relationships. UDDI registries enable potential users of a service to discover what services are available.

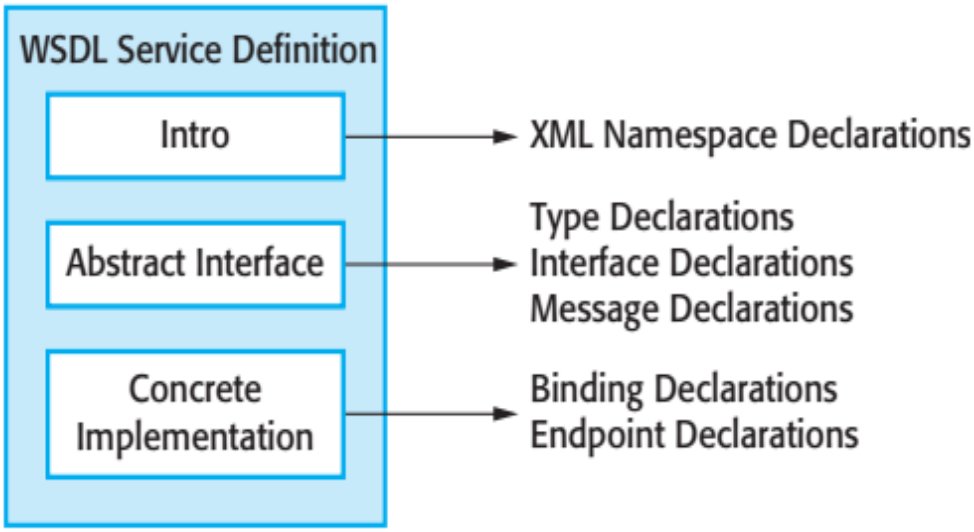
4) WS-BPEL: This standard is a standard for a workflow language that is used to define process programs involving several different services.

17.7 Services as Reusable Components

Software systems are constructed by composing software components that are based on some standard component model. Services are a natural development of software components where the component model is the set of standards associated with web services. A service can therefore be defined as:

A loosely coupled, reusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed. A web service is a service that is accessed using standard Internet and XML-based protocols.

The WSDL conceptual model shows in the below figure the elements of a service description. Each of these is expressed in XML and may be provided in separate files. These parts are:



1. An introductory part that usually defines the XML namespaces used and which may include a documentation section providing additional information about the service.

2. An optional description of the types used in the messages exchanged by the service.

3. A description of the service interface; that is, the operations that the service provides for other services or users.

4. A description of the input and output messages processed by the service.

17.8 Service Engineering

Service engineering is the process of developing services for reuse in service-oriented applications.

It has much in common with component engineering. Service engineers must ensure that the service represents a reusable abstraction that could be useful in different systems. They must design and develop generally useful functionality associated with that abstraction and ensure that the service is robust and reliable.

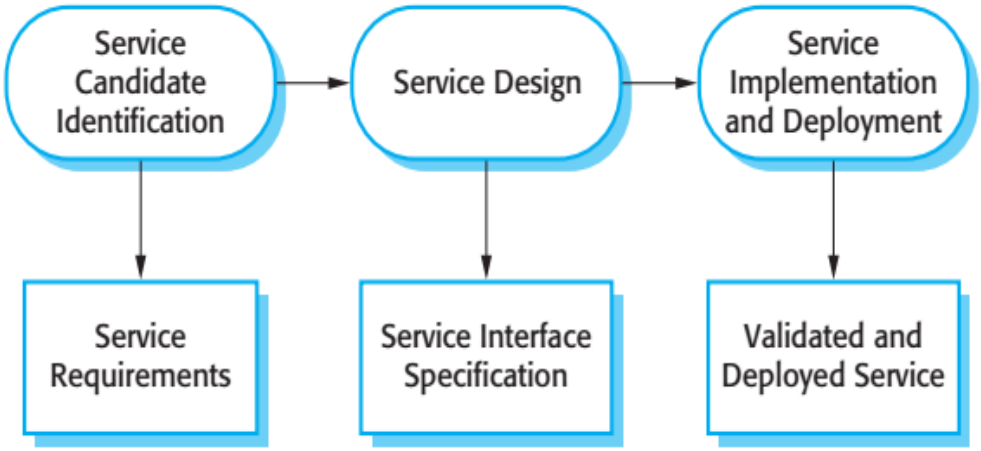


Fig: The service engineering process

There are three logical stages in the service engineering process

1. Service candidate identification, where you identify possible services that might be implemented and define the service requirements.
2. Service design, where you design the logical and WSDL service interfaces.
3. Service implementation and deployment, where you implement and test the service and make it available for use.

17.8.1 There are three fundamental types of service that may be identified:

1. **Utility services** These are services that implement some general functionality that may be used by different business processes. An example of a utility service is a currency conversion service that be accessed to compute the conversion of one currency (e.g., dollars) to another (e.g., euros).

2. **Business services** These are services that are associated with a specific business function. An example of a business function in a university would be the registration of students for a course.

3. **Coordination or process services** These are services that support a more general business process which usually involves different actors and activities. An example of a coordination service in a company is an ordering service that allows orders to be placed with suppliers, goods accepted, and payments made.

17.8.2 Service interface design

1. This involves defining the operations associated with the service and their parameters.

2. Your aim should be to minimize the number of message exchanges that must take place to complete the service request.

3. You must ensure that as much information as possible is passed to the service in a message rather than using synchronous service interactions.

There are three stages to service interface design:

1. Logical interface design, where you identify the operations associated with the service, their inputs and outputs and the exceptions associated with these operations.

2. Message design, where you design the structure of the messages that are sent and received by the service.

3. WSDL development, where you translate your logical and message design to an abstract interface description written in WSDL.

17.9 Software development with service

The development of software using services is based around the idea that you compose and configure services to create new, composite services. The services involved in the composition may be specially developed for the application, may be business services developed within a company, or may be services from an external provider.

Service composition may be used to integrate separate business processes to provide an integrated process offering more extensive functionality.

There are six key stages in the process of service construction by composition:

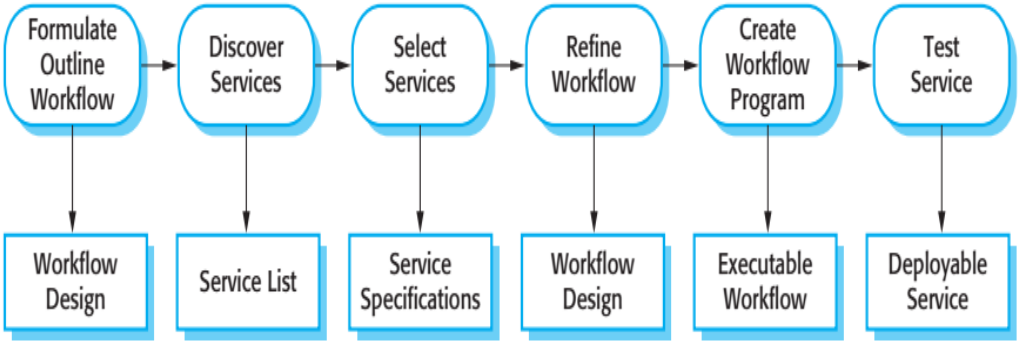


Fig: Service construction by composition

1. **Formulate outline workflow** in this initial stage of service design, you use the requirements for the composite service as a basis for creating an 'ideal' service design. You should create an abstract design at this stage with the intention of adding details once you know more about available services.

2. **Discover services** During this stage of the process, you search service registries or catalogs to discover what services exist, who provides these services, and the details of the service provision.

3. **Select possible services** from the set of possible service candidates that you have discovered, you then select possible services that can implement workflow activities. Your selection criteria will obviously include the functionality of the services offered. They may also include the cost of the services and the quality of service (responsiveness, availability, etc.) offered.

4. **Refine workflow** based on information about the services that you have selected, you then refine the workflow. This involves adding detail to the abstract description and perhaps adding or removing workflow activities. You may then repeat the service discovery and selection stages. Once a stable set of services has been chosen and the final workflow design established, you move on to the next stage in the process.

5. **Create workflow** program During this stage, the abstract workflow design is transformed to an executable program and the service interface is defined. You can use a conventional programming language, such as Java or C#, for service implementation or a workflow language, such as WS-BPEL. As I discussed in the previous section, the service interface specification should

be written in WSDL. This stage may also involve the creation of web-based user interfaces to allow the new service to be accessed from a web browser.

6. **Test completed service or application** the process of testing the completed, composite service is more complex than component testing in situations where external services are used.

Graded Question

1. Write a note on ISO 9000 quality standards.
2. Explain process improvement cycle.
3. Explain CMMI framework.
4. Explain Software as service.
5. Explain service engineering with neat diagram.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition
2. Software Engineering, Pankaj Jalote Narosa Publication
3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh

❖ ❖ ❖

18

SECURITY ENGINEERING

Unit Structure:

18.0 Objective:

The objective of this chapter is to understand what are the different issues that need to be considered when you are designing secure application system.

Understand the difference between infrastructure security and application security.

Understand how life cycle risk assessment.

As we are making use of Internet everywhere and for every information this introduce, a new challenge for software engineers designing and implementing systems that were secure. As more and more systems were connected to the Internet, a variety of different external attacks were devised to threaten these systems. The problems of producing depend able systems were hugely increased.

It is now essential to design systems to withstand external attacks and to recover from such attacks.

Security engineering is concerned with the development and evolution of systems that can resist malicious attacks, which are intended to damage the system or its data. Software security engineering is part of the more general field of computer security.

18.1 Complex applications

Infrastructure for complex application are as follows:

- an operating system platform, such as Linux or Windows.
- other generic applications that run on that system, such as web browsers and e-mail clients.
- a database management system.
- middleware that supports distributed computing and database access.
- libraries of reusable components that are used by the application software.

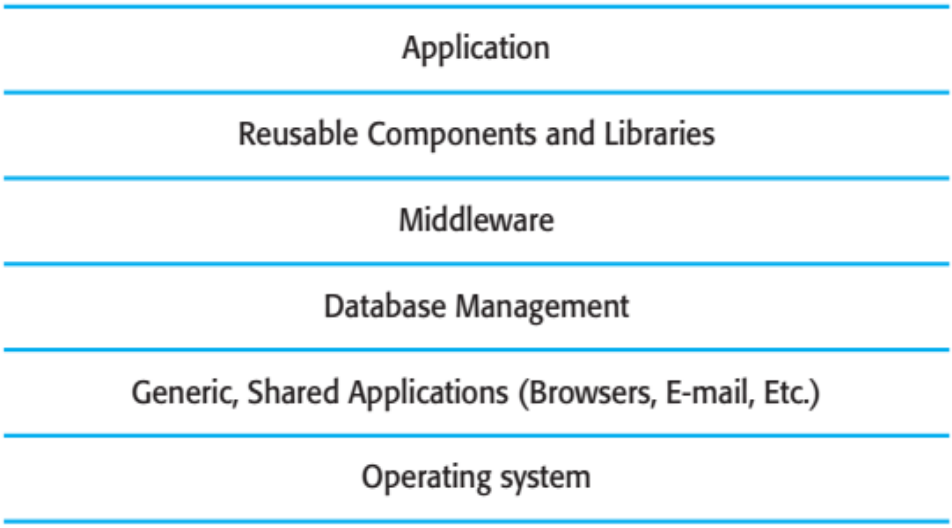


Fig: System layers were security may be compromised

Majority of attacks done on component's or we can say system infrastructure because most of hackers know how the components are hint most of the components infrastructure available in market.

There is major difference between application security and infrastructure security:

1. application security related to the software applications so software engineering need to secure the software and ensure that the system or the model should be build such a way it can resist the attack.

2. Infrastructure security is related to the infrastructure of the system or organization so infrastructure managers need to look after the infrastructure to be get secured from different attack.

System security management is not a single task but includes a range of activities such as user and permission management, system software deployment and maintenance, and attack monitoring, detection, and recovery:

1. User and permission management includes adding and removing users from the system, ensuring that appropriate user authentication mechanisms are in place and setting up the permissions in the system so that users only have access to the resources that they need.

2. System software deployment and maintenance includes installing system software and middleware and configuring these properly so that security vulnerabilities are avoided. It also involves updating this software regularly with new versions or patches, which repair security problems that have been discovered.

3. Attack monitoring, detection and recovery includes activities which monitor the system for unauthorized access, detect, and put in place strategies for resisting attacks, and backup activities so that normal operation can be resumed after an external attack.

18.2 Security risk management

Security risk assessment and management is essential for effective security engineering. Risk management is concerned with assessing the possible losses that might ensue from attacks on assets in the system and balancing these losses against the cost of security procedures that may reduce these losses.

Risk assessment starts before the decision to acquire the system has been made and should continue throughout the system development process and after the system has gone into use

The idea that this risk assessment is a staged process:

1. **Preliminary risk assessment** at this stage, decisions on the detailed system requirements, the system design, or the implementation technology have not been made. The aim of this assessment process is to decide if an adequate level of security can be achieved at a reasonable cost. If this is the case, you can then derive specific security requirements for the system. You do not have information about potential vulnerabilities in the system or the controls that are included in reused system components or middleware.

2. **Life-cycle risk assessment** This risk assessment takes place during the system development life cycle and is informed by the technical system design and implementation decisions. The results of the assessment may lead to changes to the security requirements and the addition of new requirements. Known and potential vulnerabilities are identified, and this knowledge is used to inform decision making about the system functionality and how it is to be implemented, tested, and deployed.

3. **Operational risk assessment** After a system has been deployed and put into use, risk assessment should continue to take account of how the system is used and proposals for new used and proposals for new and changed requirements. Assumptions about the operating requirement made when the system was specified may be incorrect. Organizational changes may mean that the system is used in different ways from those originally planned. Operational risk assessment therefore leads to new security requirements that have to be implemented as the system evolves.

Starting point for identifying possible misuse case which will be characterize under four headings.

Misuse cases are not just useful in preliminary risk assessment but may be used for security analysis in life-cycle risk analysis and operational risk analysis.

They provide a useful basis for playing out hypothetical attacks on the system and assess ing the security implications of design decisions that have been made.

1. Interception threats that allow an attacker to gain access to an asset. a situation where an attacker gains access to the records of an individual celebrity patient.

2. Interruption threats that allow an attacker to make part of the system unavailable. Therefore, a possible misuse case might be a denial of service attack on a system database server.

3. Modification threats that allow an attacker to tamper with a system asset, where an attacker changes the information in a patient record.

4. Fabrication threats that allow an attacker to insert false information into a system, where false transactions might be added to the system that transfer money to the perpetrator's bank account.

18.2.1 Life-cycle risk assessment

Based on organizational security policies, preliminary risk assessment should identify the most important security requirements for a system.

1. These reflect how the security policy should be implemented in that application, identify the assets to be protected, and decide what approach should be used to provide that protection.

2. maintaining security is about paying attention to detail. It is impossible for the initial security requirements to take all details that affect security into account.

3. Life-cycle risk assessment identifies the design and implementation details that affect security.

4. This is the important distinction between life-cycle risk assessment and preliminary risk assessment.

5. Life-cycle risk assessment affects the interpretation of existing security requirements, generates new requirements, and influences the overall design of the system.

6. You need to know what the parts and application are need to secure first, For example, a vulnerability in all password-based systems is that an authorized user reveals their password to an unauthorized user.

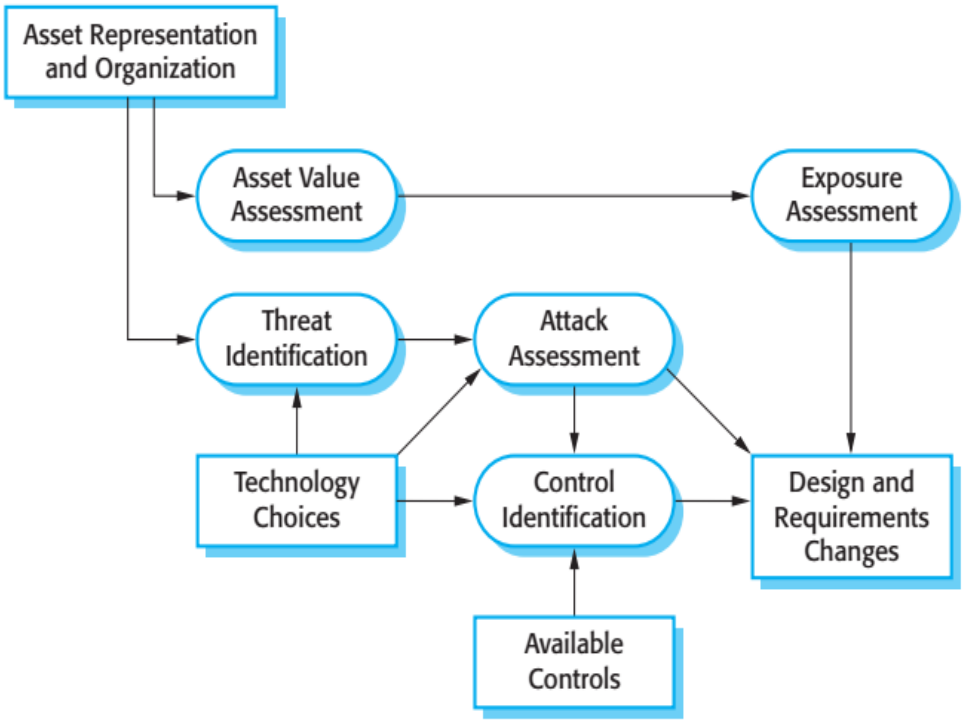


Fig: Life-cycle risk analysis

7. Security risk assessment should be part of all life-cycle activities from requirements engineering to system deployment.

8. The process followed is like the pre liminary risk assessment process with the addition of activities concerned with design vulnerability identification and assessment.

9. The outcome of the risk assessment is a set of engineering decisions that affect the system design or implementation or limit the way in which it is used.

Two examples illustrate how protection requirements are influenced by decisions on information representation and distribution:

1. You may make a design decision to separate personal patient information and information about treatments received, with a key linking these records. The treatment information is much less sensitive than the personal patient information so may not need as extensive protection. If the key is protected, then an attacker will only be able to access routine information, without being able to link this to an individual patient.

2. Assume that, at the beginning of a session, a design decision is made to copy patient records to a local client system. This allows work to continue if the server is unavailable. It makes it possible for a health-care worker to access patient records from a laptop, even if no network connection is

available. However, you now have two sets of records to protect and the client copies are subject to additional risks, such as theft of the laptop computer. You, therefore, must think about what controls should be used to reduce risk. For example, client records on the laptop may have to be encrypted.

How decisions on development technologies influence security.

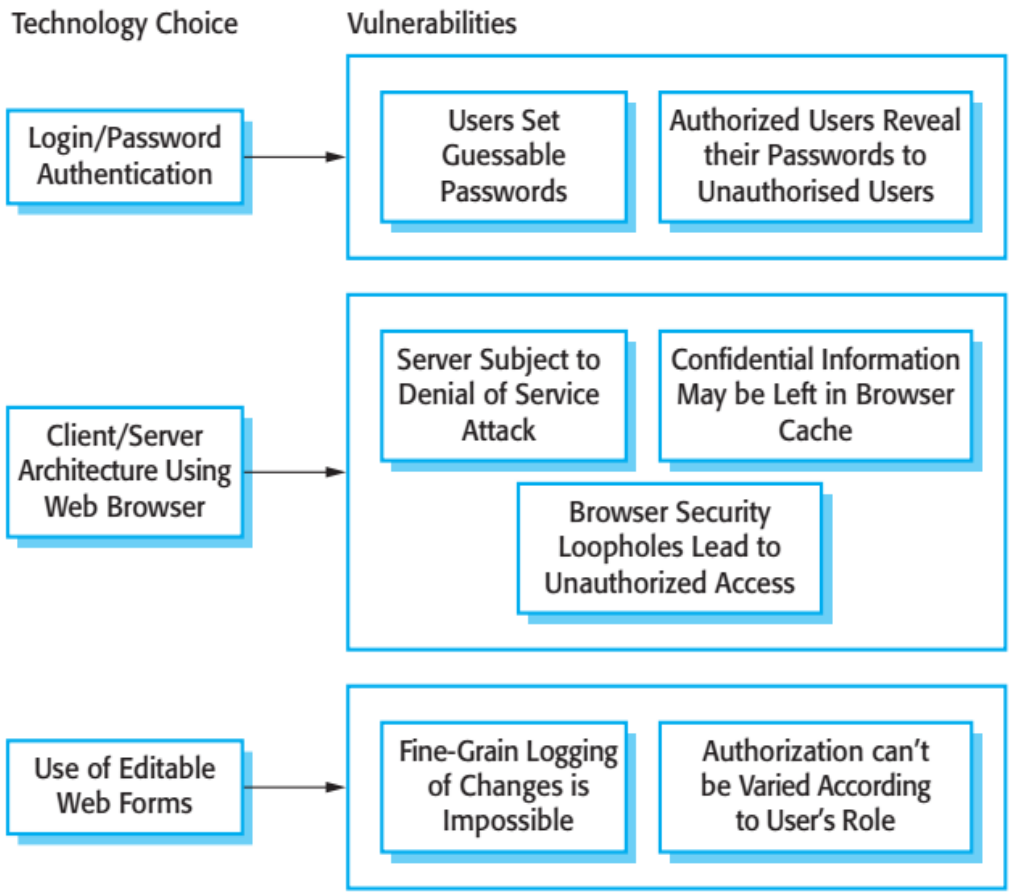
When you develop an application by reusing an existing system, you have to accept the design decisions made by the developers of that system. Let us assume that some of these design decisions are as follows:

1. System users are authenticated using a login name/password combination. No other authentication method is supported.

2. The system architecture is client-server, with clients accessing data through a standard web browser on a client PC.

3. Information is presented to users as an editable web form. They can change information in place and upload the revised information to the server.

Vulnerabilities associated with technology choices



For generic system, these design decisions are perfectly acceptable, but a life-cycle risk analysis reveals that they have associated vulnerabilities. Process vulnerabilities are shown in above diagram (**Vulnerabilities associated with technology choices)**

additional system security requirements or the operation process of using the system. but some examples of requirements are as follows:

1. A password checker program shall be made available and shall be run daily User passwords that appear in the system dictionary shall be identified and user with weak passwords reported to system administrators.
2. Access to the system shall only be allowed to client computers that have been approved and registered with the system administrators.
3. All client computers shall have a single web browser installed as approved by system administrators.

18.2.2 Operational risk assessment

Security risk assessment should continue throughout the lifetime of the system to identify emerging risks and system changes that may be required to cope with these risks.

This process is called operational risk assessment. New risks may emerge because of changing system requirements, changes in the system infrastructure, or changes in the environment in which the system is used.

The process of operational risk assessment is like the life-cycle risk assessment process, but with the addition of further information about the environment in which the system is used. The environment is important because characteristics of the environment can lead to new risks to the system.

For example, say a system is being used in an environment in which users are frequently interrupted. A risk is that the interruption will mean that the user must leave their computer unattended.

It may then be possible for an unauthorized person to gain access to the information in the system. This could then generate a requirement for a password-protected screen saver to be run after a short period of inactivity.

18.3 Design for security

Create the design such a way, which help to secure your system properly.

There are several general, application-independent issues relevant secure systems design:

1. **Architectural design-**how do architectural design decisions affect the security of a system?
2. **Good practice-**what is accepted good practice when designing secure system
3. **Design for deployment-**what support should be designed into systems to avoid the introduction of vulnerabilities when a system is deployed for use?

18.3.1 Architectural Design

The choice of software architecture can have profound effects on the emergent properties of a system.

If an inappropriate architecture is used, it may be very difficult to maintain the confidentiality and integrity of information in the system or to guarantee a required level of system availability.

In designing a system architecture that maintains security, you need to consider two fundamental issues:

1. Protection how should the system be organized so that critical assets can be protected against external attack?
2. Distribution-how should system assets be distributed so that the effects of a successful attack are minimized?

Using the system, which is distributed at different location, may increase risk if intruder gets access to any one of the systems, he can do the changes in other system as well.

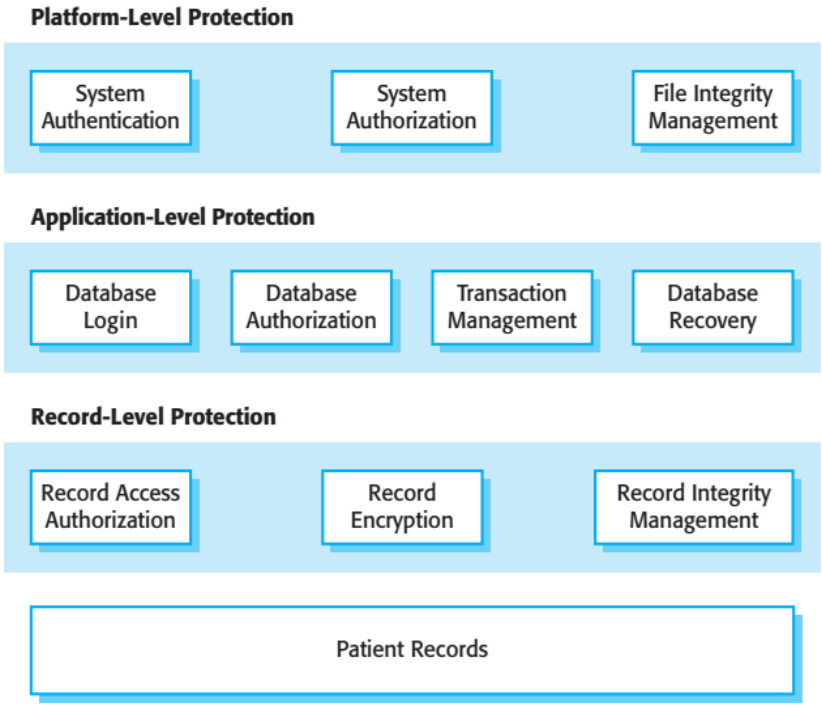


Fig Layered protection architecture

illustrates this for the patient record system in which the critical assets to be protected are the records of individual patients.

In order to access and modify patient records, an attacker has to penetrate three system layers:

1. **Platform-level protection**: The top-level controls access to the platform on which the patient record system runs. This usually involves a user signing on to a particular computer. The platform will also normally include support for maintaining the integrity of files on the system, backups, etc.

2. **Application-level protection**: The next protection level is built into the application itself. It involves a user accessing the application, being authenticated, and getting authorization to take actions such as viewing or modifying data. Application-specific integrity management support may be available.

3. **Record-level protection**: This level is invoked when access to specific records is required and involves checking that a user is authorized to carry out the requested operations on that record. Protection at this level might also involve

- The number of protection layers that you need in any application depend on the criticality of the data.

- To achieve security, you should not allow the same user credentials to be used at each lev Ideally, if you have a password-based system, then the application password shou be different from both the system password and the record-level password.

- If protection of data is a critical requirement, then a client-server architects should be used, with the protection mechanisms built into the server.

- If denial of service attacks is a major risk, you may decide to a distributed object architecture for the application.

Distributed assets in an equity trading system

If the system's assets are distributed across several different p forms, with separate protection mechanisms used for each of these. An attack on node might mean that some assets are unavailable, but it would still be possible to provide some system services.

Data can be replicated across the nodes in the system so that recovery from attacks is simplified. Below figure shows the architecture of a banking system for trading in stocks

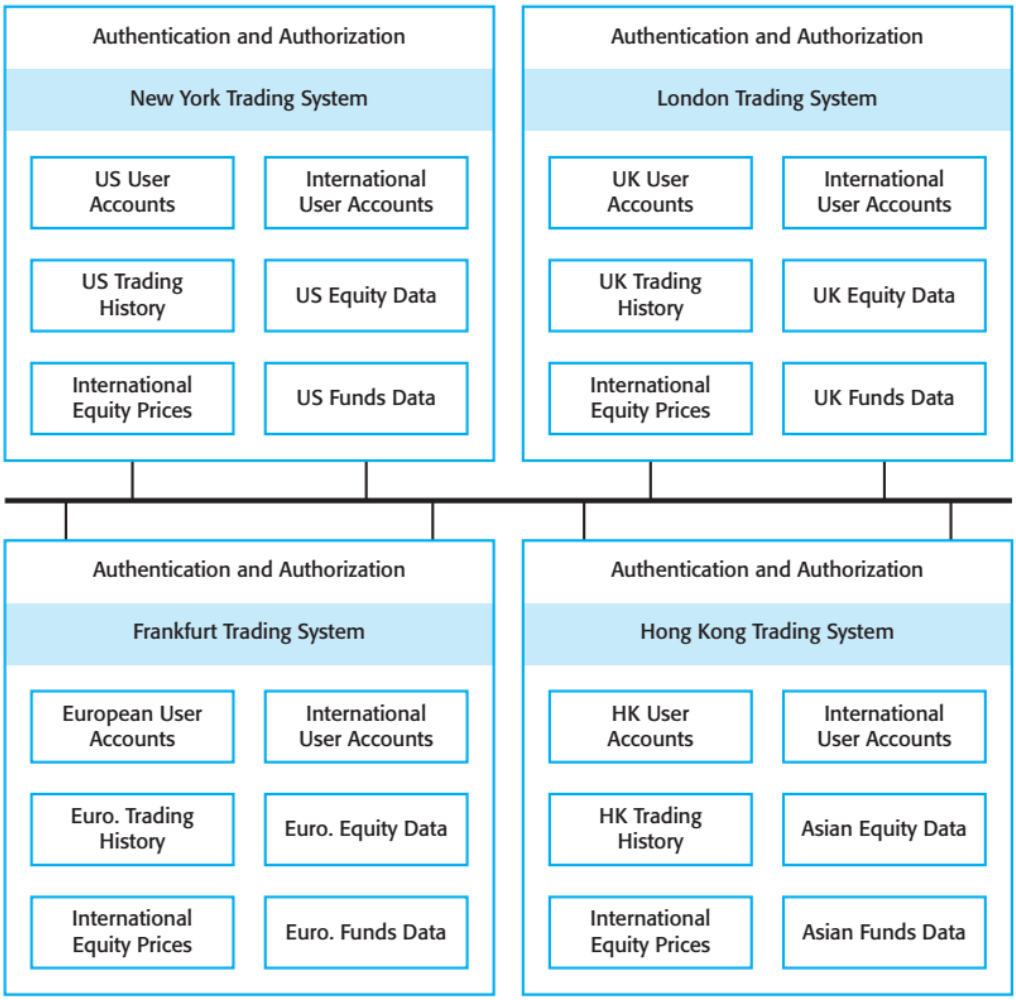


Fig: Distributed assets in an equity trading system

18.3.2 Design Guidelines

Guideline 1: best security decisions on an explicit security policy.

Guideline 2: avoid a single point of failure.

Guideline 3: fail securely

Guideline 4: balance security and usability

Guideline 5: log user actions

Guideline 6: use redundancy and diversity to reduce risk

Guideline 7: validate all inputs

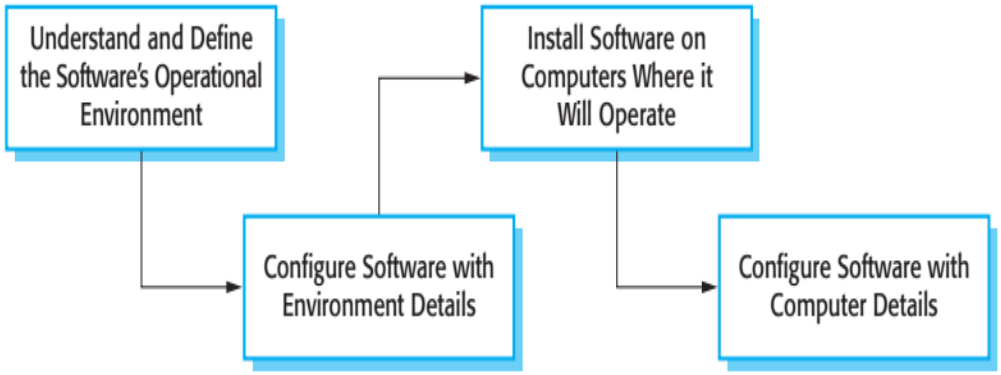
Guideline 8: compartmentalize your assets

Guideline 9: design for development

Guideline 10: design for recoverability

18.3.3 Design for development

The deployment of a system involves configuring the software to operate in an operational environment, installing the system on the computers in that environment, and then configuring the installed system for these computers.



four ways to incorporate deployment support in a system:

1. **Include support for viewing and analysing configurations**

   You should always include facilities in a system that allow administrators or permitted users to examine the current configuration of the system. This facility is, surprisingly, lacking from most software systems and users are frustrated by the difficulties of finding configuration settings. For example, in the version of the word processor that I used to write this chapter, it is impossible to see or print the settings of all system preference on single screen.

2. **Minimize default privileges**

   You should design software so that the default con figuration of a system provides minimum essential privileges. This way, the damage that any attacker can do can be limited. For example, the default system administrator authentication should only allow access to a program that enables an administrator to set up new credentials. It should not allow access to any other

system facilities. Once the new credentials have been set up, the default login and password should be deleted automatically.

3. Localize configuration settings

When designing system configuration support. you should ensure that everything in a configuration that affects the same part of a system is set up in the same place. To use the word processor example again, in the version that I use, I can set up some security information, such as a password to control access to the document, using the Preferences/Security menu. Other information is set up in the Tools/Protect Document menu. If configuration information is not localized, it is easy to forget to set it up or, in some cases, not even be aware that some security facilities are included in the system.

4. Provide easy ways to fix security vulnerabilities

You should include straight forward mechanisms for updating the system to repair security vulnerabilities that have been discovered. These could include automatic checking for security updates or downloading of these updates as soon as they are available. It is important that users cannot bypass these mechanisms as, inevitably, they will consider other work to be more important. There are several recorded examples of major security problems that arose (e.g., complete failure of a hospital network) because users did not update their software when asked to do so.

18.4 System Survivability

1. Survivability or resilience is an emergent property of a system as a whole, rather than a property of individual components, which may not themselves be survivable.

2. The survivability of a system reflects its ability to continue to deliver essential business or mission-critical services to legitimate users while it is under attack or after part of the system has been damaged.

Maintaining the availability of critical services is the essence of survivability. This means that you must know:

- the system services that are the most critical for a business:
- the minimal quality of service that must be maintained.
- how these services might be compromised:

- how these services can be protected:
- how you can recover quickly if the services become unavailable.

A method of analysis called Survivable Systems Analysis. This is used to assess vulnerabilities in systems and to support the design of system architectures and features that promote system survivability.

They argue that achieving survivability depends on three complementary strategies:

1. **Resistance** Avoiding problems by building capabilities into the system to repel attacks. For example, a system may use digital certificates to authenticate users, thus making it more difficult for unauthorized users to gain access.
2. **Recognition** Detecting problems by building capabilities into the system to detect attacks and failures and assess the resultant damage. For example, checksums may be associated with critical data so that corruptions to that data can be detected.
3. **Recovery** Tolerating problems by building capabilities into the system to deliver essential services while under attack, and to recover full functionality after an attack

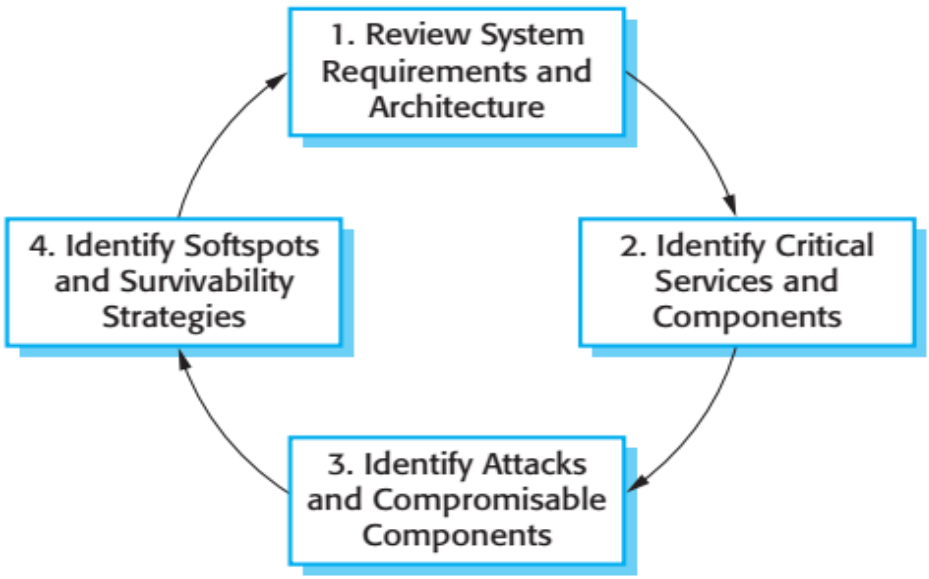


Fig: Stages in survivability analysis

Survivable systems analysis is a four-stage process that analyses the current or proposed system requirements and architecture; identifies critical services, attack scenarios, and system soft spots'; and proposes changes to improve the survivability of a system. The key activities in each of these stages are as follows:

1. **System understanding**:

   For an existing or proposed system, review the goals of the system (sometimes called the mission objectives), the system requirements, and the system architecture.

2. **Critical service identification**

   The services that must always be maintained and the components that are required to maintain these services are identified.

3. **Attack simulation**

   Scenarios or use cases for possible attacks are identified along with the system components that would be affected by these attacks.

4. **Survivability analysis** Components that are both essential and comprisable by an attack are identified and survivability strategies based on resistance. Recognition, and recovery are identified.

Graded Question

1. Explain the important difference between application security engineering and infrastructure security engineering.
2. Explain why there is a need for risk assessment.
3. What is social engineering? Why it is difficult to protect against in it large organization?
4. Explain stages in survivability analysis.
5. Write a short note on operational risk assessment.

Reference Books:

1. Software Engineering, edition, Ian Somerville Peaeson Education. Ninth Edtition

2. Software Engineering, Pankaj Jalote Narosa Publication

3. Software engineering, a practitioner's approach , Roger Pressman , Tata Mcgraw-hill , Seventh