# SOFTWARE ENGINEERING

**MASTER OF COMPUTER APPLICATIONS (MCA)**

**FIRST YEAR, SEMESTER-II, PAPER-I**

**LESSON WRITERS**

Dr. Neelima Guntupalli
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Kampa Lavanya
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

Dr. Vasantha Rudrarnalla
Faculty, Department of CS&E
Acharya Nagarjuna University

Dr. U. Surya Kameswari
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

**EDITOR**
**Dr. Kampa Lavanya**
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

**Academic Advisor**
**Dr. Kampa Lavanya**
Assistant Professor
Department of CS&E
Acharya Nagarjuna University

**Director, I/c**
**Prof. V. VENKATESWARLU**
*MA.,M.P.S., M.S.W.,M.Phil., Ph.D.*

**CENTRE FOR DISTANCEEDUCATION**
**ACHARAYA NAGARJUNA UNIVERSITY**
**NAGARJUNANAGAR – 522510**
***Ph:0863-2346222,2346208,***
***0863-2346259(Study Material)***
***Website:***www.anucde.info
***e-mail:anucdedirector@gmail.com***

# M.C.A : SOFTWARE ENGINEERING

**First Edition : 2025**

**No. of Copies :**

**Published by:**

**Prof. V. VENKATESWARLU**
**Director, I/c**
**Centre for Distance Education,**
**Acharya Nagarjuna University**

***Printed at:***

# *FOREWORD*

***Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining 'A+' grade from the NAAC in the year 2024, Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from over 221 affiliated colleges spread over the two districts of Guntur and Prakasam.***

***The University has also started the Centre for Distance Education in 2003-04 with the aim of taking higher education to the door step of all the sectors of the society. The centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even to housewives desirous of pursuing higher studies. Acharya Nagarjuna University has started offering B.Sc., B.A., B.B.A., and B.Com courses at the Degree level and M.A., M.Com., M.Sc., M.B.A., and L.L.M., courses at the PG level from the academic year 2003-2004 onwards.***

***To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise in the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.***

***It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn be part of country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will go from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic Coordinators, Editors and Lesson-writers of the Centre who have helped in these endeavors.***

***Prof. K. Gangadhara Rao***
*M.Tech., Ph.D.,*
***Vice-Chancellor I/c***
***Acharya Nagarjuna University.***

**MASTER OF COMPUTER APPLICATIONS (MCA)**

# First Year, Semester-II, Paper-I

**201MC24: SOFTWARE ENGINEERING**

**SYLLABUS**

**Unit-l:**

**Introduction to Software Engineering:** The Evolving Role of Software, Software, The Changing Nature of Softwaare, Legacy Software: The Quality of legacy software, Software Evolution, Software Myths.

**A Generic View of Process:** Software Engineering-A Layered Technology, A process Frame Work, The capability Maturity Model Integration (CMMI), process patterns, process Assessment, Personal and Team Process Models: Personal Software process (psp), Team Software Process (TSP), process Technology, product and process.

**Process Models:** Prescriptive Models, The Waterfall Model, Incremental process Models: The Incremental Model, The RAD Model, Evolutionary Process Model: prototyping, The Spiral Model, The Concurrent Development Model, Specialized Process Models: Component Based Development, The formal Methods Moder, The Unified process.

**An Agile view of Process:** what is Agility? What is Agile process? Agile process Models: Extreme Programming, Adaptive Software Development, Dynamic Systems Development Method, scrum, crystal, Feature Driven Development, Agile Modeling.

**Unit-II**

**Software Engineering Practice:** Software Engineering Practice, communication practices, Planning Practices, Modeling practices, construction practices, and Deployment.

**System Engineering:** Computer Based Systems, the System Engineering Hierarchy, Business Process Engineering: An Overview, System Modeling.

**Building the Analysis Model:** Requirement Analysis, Analysis Modeling Approaches, Data Modeling concepts, object oriented Analysis, Scenario Based Modeling, Flow oriented Modeling, class Based Modeling, creating a Behavioural Moder.

**Design Engineering:** Design within the context of Software Engineering, Design process and Design Quality, Design Concepts, The Design Model, pattern Based Software Design.

**Unit-III**

**Testing strategies:** A strategic Approach to software Testing, strategic Issues, and Test Strategies for conventional software, Testing Strategies for object oriented Software, validation Testing, System Testing, the Art of Debugging.

**Testing Tactics:** Software Testing Fundamentals, Black Box and White Box Testing, White Box Testing, Basis Path Testing, Control Structure Testing, Black Box Testing, Object

Oriented Testing Methods, Testing Methods Applicable at the class level, Inter Class Test Case Design, Testing for Specialized Environments Architectures and Applications, Testing Patterns.

**Project Management**: The Management Spectrum, the People, The Product, The Process, The Project, The W5HH Principles.

**Metrics for Process and Projects:** Metrics in the Process and Project Domains, Software Measurement, Metrics for Software Quality, Integrating Metrics within Software Process, Metrics for Small Organizations, Establishing a Software Metrics Program.

**Unit-IV**

**Estimation:** Observations on Estimations, The project planning process, Software Scope and Feasibility, Resources, Software Project Estimation, Decomposition Techniques, Empirical Estimation Models, Estimations for Object Oriented Projects, Specialized Estimation Techniques, The Make/Buy Decision

**Quality Management:** Quality Concepts, Software Quality Assurance, Software Reviews, Formal Technical Reviews, Formal Approaches to SQA, Statistical Software Quality Assurance, Software Reliability, The ISO 9000 Quality standards, the SQA Plan

**Formal Methods:** Basic Concepts, Object Constraint Language (OCL), The Z specification language, The Ten Commandments for Formal Methods'

**Cleanroom Software Engineering:** The Cleanroom Approach, Functional Specification, Cleanroom Design, Cleanroom Testing.

**Prescribed Book:**

Roger S pressman, "software Engineering-A Practitioner's Approach", Sixth Edition, TMH International.

**Reference Books:**

1. Sommerville, "Software Engineering", Seventh Edition Pearson Education (2007 )
2. S.A.Kelkar, 'osoftware Engineering - A Concise Study" . PHI.
3. Waman S.Jawadekar, "Software Engineering", TMH'
4. Ali Behforooz and Frederick J.Hudson, "software Engineering Fundamentals", Oxford (2008)

**(201MC24)**

# M.C.A. DEGREE EXAMINATION, MODEL QUESTION PAPER

**First Year, Second Semester**

**201MC24 - SOFTWARE ENGINEERING**

**Time:** 3 Hours **Max. Marks: 70**

---

**SECTION-A**

**Answer Question No.1 Compulsory** **2 Marks × 7 = 14 Marks**

**1.** a) Define Software Engineering.

b) What is CMMI?

c) Define Business Process Engineering.

d) What is Data Flow Diagram (DFD)?

e) What is Validation Testing?

f) Define Software Quality.

g) What is Cleanroom Testing?

**SECTION-B**

**Answer ONE Question from Each Unit** **4 × 14 = 56 Marks**

**UNIT – I**

**2.** a) Explain the evolving role of software in modern society and discuss the characteristics of good software.

b) Describe the Prescriptive Process Models and explain the Incremental and Spiral Models with neat diagrams.

**OR**

a) Discuss the Capability Maturity Model Integration (CMMI) framework and explain its maturity levels.

b) Explain the Agile Process Models — Extreme Programming (XP) and Feature-Driven Development (FDD).

**UNIT – II**

**3.** a) Explain the System Engineering Hierarchy and describe the major components of a computer-based system.

b) Describe the steps involved in Requirement Engineering and the structure of a good SRS document.

**OR**

a) Discuss the importance of Design Concepts — abstraction, refinement, modularity, and hierarchy.

b) Explain Design Quality Guidelines and discuss how software design is validated for correctness.

**UNIT – III**

**4.** a) Explain the Strategic Approach to Software Testing and describe various levels of testing.
b) Discuss the W$^6$HH Principle proposed by Barry Boehm for project management and control.

**OR**

a) Describe White-Box and Black-Box Testing Techniques with suitable examples.
b) Explain Process and Project Metrics and their role in improving software quality.

**UNIT – IV**

**5.** a) Discuss the Decomposition Techniques and provide an example of LOC-based estimation.
b) Explain Software Quality Assurance (SQA) Activities and the role of Formal Technical Reviews (FTRs).

**OR**

a) Explain Formal Methods and discuss the Object Constraint Language (OCL) with an example.
b) Describe the Cleanroom Software Engineering Process and explain the Box Structure Methodology.

# CONTENTS

# LESSON- 01
# INTRODUCTION TO SOFTWARE

**AIMS AND OBJECTIVES**

The primary goal of this chapter is to understand the software engineering principles. The chapter began with understanding what is software engineering, importance of software engineering and so on. After completing this chapter, the student will understand the complete knowledge about software engineering principles.

**STRUCTURE**

## 1.1. INTRODUCTION

A software engineer studies, designs, develops, maintains, and eventually phases out software, making it super important in nearly every organization. The importance of software engineering extends beyond large IT companies and MNCs, it impacts daily life by providing numerous benefits. Basically, Software engineering was introduced to address the issues of low-quality software projects. Here, the development of the software uses the well-defined scientific principal method and procedure. In other words, software engineering is a process in which the needs of users are analyzed and then the software is designed as per the requirement of the user. Software engineering builds this software and application by using designing and programming language. This chapter explores what is software engineering, the importance of software engineering, and principles of software engineering.

## 1.2 WHAT IS SOFTWARE ENGINEERING

What is Software Engineering? The term software engineering is the product of two words, software, and engineering. The software is a collection of integrated programs.

- Software subsists of carefully organized instructions and code written by developers on any of various computer languages.

- Computer programs and related documentation such as requirements, design models and user manuals.
  Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc.

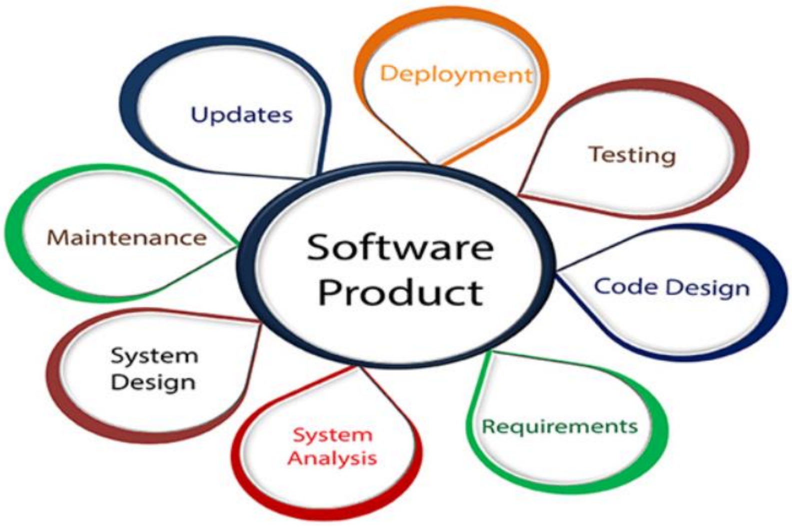


**Fig 1.1. Software Engineering**

Software Engineering is an engineering branch related to the evolution of software product using well-defined scientific principles, techniques, and procedures. The result of software engineering is an effective and reliable software product.

**Purpose of Software Engineering**

- To produce reliable and efficient software that meets user requirements.
- To manage complexity in large software projects through structured processes.
- To ensure quality, maintainability, and scalability throughout the software's lifecycle.
- To reduce cost and time of development through reusability and automation.

Software Engineering is a crucial branch of engineering that provides the foundation for building effective, dependable, and maintainable software systems. It combines creative design with disciplined methodologies to ensure that software products evolve efficiently and serve user needs over time.

## 1.3. THE EVOLVING ROLE OF SOFTWARE

Today, software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware. Whether it resides within a cellular phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information that can be as simple as a single bit or as complex as a multimedia presentation.

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

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

The role of computer software has undergone significant change over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems. The lone programmer of an earlier era has been replaced by a team of software specialists, each focusing on one part of the technology required to deliver a complex application.

## 1.4. CHANGING NATURE OF SOFTWARE

The nature of software has changed a lot over the years.

- **System Software:** Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically, system software is a collection of programs to provide service to other programs.

- **Real time Software:** These software is used to monitor, control and analyse real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather.

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

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

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

- **Artificial intelligence Software:** Artificial Intelligence software makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Examples are expert systems, artificial neural network, signal processing software etc.

- **Web based Software:** The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

| **Type of Software** | **Description** | **Examples / Applications** |
|---|---|---|
| System Software | Provides the infrastructure for other software; manages hardware and system resources. | Operating Systems, Compilers, Device Drivers, Editors |
| Real-Time Software | Monitors, controls, and responds to real-world events as they occur; often time-critical. | Weather Forecasting, Air Traffic Control, Process Control Systems |
| Embedded Software | Stored in ROM and controls hardware-specific functions of products or devices; often called intelligent software. | Aircraft Systems, Automobiles, Security Systems, Power Plant Controllers |
| Business Software | Designed for business applications and data management; supports organizational processes and decision-making. | Payroll Systems, Accounting Software, ERP, MIS, Data Warehousing Tools |
| Personal Computer Software | Applications designed for individual users on personal computers; focuses on productivity, entertainment, and creativity. | Word Processors, Graphics Tools, Multimedia Applications, Games, Databases |
| Artificial Intelligence (AI) Software | Uses algorithms that mimic human reasoning and learning to solve complex, non-numerical problems. | Expert Systems, Neural Networks, Signal Processing, Chatbots |
| Web-Based Software | Applications that run over the web or use internet technologies for functionality and user interaction. | HTML, Java, CGI, DHTML, Web Portals, E-commerce Sites |

## 1.5. LEGACY SOFTWARE

Legacy software is software that has been around a long time and still fulfils a business need. It is mission critical and tied to a particular version of an operating system or hardware model (vendor lock-in) that has gone end-of-life. Generally, the lifespan of the
hardware is shorter than that of the software.
Common Quality Issues in Legacy Software

- **Maintenance Challenges**

Legacy software often lacks modern features and may be built on outdated technologies, making it difficult to update and maintain. The original developers might no longer be available, and the documentation may be insufficient or missing, complicating the maintenance process.

- **Compatibility Issues**
  Legacy software might not be compatible with newer hardware, operating systems, or other software applications. This incompatibility can limit the integration of new technologies and systems, reducing overall efficiency and productivity.

- **Security Vulnerabilities**
  Older software often lacks the robust security features found in modern applications. This can make legacy systems vulnerable to cyber-attacks and security breaches, posing significant risks to the organization.

- **Performance Problems**
  Legacy systems may suffer from performance issues due to outdated code and inefficient algorithms. These problems can lead to slow processing times, increased downtime, and overall reduced system performance.

- **Lack of Documentation**
  Over time, documentation for legacy software may become outdated or lost. Without proper documentation, understanding and modifying the system becomes challenging, leading to potential errors and inefficiencies.

- **Technical Debt**
  Legacy software often accumulates technical debt, which refers to the extra work required to fix issues that arise when code that is easy to implement in the short term is used instead of applying the best overall solution. This debt can hinder the software's ability to evolve and adapt to new requirements.

## 1.6. SOFTWARE EVOLUTION

The process of developing a software product using software engineering principles and methods is referred to as software evolution. This includes the initial development of software and its maintenance and updates, till the desired software product is developed, which satisfies the expected requirements.

Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

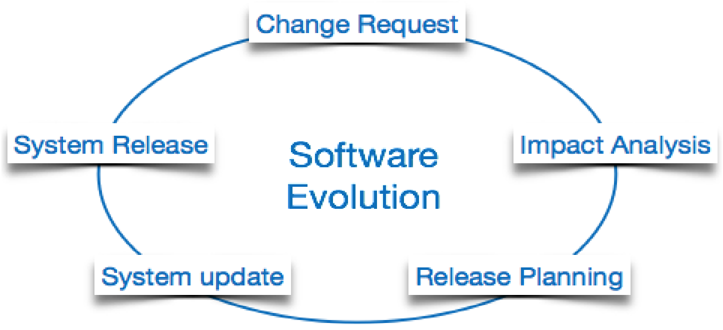


Figure.1.2. Software Evolution

Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and going one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

**Need for Software Evolution**

Software evolution is necessary due to several factors:

1. **Changing User Requirements:** As organizations grow and markets evolve, users' needs and expectations change.
2. **Technological Advancements:** New hardware, operating systems, databases, and frameworks emerge, requiring software adaptation.
3. **Bug Fixes and Maintenance:** Errors discovered after deployment must be corrected to maintain reliability.
4. **Performance Improvement:** To enhance speed, scalability, or resource utilization.
5. **Legal and Regulatory Compliance:** Updates may be required to meet new standards or laws.
6. **Business Competition:** Continuous enhancement keeps software competitive in the market.

**Lehman's Laws of Software Evolution**

Meir M. Lehman formulated a series of empirical observations known as the **"Laws of Software Evolution"**, which describe the behavior of evolving systems, particularly large and long-lived ones.

1. **Continuing Change:** Software must continually adapt or it becomes progressively less useful.
2. **Increasing Complexity:** As software evolves, its complexity increases unless work is done to reduce it.
3. **Self-Regulation:** Software evolution processes are self-regulating with measurable attributes.
4. **Conservation of Organizational Stability:** The rate of change remains roughly constant over time.
5. **Conservation of Familiarity:** Developers must maintain familiarity with the system — excessive change can make it unmanageable.
6. **Continuing Growth:** Functionality must be continually increased to satisfy user needs.
7. **Declining Quality:** Unless rigorously maintained, quality will decline over time.
8. **Feedback System:** Evolution processes involve feedback loops and must be treated as such to be properly managed.

**Types of Software Maintenance (Evolution Activities)**

Software evolution primarily occurs through **maintenance activities**, which can be categorized as:

1. **Corrective Maintenance**
   - Fixing discovered defects or errors.
   - Example: Repairing a faulty calculation or correcting a crash issue.
2. **Adaptive Maintenance**
   - Modifying software to accommodate environmental changes (OS updates, new hardware, etc.).

   - Example: Updating software to run on a new version of Windows or Android.
3. **Perfective Maintenance**
   - Enhancing performance, maintainability, or adding new features based on user feedback.
   - Example: Improving user interface responsiveness or adding new reporting tools.
4. **Preventive Maintenance**
   - Making changes to prevent future problems or improve system stability.
   - Example: Refactoring legacy code to simplify future updates.

**Software Re-engineering and Evolution**

Over time, extensive evolution may make software systems **complex and inefficient**. In such cases, **Software Re-engineering** may be required.

Re-engineering involves:

- Analyzing existing systems.
- Reconstructing them using modern design and technology.
- Improving structure and performance without changing core functionality.

This helps extend the software's useful life while maintaining quality and reducing costs.

**Challenges in Software Evolution**

- Managing **increasing complexity** and technical debt.
- Maintaining **consistency** across versions.
- Balancing **new feature development** with **bug fixing**.
- Ensuring **backward compatibility**.
- Handling **changing technologies and platforms**.
- Preserving **documentation and knowledge** as teams change.

**Tools** and Practices Supporting Evolution

- Version Control Systems (e.g., Git, SVN)
- Automated Testing & Continuous Integration (CI/CD)
- Configuration Management Tools
- Refactoring and Code Review Processes
- Change Impact Analysis and Documentation

## 1.7. SOFTWARE MYTHS

Software Myths- beliefs about software and the process used to build it - can be traced to the earliest days of computing. Myths have a number of attributes that have made them insidious. For instance, myths appear to be reasonable statements of fact, they have an intuitive feel, and they are often promulgated by experienced practitioners "who know the score".

**Myth:** Software that works perfectly will never need maintenance

**Reality**: All software requires maintenance due to evolving requirements, environments, and the discovery of bugs. Even the best-designed software must adapt to changes in user needs, operating environments, and technological advancements.

**Myth:** Adding more developers to a late project will speed it up

**Reality:** Adding more developers to a late project often leads to further delays due to increased complexity and communication overhead. This phenomenon is known as Brooks' Law, which states, "Adding manpower to a late software project makes it later."

**Myth:** Once software is written, it’s done

**Reality:** Software development is an ongoing process that involves continuous updates, improvements, and adaptations. Software must evolve to meet changing requirements and environments.

**Myth:** More features make better software

**Reality:** More features do not necessarily make better software. Adding unnecessary features can lead to bloated software, which is harder to use, maintain, and secure. Focusing on core functionalities and usability is often more important.

**Myth:** Software can be completely bug-free
**Reality:** While rigorous testing and quality assurance can minimize bugs, it is virtually impossible to create completely bug-free software. The goal should be to identify and fix critical bugs and continuously improve the software.

**Myth:** Software development is purely technical

**Reality:** Software development is not just a technical activity; it involves significant collaboration, communication, and problem-solving among team members, stakeholders, and users.

**Myth:** Open-source software is less secure than proprietary software

**Reality:** Open-source software can be as secure, if not more secure, than proprietary software. The open-source community actively reviews and improves the code, leading to robust security practices.

| **Myth** | **Reality** |
|---|---|
| **1. Software that works perfectly will never need maintenance** | Even perfectly functioning software requires **maintenance** to address changing user needs, operating environments, security vulnerabilities, and emerging technologies. Software evolution is inevitable. |
| **2. Adding more developers to a late project will speed it up** | This is a well-known misconception described by **Brooks’ Law**: *“Adding manpower to a late software project makes it later.”* More developers increase communication overhead, coordination challenges, and integration issues, often slowing progress. |
| **3. Once software is written, it’s done** | Software development doesn’t end with coding. It is an **ongoing process** involving testing, deployment, maintenance, updates, and adaptation to change. Long-term support is part of the software lifecycle. |
| **4. More features make better software** | Adding unnecessary features can cause **feature bloat**, making the software complex, harder to use, maintain, and secure. Good |

| | |
|---|---|
| | software emphasizes **usability, simplicity, and performance**, not the sheer number of features. |
| **5. Software can be completely bug-free** | Achieving absolutely bug-free software is practically **impossible**. Even with rigorous testing, some defects may remain. The goal should be to minimize **critical bugs** and continuously improve quality through testing and maintenance. |
| **6. Software development is purely technical** | Software development is as much about **communication, teamwork, and problem-solving** as it is about technical skill. Successful projects depend on collaboration among developers, managers, and stakeholders. |
| **7. Open-source software is less secure than proprietary software** | This is a misconception. **Open-source software** can be equally or even **more secure**, as the community actively reviews, tests, and enhances the code, identifying vulnerabilities faster than closed systems. |

However, myths can distort understanding, lead to poor management decisions, and negatively impact software quality and project success.Recognizing and dispelling these myths is crucial to building realistic expectations among developers, managers, and users.

## 1.8 SUMMARY

This chapter begins with an introduction to the fundamental concept of Software Engineering, outlining its definition, objectives, and the need for a systematic approach to software development. It then explores several important introductory concepts that establish the foundation for understanding discipline in depth. The discussion moves on to the Evolving Role of Software, highlighting how software has become an integral part of modern society, influencing every domain — from communication and transportation to healthcare and education. The section on Software itself explains what software is, its essential characteristics, and how it differs from hardware. Next, the Changing Nature of Software is examined, emphasizing the continuous growth in complexity, diversity of applications, and demand for adaptability. The chapter also introduces the concept of Legacy Software, focusing on its definition, challenges, and the importance of maintaining and improving the quality of legacy systems to meet current and future needs. A detailed explanation of Software Evolution follows, describing how software must adapt and grow over time to remain useful and relevant in changing environments. Finally, the chapter concludes with a discussion on Software Myths, addressing common misconceptions held by managers, developers, and users regarding software development and maintenance. Overall, this chapter provides a comprehensive foundation for understanding the nature of software and its engineering discipline, setting the stage for more advanced topics in subsequent chapters.

## 1.9 TECHNICAL TERMS

Software Engineering, Reusability, Maintenance, Design, Testing, Mobile Applications, Web Applications, Artificial Intelligence, Machine Learning.

## 1.10 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. Illustrate about software engineering principles.
2. Describe about importance of software engineering.
3. Explain about advantages and disadvantages of software engineering

**Short Notes:**

1. Write is software engineering.
2. Discuss about software engineering principle.
3. List out benefits of software engineering.

## 1.11 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Neelima Guntupalli

# LESSON- 02
# A GENERIC VIEW OF PROCESS

**AIMS AND OBJECTIVES**

The primary goal of this chapter is to understand the generic view of process. The chapter began with Software Engineering-A Layered Technology, A Process Framework, The capability Maturity Model Integration (CMMI), Process Patterns, Process Assessment, Personal and Team Process Models: Personal Software Process (PSP), Team Software Process (TSP), Process Technology, Product and Process. After completing this chapter, the student will understand the complete knowledge about generic view of process.

**STRUCTURE**

## 2.1. INTRODUCTION

Software engineering is structured as a layered technology to manage the complexity and improve the quality of software development. At the base is the quality focus, which permeates all activities. The next layer is the process, which provides the framework and methodologies for managing the development lifecycle. The third layer consists of the methods, offering the technical know-how, such as design techniques and coding practices. At the top are the tools, which support the processes and methods with automation and efficiency. A process framework in software engineering refers to a structured approach that guides the development and maintenance of software products. It encompasses all phases of the software lifecycle, from planning and design to implementation, testing, and maintenance. This framework ensures consistency, repeatability, and quality in software production.
The chapter began with Software Engineering-A Layered Technology, A Process Frame Work, The capability Maturity Model Integration (CMMI), Process Patterns, Process Assessment, Personal and Team Process Models: Personal Software Process (PSP), Team Software Process (TSP), Process Technology, Product and Process.

## 2.2. SOFTWARE ENGINEERING-A LAYERED TECHNOLOGY

Software engineering encompasses a process, the management of activities, technical methods, and use of tools to develop software products

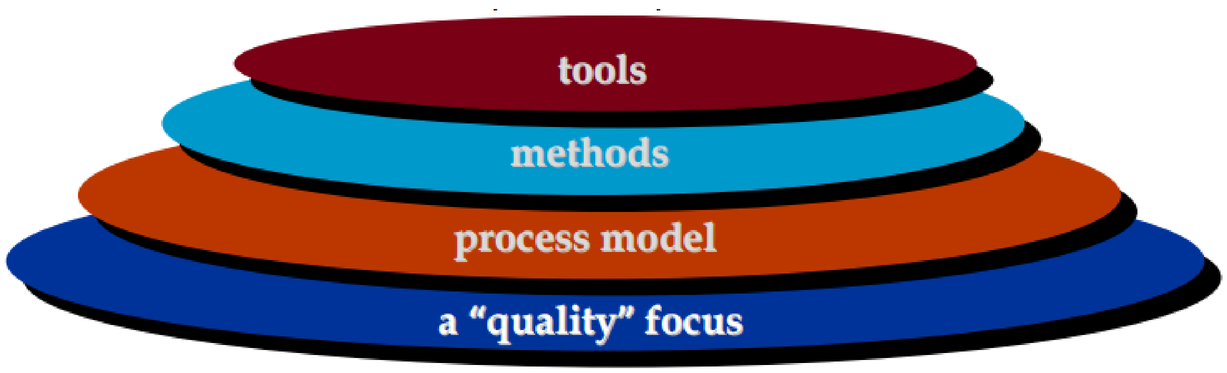


**Fig.2.1. Software Engineering Layered Technology**

### 2.2.1. a generic view of process

- **Process:** A set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, and user manuals).
- The foundation for software engineering is the process layer. It is the glue that holds the technology layers together and enables rational and timely development of computer software.
- Process defines a framework that must be established for effective delivery of software engineering technology.
- The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.
- Software engineering methods provide the technical —'how to' for building software. Methods encompass a broad array of tasks that include communication, req. analysis, design, coding, testing and support.
- Software engineering tools provide automated or semi-automated support for the process and the methods.

### 2.2.2. a process framework

- Establishes the foundation for a complete software process
- Identifies a number of framework activities applicable to all software projects
- Also include a set of umbrella activities that are applicable across the entire software process.
- Used as a basis for the description of process models
- Generic process activities - Communication - Planning - Modeling - Construction – Deployment

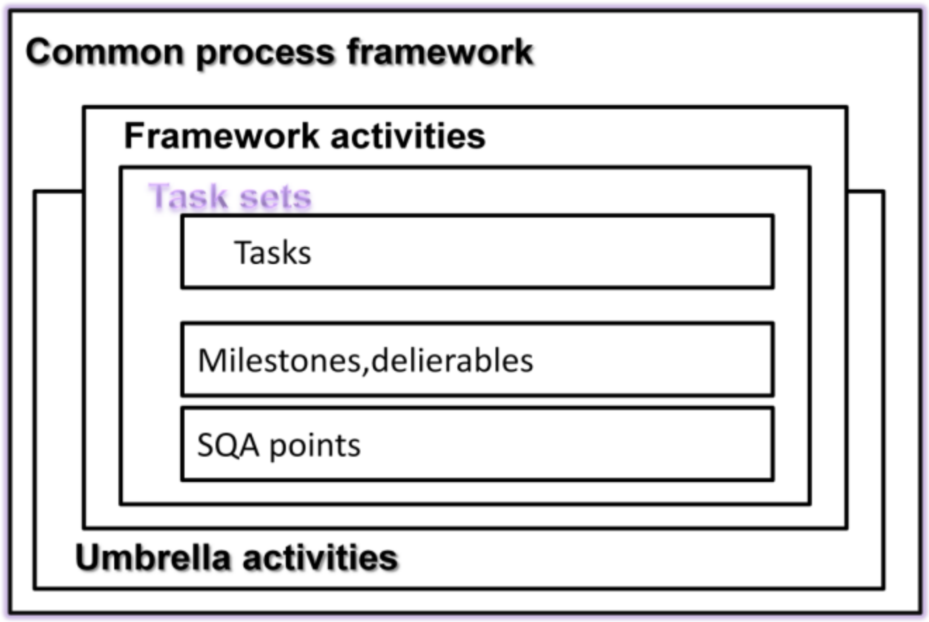


Fig.2.2. Software Engineering  Process Framework

- Communication activity
- Planning activity
- Modeling activity
- analysis action
- requirements gathering work task
- elaboration work task
- negotiation work task
- specification work task
- validation work task
- design action
- data design work task
- architectural design work task
- interface design work task
- component-level design work task
- Construction activity o Deployment activity
- Umbrella activities (examples)
- software project tracking and control
- risk management
- software quality assurance
- formal technical reviews
- measurement
- s/w configuration management
- reusability management
- work product preparation and production (e.g., models, documents, logs)

## 2.2.3. the capability maturity model integration (cmmi)

CMM Levels.

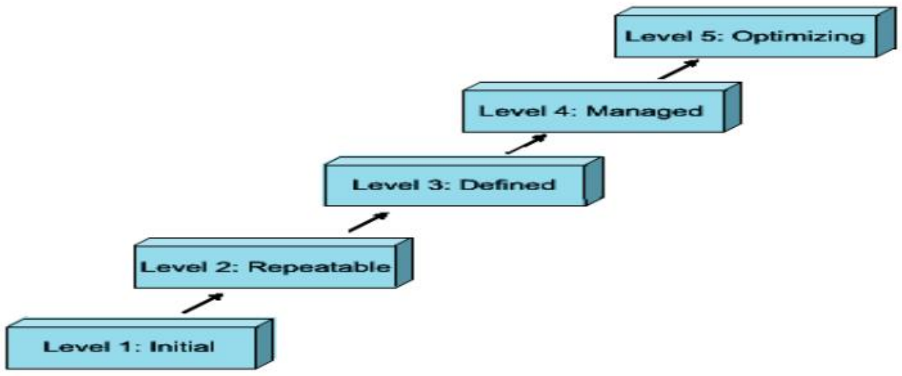


**Fig.2.3. CMM Levels**

**Level 1: Initial**

- A software development organization at this level is characterized by ad hoc activities.
- Very few or no processes are defined and followed.
- Since software production processes are not defined, different engineers follow their own process and as a result development efforts become chaotic.
- The success of projects depends on individual efforts and heroics.
- Since formal project management practices are not followed, under time pressure short cuts are tried out leading to low quality.

**Level 2: Repeatable**

- At this level, the basic project management practices such as tracking cost and scheduling are established.
- Size and cost estimation techniques like function point analysis, COCOMO, etc. are used.
- The necessary process discipline is in place to repeat earlier success on projects with similar applications. Opportunity to repeat a process exists only when a company produces a family of products.

**Level 3: Defined**

- At this level the processes for both management and development activities are defined and documented.
- There is a common organization-wide understanding of activities, roles, and responsibilities.
- The processes though defined, the process and product qualities are not measured.
- ISO 9000 aims to achieve this level.

**Level 4: Managed**

- At this level, the focus is on software metrics.
- Two types of metrics are collected.
- Product metrics measure the characteristics of the product being developed, such as its size, reliability, time complexity, understandability, etc.

- Process metrics reflect the effectiveness of the process being used, such as average defect correction time, productivity, average number of defects found per hour inspection, average number of failures detected during testing per LOC, etc.
- Quantitative quality goals are set for the products. The software process and product quality are measured and quantitative quality requirements for the product are met.
- Various tools like Pareto charts, fishbone diagrams, etc. are used to measure the product and process quality.
- Thus, the results of process measurements are used to evaluate project performance rather than improve the process.

**Level 5: Optimizing**

- Process and product measurement data are analyzed for continuous process improvement.
- The process may be fine-tuned to make the review more effective.
- The lessons learned from specific projects are incorporated into the process.
- Continuous process improvement is achieved both by carefully analyzing the quantitative feedback from the process measurements and also from application of innovative ideas and technologies.
- These best practices are transferred throughout the organization.

## 2.3 PROCESS PATTERNS

Software process patterns are templates that provide solutions to recurring problems within the software development lifecycle. These patterns help streamline processes, improve efficiency, and ensure consistency across projects. Roger S. Pressman describes process patterns using a specific structure comprising several key elements:

1. **Process Template**:

   - **Pattern Name**: A unique identifier for the pattern.
   - **Intent**: The purpose or goal of the pattern.
   - **Types**: Categories of the pattern, such as task patterns, stage patterns, and phase patterns.
   - **Task Pattern**: Describes specific tasks within the software process.
   - **Stage Pattern**: Outlines activities within a particular stage of the software process.
   - **Phase Pattern**: Defines activities within a specific phase of the software lifecycle.

2. **Initial Context**:

   - Describes the conditions under which the pattern is applicable. This includes the state of the project, the development environment, and any prerequisites for applying the pattern.

3. **Problem**:

   - Clearly states the problem that the pattern aims to solve. This involves identifying the challenges or issues commonly faced in a specific context.

4. **Solution**:
- Provides a detailed description of how to implement the pattern to address the identified problem. This includes steps, activities, tools, and techniques required to achieve the desired outcome.

5. **Resulting Context**:
- Describes the state of the project after applying the pattern. This includes the expected improvements, changes in the process, and any new conditions that arise as a result.

6. **Related Patterns**:

- Lists other patterns that are related or can be used in conjunction with the current pattern. This helps in understanding the broader context and potential combinations for effective process improvement.
- **Example of a Process Pattern Template**
- **Pattern Name**
- **Requirement Validation**
- **Intent**
- To ensure that all requirements are complete, consistent, and correct before proceeding to the design phase.
- **Types**
- **Task Pattern**
- **Initial Context**
- The requirements have been gathered and documented, but there is a need to validate them to prevent errors in subsequent phases.
- **Problem**
- Incomplete or incorrect requirements can lead to significant issues during the design and development phases, resulting in increased costs and delays.
- **Solution**
- Conduct a requirements review meeting involving key stakeholders.
- Use checklists to verify completeness and consistency.
- Perform traceability analysis to ensure all requirements are linked to business needs.
- **Resulting Context**
- Requirements are validated, reducing the risk of errors and ensuring that the design phase proceeds with accurate and complete information.
- **Related Patterns**
- **Requirements Elicitation**
- **Requirements Management**

## 2.4 PROCESS ASSESSMENT

Process assessment and improvement are crucial for maintaining and enhancing the quality and efficiency of software development processes. Various standards and methodologies have been developed to facilitate these activities. Here, we discuss several key frameworks and models used for process assessment and improvement in software engineering:

❖ Standard CMMI Assessment Method for Process Improvement (SCAMPI)

The Standard CMMI Assessment Method for Process Improvement (SCAMPI) is a comprehensive methodology used to assess an organization's process maturity based on the

Capability Maturity Model Integration (CMMI) framework. SCAMPI provides a structured approach for evaluating process implementation and effectiveness, identifying strengths and weaknesses, and establishing a basis for continuous improvement. SCAMPI assessments are classified into three classes:

- ❖ **Class A**: Provides the most rigorous assessment, often used for official ratings.
- ❖ **Class B**: Less formal and typically used for internal assessments to identify areas for improvement.
- ❖ **Class C**: The least formal, used for quick evaluations and initial assessments.
- ❖ CMM-Based Appraisal for Internal Process Improvement (CBA IPI)

The CMM-Based Appraisal for Internal Process Improvement (CBA IPI) is another methodology designed for assessing and improving software processes based on the Capability Maturity Model (CMM). It focuses on:

- **Identifying the maturity level** of the current processes.
- **Highlighting process strengths** and areas needing improvement.
- **Providing a roadmap** for achieving higher maturity levels.

CBA IPI involves detailed data collection through interviews, document reviews, and observations to provide a thorough analysis of process performance.

❖ SPICE (ISO/IEC 15504)

SPICE, or Software Process Improvement and Capability Determination, is an international standard (ISO/IEC 15504) for assessing and improving software processes. It provides a framework for:

- **Process assessment**: Evaluating the capability of software processes against a predefined set of criteria.
- **Process improvement**: Identifying and implementing improvements based on assessment results.

SPICE is widely used in various industries to ensure that software processes are efficient, effective, and capable of producing high-quality products.

❖ ISO 9001:2000 for Software

ISO 9001:2000 is part of the ISO 9000 family of standards for quality management systems. When applied to software development, it focuses on:

- **Establishing a quality management system** that meets customer and regulatory requirements.
- **Continuous improvement** of processes through regular audits and reviews.
- **Documenting processes** to ensure consistency and repeatability.

ISO 9001:2000 emphasizes a process-oriented approach, ensuring that software development processes are systematically managed and improved.

**Key Steps in Process Assessment**

Regardless of the specific methodology used, process assessment typically involves the following steps:

1. **Preparation**:
    - Define the scope and objectives of the assessment.
    - Select the assessment team and prepare necessary documentation.

2. **Data Collection**:
   - Gather data through interviews, surveys, and document reviews.
   - Observe process execution to understand current practices.

3. **Analysis**:
   - Evaluate the collected data against predefined criteria or standards.
   - Identify strengths, weaknesses, and areas for improvement.

4. **Reporting**:
   - Document the findings and provide actionable recommendations.
   - Communicate results to stakeholders and develop an improvement plan.

5. **Implementation**:
   - Implement the recommended improvements.
   - Monitor progress and adjust the plan as needed.

By following these steps, organizations can systematically assess and improve their software development processes, leading to higher efficiency, better quality products, and increased customer satisfaction.

## 2.5 PERSONAL AND TEAM PROCESS MODELS

**Personal and Team Process Models**
**Personal Software Process (PSP)**
The Personal Software Process (PSP) was developed by Watts S. Humphrey to help individual software engineers improve their work. PSP is designed to provide engineers with a disciplined approach to managing their work and improving the quality of the software they produce.
**Key Components of PSP**:

1. **Planning**:
   - **Size Estimation**: Estimating the size of the work product.
   - **Effort Estimation**: Estimating the time required for each task.
   - **Task Planning**: Creating a detailed plan for the project.

2. **High-Level Design**:
   - **Design Reviews**: Reviewing design documents to ensure completeness and correctness.
   - **Prototyping**: Building prototypes to understand system requirements and design options.

3. **Development**:
   - **Coding**: Writing the code according to the design specifications.
   - **Code Reviews**: Performing peer reviews to identify defects early.

4. **Testing**:
   - **Unit Testing**: Testing individual units of code to ensure they function correctly.
   - **Integration Testing**: Ensuring that integrated components work together.

5. **Postmortem Analysis**:
   - **Defect Analysis**: Analyzing defects to identify root causes.
   - **Process Improvement**: Using lessons learned to improve future processes.

**Benefits of PSP**:
- Enhanced estimation accuracy.
- Improved schedule adherence.
- Reduced defect rates.
- Increased personal accountability and productivity.

**Team Software Process (TSP)**

The Team Software Process (TSP), also created by Watts S. Humphrey, extends the principles of PSP to team environments. TSP provides a framework for teams to manage their projects collectively, aiming to produce high-quality software on time and within budget.

**Key Components of TSP**:
1. **Team Formation and Launch**:
   - **Role Assignments**: Assigning roles such as team leader, development manager, and quality manager.
   - **Goal Setting**: Establishing common goals and objectives for the team.

2. **Planning**:
   - **Detailed Project Planning**: Developing a comprehensive plan that includes task assignments, schedules, and resource allocation.
   - **Risk Management**: Identifying and mitigating potential risks.

3. **Process Implementation**:
   - **Process Definition**: Defining and adhering to a specific software process.
   - **Process Improvement**: Continuously refining the process based on team feedback and performance data.

4. **Quality Management**:
   - **Defect Prevention**: Implementing practices to prevent defects from occurring.
   - **Quality Assurance**: Conducting reviews and tests to ensure high quality.

5. **Project Monitoring and Control**:
   - **Tracking Progress**: Regularly monitoring project progress against the plan.
   - **Issue Resolution**: Addressing issues promptly to keep the project on track.

**Benefits of TSP**:
- Improved team collaboration and communication.
- Higher productivity and efficiency.
- Enhanced product quality.
- Better project predictability and control.

**Integration of PSP and TSP**

PSP and TSP are complementary models that can be integrated to improve both individual and team performance. While PSP focuses on personal discipline and improvement, TSP builds on these principles to enhance team dynamics and project management. By implementing PSP, team members develop strong personal practices that contribute to the overall effectiveness of the team when working within the TSP framework.

## 2.6 PROCESS TECHNOLOGY

Process technology in software engineering refers to the tools, techniques, and methodologies used to support and enhance software processes. These technologies enable the efficient execution, management, and improvement of software development activities, contributing to higher productivity and better-quality products.
Here are some key components and concepts related to process technology:

1. **Software Process Automation Tools**
   - **Integrated Development Environments (IDEs)**: Tools like Eclipse, Visual Studio, and IntelliJ IDEA provide comprehensive environments that support coding, debugging, and testing.
   - **Version Control Systems**: Tools like Git, SVN, and Mercurial help manage changes to source code over time, enabling collaboration and maintaining a history of modifications.
   - **Build Automation Tools**: Tools such as Maven, Gradle, and Ant automate the compilation, testing, and deployment of software, ensuring consistent builds and reducing manual errors.

2. **Continuous Integration (CI) and Continuous Deployment (CD)**
   - **CI Tools**: Jenkins, Travis CI, and CircleCI automatically integrate code changes from multiple contributors, run tests, and detect issues early in the development cycle.
   - **CD Tools**: Tools like AWS CodePipeline, GitLab CI, and Bamboo automate the deployment of software to production environments, ensuring that new features and fixes are delivered rapidly and reliably.

3. **Project Management and Collaboration Tools**
   - **Agile Project Management**: Tools like JIRA, Trello, and Asana support agile methodologies by enabling sprint planning, task tracking, and collaboration among team members.
   - **Communication Platforms**: Slack, Microsoft Teams, and Zoom facilitate real-time communication and collaboration, enhancing team coordination and decision-making.

4. **Quality Assurance and Testing Tools**
   - **Automated Testing**: Tools such as Selenium, JUnit, and TestNG enable the automation of unit, integration, and functional testing, ensuring that software meets quality standards.
   - **Code Analysis**: Tools like SonarQube and Checkmarx perform static code analysis to detect code smells, security vulnerabilities, and adherence to coding standards.

5. **Process Improvement and Monitoring Tools**
   - **Process Mining**: Tools like Celonis and ProM analyze process data to identify bottlenecks and inefficiencies, providing insights for process improvement.
   - **Performance Monitoring**: Tools like New Relic, Dynatrace, and Splunk monitor the performance of software applications in real-time, helping to identify and resolve performance issues.

- Benefits of Process Technology
- **Enhanced Efficiency**: Automation tools reduce manual effort, streamline workflows, and increase productivity.
- **Improved Quality**: Continuous integration, automated testing, and code analysis tools help detect and fix defects early, resulting in higher quality software.
- **Better Collaboration**: Project management and communication tools foster better teamwork and coordination.
- **Faster Delivery**: Continuous deployment tools enable rapid and reliable delivery of new features and fixes.

## 2.7 PRODUCT AND PROCESS

The relationship between product and process is a key focus. Understanding this relationship is essential for producing high-quality software. Here's an overview of the key concepts and their interrelation:

1. Product
The **product** in software engineering refers to the software system or application that is developed to meet the needs of users. It includes all the deliverables produced during the software development lifecycle, such as code, documentation, user manuals, and test cases.

**Key Aspects of Product**:

- **Functionality**: The set of features and capabilities the software provides to meet user requirements.
- **Performance**: How well the software performs under various conditions, including speed, responsiveness, and resource usage.
- **Reliability**: The software's ability to perform consistently and without failure.
- **Usability**: The ease with which users can interact with the software.
- **Maintainability**: The ease with which the software can be modified to fix defects, improve performance, or adapt to a changing environment.

**2. Process**
The **process** refers to the series of activities, methods, and practices used to develop and maintain the software product. A well-defined process ensures that the software is developed systematically and efficiently, adhering to quality standards.

**Key Aspects of Process**:

- **Process Models**: Frameworks like Waterfall, Agile, and DevOps that define the sequence of phases in software development.
- **Best Practices**: Techniques and methods that have been proven to be effective in software development, such as code reviews, pair programming, and continuous integration.
- **Tools and Techniques**: Software and methodologies that support the process, including version control systems, project management tools, and automated testing frameworks.

The Relationship between Product and Process
The quality of the software product is directly influenced by the process used to develop it. A well-defined and consistently followed process leads to higher quality products. Conversely, deficiencies in the process can result in defects, inefficiencies, and poor-quality software.

**Pressman emphasizes the following points**:

- **Process Maturity**: Organizations with mature processes (e.g., those that follow CMMI or ISO standards) tend to produce higher quality products.
- **Continuous Improvement**: Regular assessment and improvement of the process (using models like SCAMPI or SPICE) lead to better product quality over time.
- **Process Tailoring**: Adapting the process to fit the specific needs of a project or organization can enhance both process efficiency and product quality.

Integration of Product and Process

**Key Strategies**:

- **Feedback Loops**: Incorporating feedback from product testing and user experience into process improvement.
- **Quality Metrics**: Using metrics such as defect density, code complexity, and user satisfaction to evaluate and improve both product and process.
- **Automated Tools**: Leveraging tools for automated testing, continuous integration, and performance monitoring to ensure that the product meets quality standards and the process remains efficient.

**Benefits**:

- Higher customer satisfaction due to improved product quality.
- Reduced development costs and time due to efficient processes.
- Increased ability to adapt to changing requirements and market conditions.
- Improved team morale and productivity through systematic process management.

## 2.8 SUMMARY

Software engineering is a layered technology that integrates multiple aspects to ensure the development of high-quality software. A robust process framework underpins this technology, guiding the systematic execution of development activities. The Capability Maturity Model Integration (CMMI) offers a structured path for process improvement, enhancing organizational maturity and performance. Process patterns provide reusable solutions to common problems, promoting efficiency and consistency. Through rigorous process assessment methods, organizations can identify strengths and areas for improvement, fostering continuous enhancement of their processes. Personal and team process models, such as the Personal Software Process (PSP) and Team Software Process (TSP), emphasize disciplined practices and collaborative teamwork to achieve superior results. Process technology, encompassing tools and techniques, supports the effective implementation of these processes. Lastly, the intrinsic link between product and process underscores the importance of a well-defined process in achieving high-quality software products. By integrating these elements, organizations can achieve greater productivity, quality, and customer satisfaction in their software engineering endeavors.

## 2.9 TECHNICAL TERMS

Quality Focus, Process, Pattern, Software, Framework, Software Layer Technology, Process Maturity, Process Tailoring.

## 2.10 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. Explain the concept of Software Engineering as a Layered Technology. Discuss its significance and the key layers involved.
2. Describe the Capability Maturity Model Integration (CMMI) and its importance in process improvement. How does it help organizations achieve their goals?
3. Discuss the role of Process Patterns in software engineering. Provide examples of different types of process patterns and their applications.

**Short questions:**

1. What is the role of Process Technology in software engineering?
2. How does the Team Software Process (TSP) extend the principles of PSP?
3. Explain the relationship between Product and Process in software engineering.

## 2.11 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Neelima Guntupalli

# LESSON- 03
# PROCESS MODELS

## AIMS AND OBJECTIVES

The primary goal of this chapter is to understand the process model. The chapter began with course objectives on Process Models includes Prescriptive Models, The Waterfall Model, Incremental Process Models, The Incremental Model, and The RAD Model. In addition, Evolutionary Process Model includes Prototyping, The Spiral Model, The Concurrent Development Model. The additional Specialized Process Models includes Component Based Development, The formal Methods Model, The Unified Process. After completing this chapter, the student will understand the complete knowledge about process models.

## STRUCTURE

## 3.1. INTRODUCTION

Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as Software Development Process.
- SDLC is a framework defining tasks performed at each step in the software development process.

- ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

What is SDLC?

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.
The following figure is a graphical representation of the various stages of a typical SDLC.

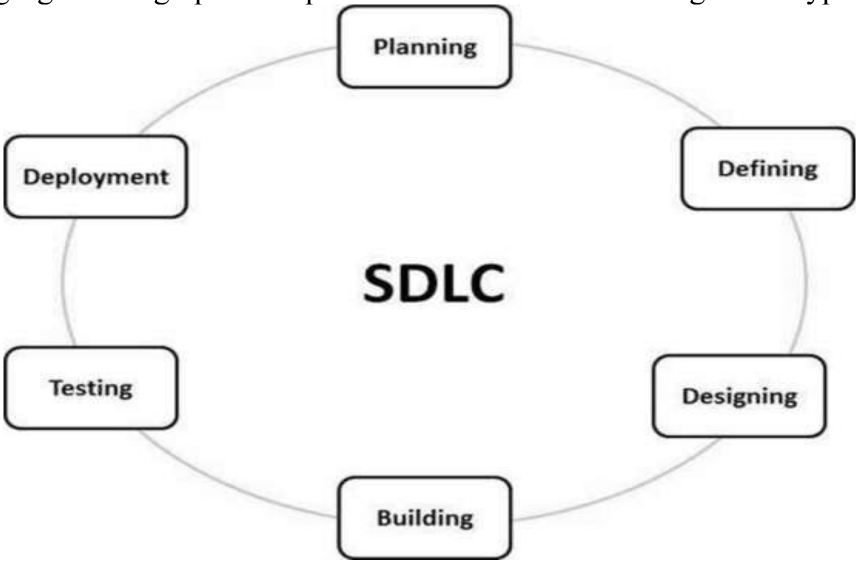


Fig 3.1 A typical Software Development Life Cycle consists of the following stages

**SDLC Models**

There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development Process Models". Each process model follows a Series of steps unique to its type to ensure success in the process of software development.
Following are the most important and popular SDLC models followed in the industry –

- Waterfall Model
- Iterative Model
- Spiral Model
- V-Model
- Big Bang Model

Other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

The chapter began with course objectives on Process Models includes Prescriptive Models, The Waterfall Model, Incremental Process Models, The Incremental Model, and The RAD Model. In addition, Evolutionary Process Model includes Prototyping, The Spiral Model, The Concurrent Development Model. The additional Specialized Process Models includes Component Based Development, The formal Methods Model, The Unified Process.

## 3.2. WATERFALL MODEL

The Waterfall Model was the first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed before the next phase can begin and there is no overlapping in the phases.

The Waterfall model is the earliest SDLC approach that was used for software development. The waterfall Model illustrates the software development process in a linear sequential flow. This means that any phase in the development process begins only if the previous phase is complete. In this waterfall model, the phases do not overlap.

### Waterfall Model – Design

Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.

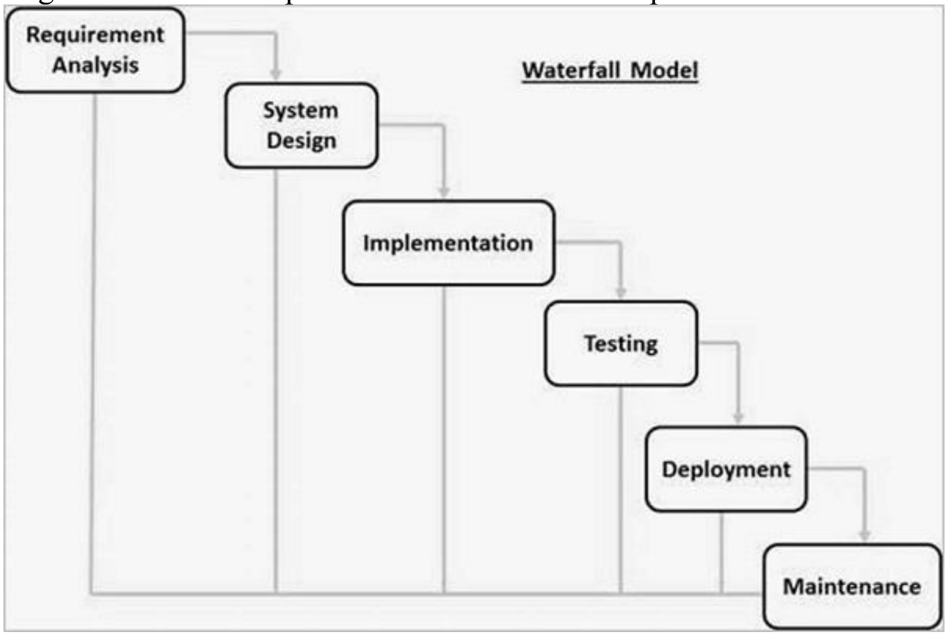


Fig 3.2 waterfall model life cycle

The sequential phases in Waterfall model are –

- **Requirement Gathering and analysis** − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.
- **System Design** − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.
- **Implementation** − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

- **Integration and Testing** − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.
- **Deployment of system** − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.
- **Maintenance** − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.
- All these phases are cascaded to each other in which progress is seen as flowing steadily downwards (like a waterfall) through the phases. The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name "Waterfall Model". In this model, phases do not overlap.

**Waterfall Model – Application**

Every software developed is different and requires a suitable SDLC approach to be followed based on the internal and external factors. Some situations where the use of Waterfall model is most appropriate are −

- Requirements are very well documented, clear and fixed.
- Product definition is stable.
- Technology is understood and is not dynamic.
- There are no ambiguous requirements.
- Ample resources with required expertise are available to support the product.
- The project is short.

**Waterfall Model – Advantages**

The advantages of waterfall development are that it allows for departmentalization and control. A schedule can be set with deadlines for each stage of development and a product can proceed through the development process model phases one by one.
Development moves from concept, through design, implementation, testing, installation, troubleshooting, and ends up at operation and maintenance. Each phase of development proceeds in strict order.
Some of the major advantages of the Waterfall Model are as follows –

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model. Each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.
- Clearly defined stages.
- Well understood milestones.
- Easy to arrange tasks.
- Process and results are well documented.

**Waterfall Model – Disadvantages**
The disadvantage of waterfall development is that it does not allow much reflection or revision. Once an application is in the testing stage, it is very difficult to go back and change something that was not well-documented or thought upon in the concept stage.

The major disadvantages of the Waterfall Model are as follows −

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model.
- It is difficult to measure progress within stages.
- Cannot accommodate changing requirements.
- Adjusting scope during the life cycle can end a project.

Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early.

## 3.3 ITERATIVE MODEL

In the Iterative model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to be deployed.

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which is then reviewed to identify further requirements. This process is then repeated, producing a new version of the software at the end of each iteration of the model.

### Iterative Model – Design

Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented. At each iteration, design modifications are made and new functional capabilities are added. The basic idea behind this method is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental).

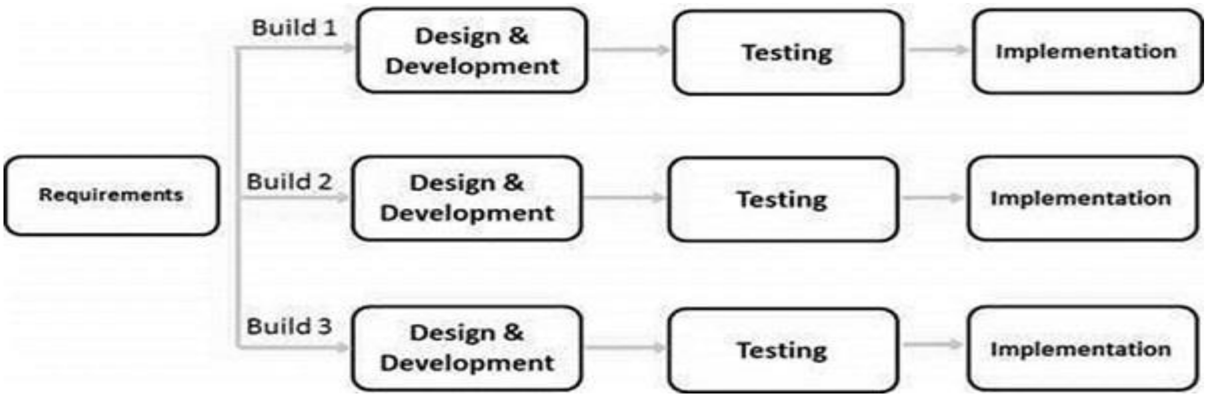


Fig 3.2 Iterative and Incremental Model Life Cycle

Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach."

In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

The key to a successful use of an iterative software development lifecycle is rigorous validation of requirements, and verification & testing of each version of the software against those requirements within each cycle of the model. As the software evolves through successive cycles, tests must be repeated and extended to verify each version of the software.

**Iterative Model – Application**

Like other SDLC models, Iterative and incremental development has some specific applications in the software industry. This model is most often used in the following scenarios −

- Requirements of the complete system are clearly defined and understood.
- Major requirements must be defined; however, some functionalities or requested enhancements may evolve with time.
- There is a time to the market constraint.
- A new technology is being used and is being learnt by the development team while working on the project.
- Resources with needed skill sets are not available and are planned to be used on contract basis for specific iterations.
- There are some high-risk features and goals which may change in the future.

**Iterative Model - Pros and Cons**

The advantage of this model is that there is a working model of the system at a very early stage of development, which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.
The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.
The advantages of the Iterative and Incremental SDLC Model are as follows −

- Some working functionality can be developed quickly and early in the life cycle.
- Results are obtained early and periodically.
- Parallel development can be planned.
- Progress can be measured.
- Less costly to change the scope/requirements.
- Testing and debugging during smaller iteration is easy.
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone.
- Easier to manage risk - High risk part is done first.
- With every increment, operational product is delivered.
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment.
- Risk analysis is better.
- It supports changing requirements.
- Initial Operating time is less.
- Better suited for large and mission-critical projects.

- During the life cycle, software is produced early which facilitates customer evaluation and feedback.
- The disadvantages of the Iterative and Incremental SDLC Model are as follows −
- More resources may be required.
- Although cost of change is lesser, but it is not very suitable for changing requirements.
- More management attention is required.
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle.
- Defining increments may require definition of the complete system.
- Not suitable for smaller projects.
- Management complexity is more.
- End of project may not be known which is a risk.
- Highly skilled resources are required for risk analysis.
- Projects progress is highly dependent upon the risk analysis phase.

## 3.4 RAD MODEL

RAD model distributes the analysis, design, build and test phases into a series of short, iterative development cycles.
Following are the various phases of the RAD Model −

Business Modelling
The business model for the product under development is designed in terms of flow of information and the distribution of information between various business channels. A complete business analysis is performed to find the vital information for business, how it can be obtained, how and when is the information processed and what are the factors driving successful flow of information.

Data Modelling
The information gathered in the Business Modelling phase is reviewed and analyzed to form sets of data objects vital for the business. The attributes of all data sets is identified and defined. The relation between these data objects are established and defined in detail in relevance to the business model.

Process Modelling
The data object sets defined in the Data Modelling phase are converted to establish the business information flow needed to achieve specific business objectives as per the business model. The process model for any changes or enhancements to the data object sets is defined in this phase. Process descriptions for adding, deleting, retrieving or modifying a data object are given.
Application Generation
The actual system is built and coding is done by using automation tools to convert process and data models into actual prototypes.

Testing and Turnover
The overall testing time is reduced in the RAD model as the prototypes are independently tested during every iteration. However, the data flow and the interfaces between all the components need to be thoroughly tested with complete test coverage. Since most of the programming components have already been tested, it reduces the risk of any major issues.

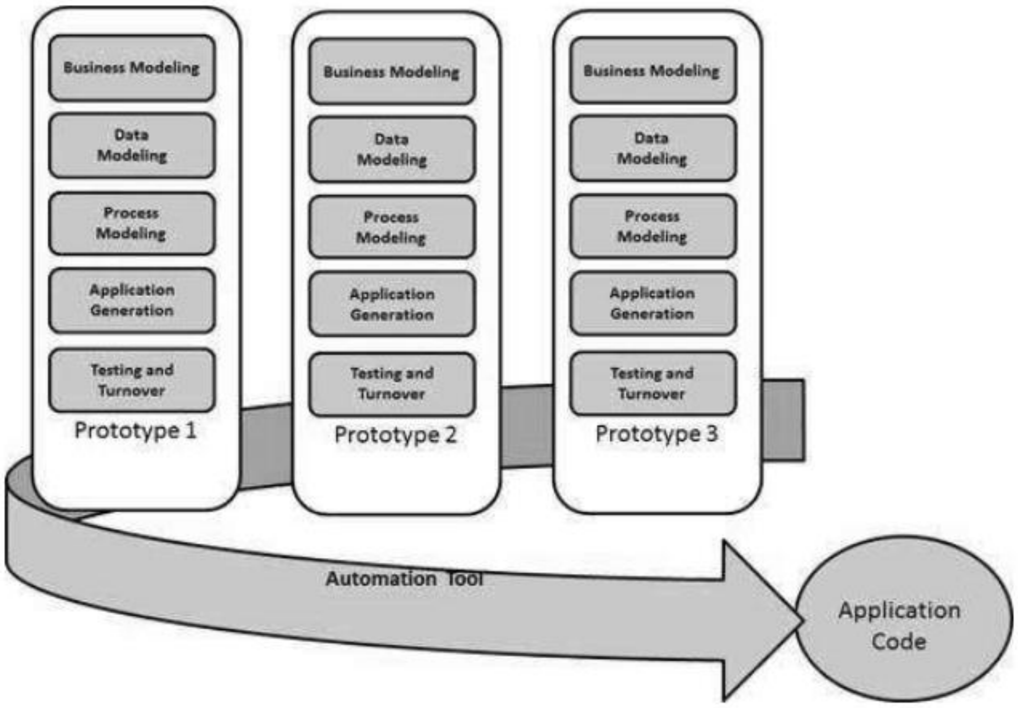


Fig 3.3 The RAD Model Life Cycle

The following pointers describe the typical scenarios where RAD can be used −

- RAD should be used only when a system can be modularized to be delivered in an incremental manner.
- It should be used if there is a high availability of designers for Modelling.
- It should be used only if the budget permits use of automated code generating tools.
- RAD SDLC model should be chosen only if domain experts are available with relevant business knowledge.
- Should be used where the requirements change during the project and working prototypes are to be presented to customer in small iterations of 2-3 months.

RAD Model - Pros and Cons

RAD model enables rapid delivery as it reduces the overall development time due to the reusability of the components and parallel development. RAD works well only if high skilled engineers are available and the customer is also committed to achieve the targeted prototype in the given time frame. If there is commitment lacking on either side the model may fail.

The advantages of the RAD Model are as follows −

- Changing requirements can be accommodated.
- Progress can be measured.
- Iteration time can be short with use of powerful RAD tools.
- Productivity with fewer people in a short time.
- Reduced development time.
- Increases reusability of components.
- Quick initial reviews occur.
- Encourages customer feedback.
- Integration from very beginning solves a lot of integration issues.
- The disadvantages of the RAD Model are as follows −
- Dependency on technically strong team members for identifying business requirements.
- Only system that can be modularized can be built using RAD.

- Requires highly skilled developers/designers.
- High dependency on Modelling skills.
- Inapplicable to cheaper projects as cost of Modelling and automated code generation is very high.
- Management complexity is more.
- Suitable for systems that are component based and scalable.
- Requires user involvement throughout the life cycle.
- Suitable for project requiring shorter development times.

## 3.5 PROTOTYPE MODEL

The Software Prototyping refers to building software application prototypes which displays the functionality of the product under development, but may not actually hold the exact logic of the original software.

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 a stepwise approach explained 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. While, 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 the actual implementation. The changes accepted are again incorporated in the new Prototype developed and the cycle repeats until the customer expectations are met.

Prototypes can have horizontal or vertical dimensions. A 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 softwa
re prototypes used in the industry. Following are the major software prototyping types used widely –

**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.

**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. By using evolutionary prototyping, the well-understood requirements are included in the prototype and the requirements are added as and when they are understood.

**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.

**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.

**Software Prototyping - Pros and Cons**

Software prototyping is used in typical cases and the decision should be taken very carefully so that the efforts spent in building the prototype add considerable value to the final software developed. The model has its own pros and cons discussed as follows.
The advantages of the Prototyping Model are as follows −

- Increased user involvement in the product even before its implementation.
- Since a working model of the system is displayed, the users get a better understanding of the system being developed.

- Reduces time and cost as the defects can be detected much earlier.

- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily.
- Confusing or difficult functions can be identified.
- The Disadvantages of the Prototyping Model are as follows −
- Risk of insufficient requirement analysis owing to too much dependency on the prototype.
- Users may get confused in the prototypes and actual systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Developers may try to reuse the existing prototypes to build the actual system, even when it is not technically feasible.
- The effort invested in building prototypes may be too much if it is not monitored properly.

## 3.6 SPIRAL MODEL

The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each iteration around the spiral.

**Spiral Model – Design**

The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.

**Identification**

This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.

This phase also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral, the product is deployed in the identified market.

**Design** The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

**Construct or Build**

The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.
Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to the customer for feedback.

**Evaluation and Risk Analysis**

Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.

The following illustration is a representation of the Spiral Model, listing the activities in each phase.

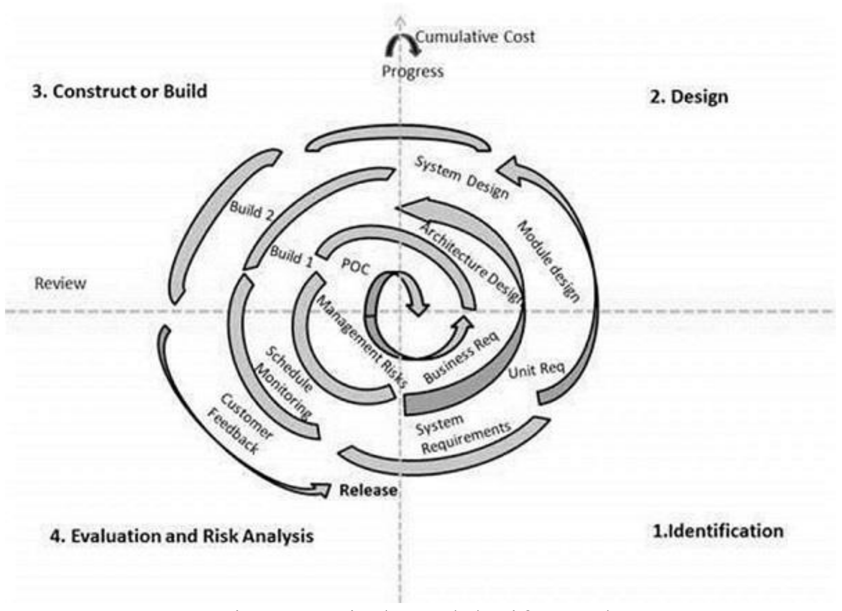


Fig 3.4 Spiral Model Life Cycle

Based on the customer evaluation, the software development process enters the next iteration and subsequently follows the linear approach to implement the feedback suggested by the customer. The process of iterations along the spiral continues throughout the life of the software.

**Spiral Model Application**

The Spiral Model is widely used in the software industry as it is in sync with the natural development process of any product, i.e. learning with maturity which involves minimum risk for the customer as well as the development firms.
The following pointers explain the typical uses of a Spiral Model –

- When there is a budget constraint and risk evaluation is important.
- For medium to high-risk projects.
- Long-term project commitment because of potential changes to economic priorities as the requirements change with time.
- Customer is not sure of their requirements which is usually the case.
- Requirements are complex and need evaluation to get clarity.
- New product line which should be released in phases to get enough customer feedback.
- Significant changes are expected in the product during the development cycle.

**Spiral Model - Pros and Cons**
The advantage of spiral lifecycle model is that it allows elements of the product to be added in, when they become available or known. This assures that there is no conflict with previous requirements and design.
This method is consistent with approaches that have multiple software builds and releases which allows making an orderly transition to a maintenance activity. Another positive aspect of this method is that the spiral model forces an early user involvement in the system development effort.
On the other side, it takes a very strict management to complete such products and there is a risk of running the spiral in an indefinite loop. So, the discipline of change and the extent of taking change requests is very important to develop and deploy the product successfully.
The advantages of the Spiral SDLC Model are as follows −

- Changing requirements can be accommodated.
- Allows extensive use of prototypes.
- Requirements can be captured more accurately.
- Users see the system early.
- Development can be divided into smaller parts and the risky parts can be developed earlier which helps in better risk management.
- The disadvantages of the Spiral SDLC Model are as follows −
- Management is more complex.
- End of the project may not be known early.
- Not suitable for small or low risk projects and could be expensive for small projects.
- Process is complex
- Spiral may go on indefinitely.
- Large number of intermediate stages requires excessive documentation.

## 3.7 CONCURRENT DEVELOPMENT MODEL

The concurrent development model is called as concurrent model. The communication activity has completed in the first iteration and exits in the awaiting changes state. The modelling activity completed its initial communication and then go to the underdevelopment state. If the customer specifies the change in the requirement, then the modelling activity moves from the under-development state into the awaiting change state. The concurrent process model activities moving from one state to another state.

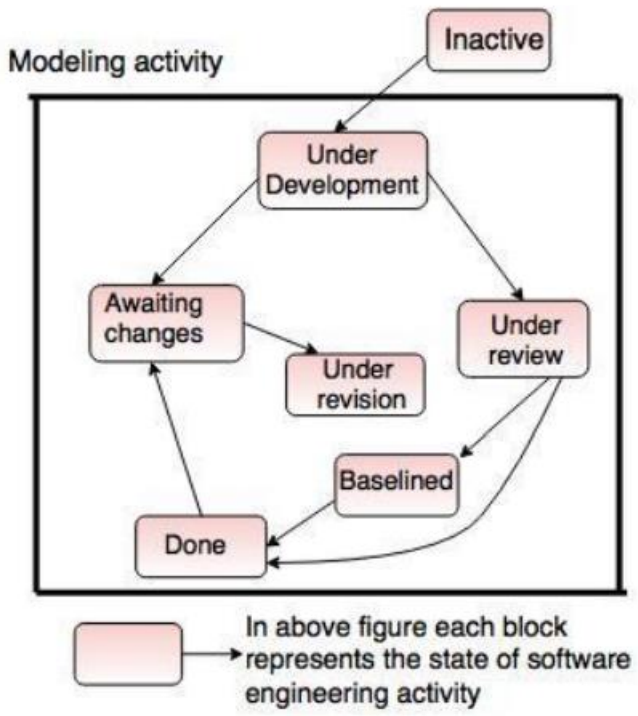


Fig 3.5 One element of the Concurrent Model

**Advantages of the concurrent development model**

- This model is applicable to all types of software development processes.
- It is easy for understanding and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

**Disadvantages of the concurrent development model**

- It needs better communication between the team members. This may not be achieved all the time.
- It requires to remember the status of the different activities.

## 3.8 COMPONENT BASED MODEL

The component-based assembly model uses object-oriented technologies. In object-oriented technologies, the emphasis is on the creation of classes. Classes are the entities that encapsulate data and algorithms. In component-based architecture, classes (i.e., components required to build application) can be uses as reusable components. This model uses various characteristics of spiral model. This model is evolutionary by nature. Hence, software development can be done using iterative approach. In CBD model, multiple classes can be used. These classes are basically the prepackaged components. The model works in following manner:

- **Step-1:** First identify all the required candidate components, i.e., classes with the help of application data and algorithms.
- **Step-2:** If these candidate components are used in previous software projects then they must be present in the library.
- **Step-3:** Such preexisting components can be excited from the library and used for further development.
- **Step-4:** But if the required component is not present in the library then build or create the component as per requirement.
- **Step-5:** Place this newly created component in the library. This makes one iteration of the system.
- **Step-6:** Repeat steps 1 to 5 for creating n iterations, where n denotes the number of iterations required to develop the complete application.

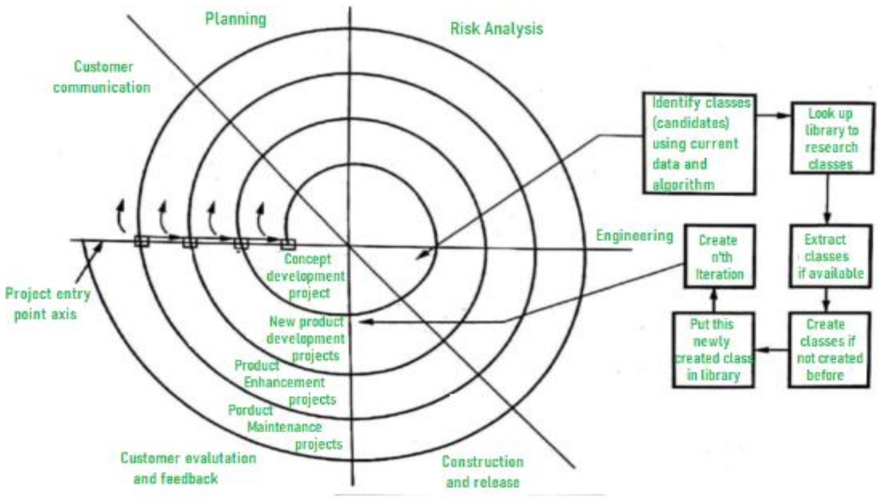

Planning
Risk Analysis
Customer communication
Identify classes (candidates) using current data and algorithm
Look up library to research classes
Engineering
Create n'th Iteration
Extract classes if available
Project entry point axis
Concept development project
New product development projects
Product Enhancement projects
Porduct Maintenance projects
Put this newly created class in library
Create classes if not created before
Customer evalutation and feedback
Construction and release

Fig 3.6 Component Based Model Life Cycle

**Characteristics of Component Assembly Model:**

- Uses object-oriented technology.
- Components and classes encapsulate both data and algorithms.
- Components are developed to be reusable.
- Paradigm similar to spiral model, but engineering activity involves components.
- The system produced by assembling the correct components.

## 3.9 FORMAL MODEL METHOD

The **formal methods model** is concerned with the application of a mathematical technique to design and implement the software. This model lays the foundation for developing a complex system and supporting the program development. The formal methods used during the development process provide a mechanism for eliminating problems, which are difficult to overcome using other software process models. The software engineer creates formal specifications for this model. These methods minimize specification errors and this result in fewer errors when the user begins using the system.

Formal methods comprise *formal specification* using mathematics to specify the desired properties of the system. Formal specification is expressed in a language whose syntax and semantics are formally defined. This language comprises a syntax that defines specific notation used for specification representation; semantic, which uses objects to describe the system; and a set of relations, which uses rules to indicate the objects for satisfying the specification.

Generally, the formal method comprises two approaches, namely, property based and model-based. The **property-based specification** describes the operations performed on the system. In addition, it describes the relationship that exists among these operations. A property-based specification consists of two parts: signatures, which determine the syntax of operations and an equation, which defines the semantics of the operations through a set of equations known as **axioms.** The **model-based specification** utilizes the tools of set theory, function theory, and logic to develop an abstract model of the system. In addition, it specifies the operations performed on the abstract model. The model thus developed is of a high level and idealized. A model-based specification comprises a definition of the set of states of the system and definitions of the legal operations performed on the system to indicate how these legal operations change the current state.

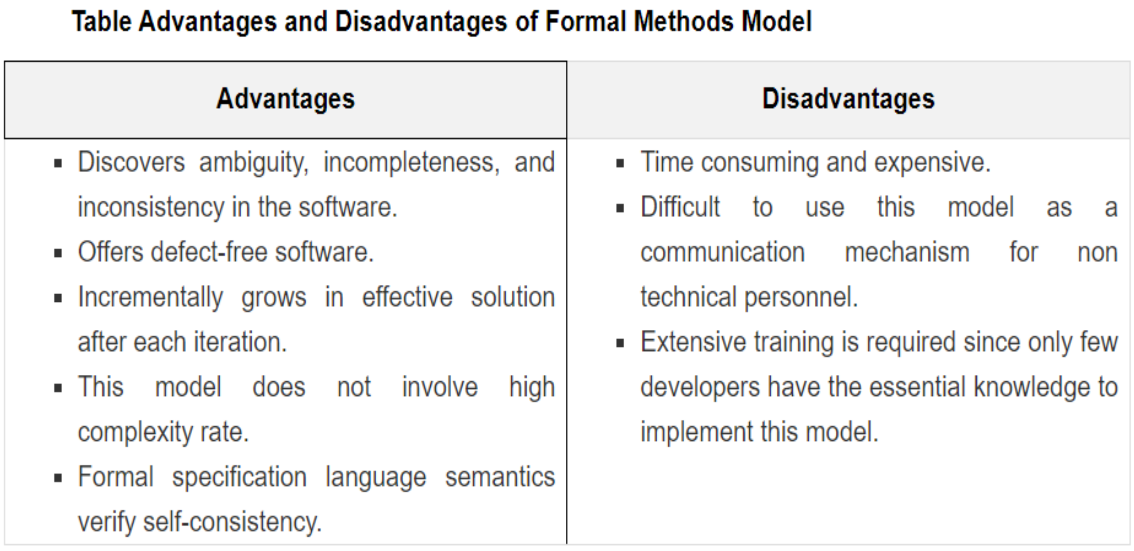


**Table Advantages and Disadvantages of Formal Methods Model**

| Advantages | Disadvantages |
|---|---|
| ▪ Discovers ambiguity, incompleteness, and inconsistency in the software.<br>▪ Offers defect-free software.<br>▪ Incrementally grows in effective solution after each iteration.<br>▪ This model does not involve high complexity rate.<br>▪ Formal specification language semantics verify self-consistency. | ▪ Time consuming and expensive.<br>▪ Difficult to use this model as a communication mechanism for non technical personnel.<br>▪ Extensive training is required since only few developers have the essential knowledge to implement this model. |

Fig 3.7 Advantages and Disadvantages of Formal Model Method

## 3.10 SUMMARY

In this chapter, we explored a diverse range of software process models, each providing distinct methodologies for software development. We began with prescriptive models, including the Waterfall Model, which emphasizes a linear and sequential approach. We then examined incremental process models, such as the Incremental Model and the RAD Model, which focus on iterative development and quick delivery. Evolutionary process models, including Prototyping, the Spiral Model, and the Concurrent Development Model, were discussed for their iterative and risk-driven approaches. Finally, we looked at specialized process models, like Component-Based Development, the Formal Methods Model, and the Unified Process, which offer tailored solutions for specific project needs. This comprehensive understanding enables the selection of the most appropriate model for different software projects, balancing factors such as project size, complexity, risk, and the need for flexibility.

## 3.11 TECHNICAL TERMS

Process Mode, Waterfall Model, Incremental Model, RAD, Spiral, Concurrent, Formal Model

## 3.12 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. Discuss the Advantages and Disadvantages of Prescriptive Models in Software Development.
2. Analyze the Waterfall Model and Its Applicability to Modern Software Development Projects
3. Compare and Contrast Incremental Process Models: The Incremental Model vs. The RAD Model.

**Short questions:**

1. What are the main characteristics of prescriptive process models?
2. Describe the phases of the Waterfall Model.
3. What are the key differences between the Incremental Model and the RAD Model?
4. How does the Prototyping Model help in refining user requirements?
5. Explain the risk-driven approach of the Spiral Model.

## 3.13 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Neelima Guntupalli

LESSON- 04

# AN AGILE VIEW OF PROCESS

## AIMS AND OBJECTIVES

The primary goal of this chapter is to understand An Agile View of Process. The chapter began with course objectives on What is Agility? What is Agile Process? Agile Process Models: Extreme Programming, Adaptive Software Development, Dynamic Systems Development Method, Scrum, Crystal, Feature Driven Development, Agile Modeling. objectives in brief list. After completing this chapter, the student will understand the complete knowledge about An Agile View of Process.

## STRUCTURE

## 4.1 INTRODUCTION

In the rapidly evolving field of software development, agility has emerged as a crucial attribute for teams seeking to adapt to changing requirements and deliver high-quality products efficiently. This chapter introduces the concept of agility, outlining its significance in fostering flexibility, collaboration, and customer-centric development. We will explore the essence of agile processes, emphasizing their iterative and incremental nature, which contrasts sharply with traditional linear approaches. The chapter will delve into various agile process models, including Extreme Programming (XP), Adaptive Software Development (ASD), Dynamic Systems Development Method (DSDM), Scrum, Crystal, Feature Driven Development (FDD), and Agile Modeling, providing a comprehensive overview of each methodology's principles, practices, and applicability.

## 4.2 AGILE PROCESS

Agility means characteristics of being dynamic, content specific, aggressively change embracing and growth oriented.

Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software, small highly motivated project teams, informal methods, minimal software engineering work products and overall development simplicity. The development guidelines stress delivery over analysis and design and active continuous communication between developers and customers. The team of software engineers and other project stakeholders work together as an agile team (a team that is self organizing and in control of its own destiny. An agile team fosters communication and collaboration among all who serve on it.

The Processes which are adaptable of changes in requirements, which have incrementality and work on unpredictability. These processes are based on three assumptions which all do refer to the unpredictability in different stages of software process development such unpredictability at time requirements, at analysis and design or at time construction. So these processes are adaptable at all stages on SDLC.

## 4.3 AGILE PROCESS MODELS

The agile model is additionally a sort of Incremental model. Programming is created in gradual, fast cycles. This outcome in little gradual deliveries with each delivery expanding on past usefulness. Each delivery is completely tried to guarantee programming quality is kept up with. It is utilized for time-critical applications.

The stages of the agile process model are as follows:

1. Requirements gathering
2. Design the requirements
3. Construction/ iteration
4. Testing
5. Deployment
6. Feedback

Read in the brief about the SDLC process model:

1. Requirements gathering

In this phase, requirements should be characterized. Different business opportunities are explained and for the building of the project efforts and time are planned.

2. Design the requirements

At the point when you have distinguished the task, work with partners to define the requirements. You can utilize the client stream graph to show crafted new elements and show how they will apply to your current framework.

3. Construction/ iteration

At the point when the group characterizes the requirements, the work starts. Creators and designers begin to work at their task, which means sending a functioning item. The item will go through different phases of progress, so it incorporates easy minimal functionality.

4. Testing

In this stage, the Quality Assurance group analyzes the item's presentation and searches for the bug.

5. Deployment

In this stage, the group gives an item for the client's workplace.

6. Feedback

When a product is released, feedback is the last step. Feedback is received regarding the product and it manages the input.

## 4.4 EXTREME PROGRAMMING(XP)

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing.

Key XP activities are summarized in the paragraphs that follow

❖ **Planning**. The planning activity begins with listening—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality. Listening leads to the creation of a set of "stories" or user stories that describe required output, features, and functionality for software to be built. Each story is written by the customer and is placed on an index card. The customer assigns a value (i.e., a priority) to the story based on the overall business value of the feature or function. Members of the XP team then assess each story and assign a cost—measured in development weeks—to it. If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time. Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team.

Once a basic commitment is made for a release, the XP team orders the stories that will be developed in one of three ways:

(1) all stories will be implemented immediately (within a few weeks),
(2) the stories with highest value will be moved up in the schedule and implemented first, or
(3) the riskiest stories will be moved up in the schedule and implemented first.

After the first project release (also called a software increment) has been delivered, the XP team computes project velocity. Stated simply, project velocity is the number of customer stories implemented during the first release. Project velocity can then be used to (1) help estimate delivery dates and schedule for subsequent releases and (2) determine whether an overcommitment has been made for all stories across the entire development project. As development work proceeds, the customer can add stories, change the value of an existing story, split stories, or eliminate them. The XP team then reconsiders all remaining releases and modifies its plans accordingly.

❖ **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context. CRC (class responsibility collaborator) cards identify and organize the object-oriented classes that are relevant to the current software increment. The CRC cards are the only design work product produced as part of the XP process. If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a spike solution, the design prototype is implemented and evaluated. The intent is to lower risk when true implementation starts and to validate the original estimates for the story containing the design problem A central notion in XP is that design occurs both before and after coding commences. Refactoring means that design occurs continuously as the system is constructed. In fact, the construction activity itself will provide the XP team with guidance on how to improve the design.

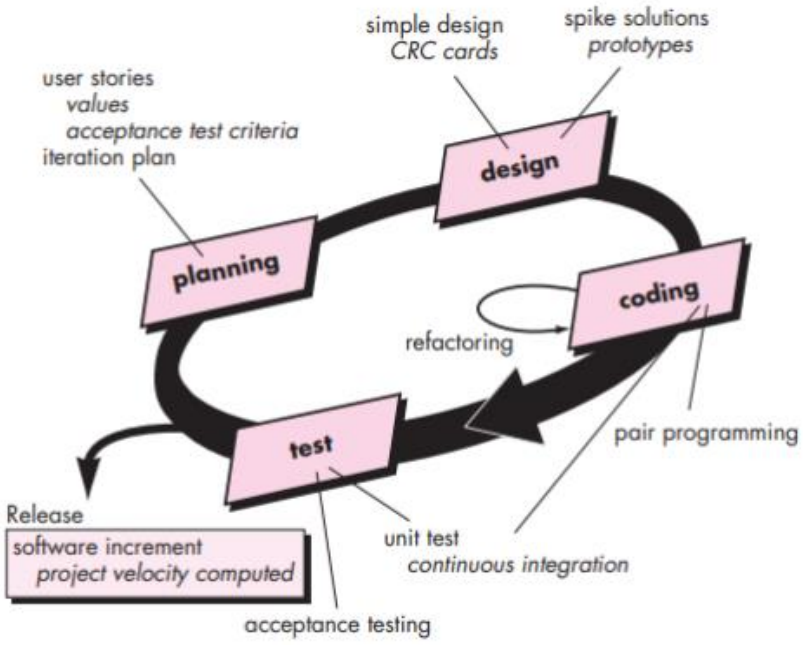


Fig 4.1 Life Cycle of XP Model

❖ **Coding.** After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release (software increment).8 Once the unit test9 has been created, the developer is better able to focus on what must be implemented to pass the test. Nothing extraneous is added (KIS). Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers. A key concept during the coding activity is pair programming. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving and real-time quality assurance It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role. For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story.

As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases, this is performed on a daily basis by an integration team. In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

❖ **Testing** The unit tests that are created should be implemented using a framework that enables them to be automated This encourages a regression testing strategy whenever code is modified As the individual unit tests are organized into a "universal testing suite" [Wel99], integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells [Wel99] states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline." XP acceptance tests, also called customer tests, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

## 4.5 ADAPTIVE SOFTWARE DEVELOPMENT(ASD)

Adaptive Software Development (ASD) has been proposed by Jim Highsmith, as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization. He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

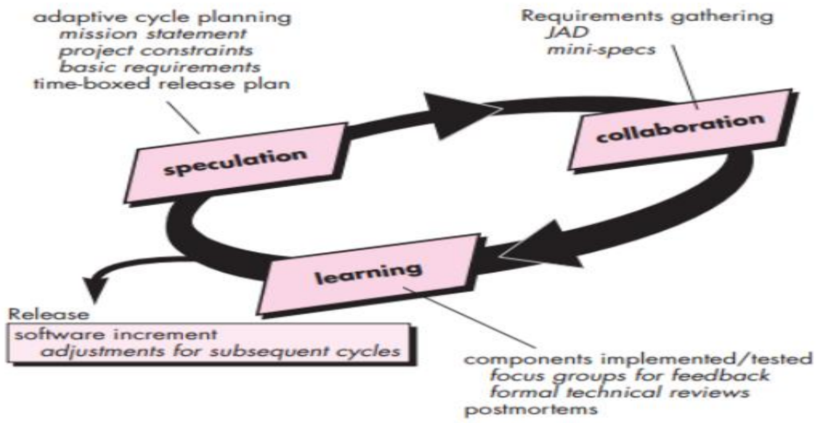


Fig 4.2 Life Cycle of ASD Model

During speculation, the project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information—the customer's mission statement, project constraints , and basic requirements—to define the set of release cycles that will be required for the project. No matter how complete and farsighted the cycle plan, it will invariably change. Based on information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality in which an ASD team is working. Motivated people use collaboration in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust.

People working together must trust one another to

1) criticize without animosity,
2) assist without resentment,
3) work as hard as or harder than they do,
4) have the skill set to contribute to the work at hand, and
5) communicate problems or concerns in a way that leads to effective action.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on "learning" as much as it is on progress toward a completed cycle. software developers often overestimate their own understanding (of the technology, the process, and the project) and that learning will help them to improve their level of real understanding. ASD teams learn in three ways: focus groups , technical reviews, , and project postmortems. The ASD philosophy has merit regardless of the process model that is used. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

## 4.6 DYNAMIC SOFTWARE DEVELOPMENT METHOD(DSDM)

The Dynamic Systems Development Method is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment". The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete application. DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated. The DSDM Consortium is a worldwide group of member companies that collectively take on the role of "keeper" of the method.

The consortium has defined an agile process model, called the DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

- ❖ Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

- ❖ Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

- ❖ Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer. The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

- Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, functional model iteration and design and build iteration occur concurrently.

- Implementation—places the latest software increment into the operational environment. It should be noted that

  1) the increment may not be 100 percent complete or
  2) changes may be requested as the increment is put into place.

In either case, DSDM development work continues by returning to the functional model iteration activity. DSDM can be combined with XP to provide a combination approach that defines a solid process model with the nuts-and-bolts practices (XP) that are required to build software increments. In addition, the ASD concepts of collaboration and self-organizing teams can be adapted to a combined process model.

**4.7 SCRUM**

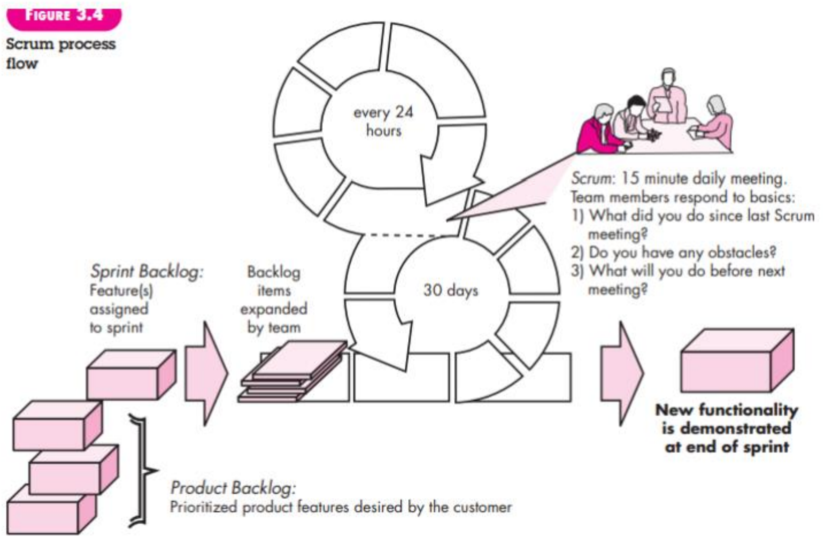


FIGURE 3.4
Scrum process flow
every 24 hours
Scrum: 15 minute daily meeting.
Team members respond to basics:
1) What did you do since last Scrum meeting?
2) Do you have any obstacles?
3) What will you do before next meeting?
Sprint Backlog: Feature(s) assigned to sprint
Backlog items expanded by team
30 days
New functionality is demonstrated at end of sprint
Product Backlog: Prioritized product features desired by the customer

Fig 4.3 SCRUM Model

Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. In recent years, further development on the Scrum methods has been performed by Schwaber and Beedle. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a sprint. The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality.

Each of these process patterns defines a set of development actions: Backlog—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time (this is how changes are introduced). The product manager assesses the backlog and updates priorities as required. Sprints—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box14 (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment. Scrum meetings—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members. • What did you do since the last team meeting? • What obstacles are you encountering? • What do you plan to accomplish by the next team meeting? A team leader, called a Scrum master, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "knowledge socialization" [Bee99] and thereby promote a self-organizing team structure. Demos—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established. The Scrum process patterns enable a software team to work successfully in a world where the elimination of uncertainty is impossible.

## 4.8 CRYSTAL

Alistair Cockburn and Jim Highsmith created the Crystal family of agile methods15 in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game". To achieve maneuverability, Cockburn and Highsmith have defined a set of methodologies, each with core elements that are common to all, and roles, process patterns, work products, and practice that are unique to each. The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

## 4.9 FEATURE DRIVEN DEVELOPMENT (FDD)

The FDD approach defines five "collaborating" framework activities (in FDD these are called "processes"). FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. As projects grow in size and complexity, ad hoc project management is often inadequate. It is essential for developers, their managers, and other stakeholders to understand project status—what accomplishments have been made and problems have been encountered. If deadline pressure is significant, it is critical to determine if software increments (features) are properly scheduled. To accomplish this, FDD defines six milestones during the design and implementation of a feature: "design walkthrough, design, design inspection, code, code inspection, promote to build". Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improved Coad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

Like other agile approaches, FDD adopts a philosophy that

1) emphasizes collaboration among people on an FDD team;
2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and
3) communication of technical detail using verbal, graphical, and text-based means.

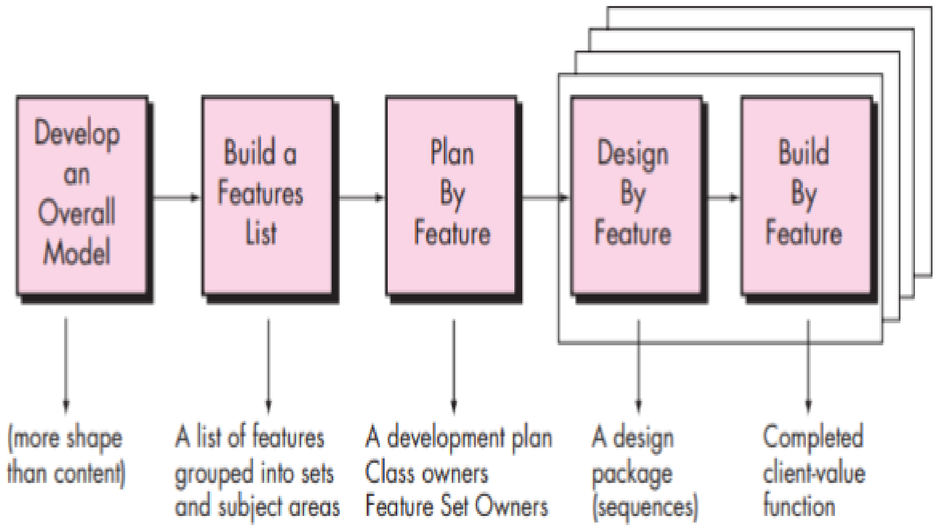

Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

Fig 4.4 The FDD Model

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits the collection of metrics, and the use of patterns (for analysis, design, and construction). FDD has following benefits: • Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions. • Features can be organized into a hierarchical business-related grouping. • Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks. • Because features are small, their design and code representations are easier to inspect effectively. • Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

## 4.10 AGILE MODELING(AM) :

There are many situations in which software engineers must build large, businesscritical systems. The scope and complexity of such systems must be modeled so that (1) all constituencies can better understand what needs to be accomplished, (2) the problem can be partitioned effectively among the people who must solve it, and (3) quality can be assessed as the system is being engineered and built. Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don’t have to be perfect. Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

## 4.11 SUMMARY

This chapter has provided a comprehensive overview of agility and agile processes, highlighting their importance in modern software development. We explored the core concepts of agility, emphasizing flexibility, customer collaboration, and responsiveness to change. The examination of various agile process models—including Extreme Programming (XP), Adaptive Software Development (ASD), Dynamic Systems Development Method (DSDM), Scrum, Crystal, Feature Driven Development (FDD), and Agile Modeling—demonstrated the diverse approaches teams can adopt to implement agile principles. Each model's unique practices and benefits illustrate how agility fosters effective, efficient, and adaptive software development. By understanding and applying these agile methodologies, development teams can enhance their ability to deliver high-quality software that meets evolving customer needs.

## 4.12 TECHNICAL TERMS

Process Mode, Waterfall Model ,Incremental Model ,RAD, Spiral, Concurrent, Formal Model

## 4.13 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. What is Agility in Software Development and Why is it Important?
2. Define Agile Processes and Their Key Characteristics.
3. Compare and Contrast Various Agile Process Models.

**Short questions:**

1. What is agility in software development?
2. List the core principles of agile methodologies.
3. Define an agile process.
4. What are the key characteristics of agile processes?
5. Name the primary agile process models.

## 4.14 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Neelima Guntupalli

LESSON- 05

# SOFTWARE ENGINEERING PRATICES

**AIMS AND OBJECTIVES**

The primary goal of this chapter is to understand Software Engineering Practices. The chapter began with understand of Software Engineering Practice, communication practices, Planning Practices, Modelling Practices, Construction Practices, and Deployment. After completing this chapter, the student will understand the complete knowledge about Software Engineering Practices.

**STRUCTURE**

## 5.1 INTRODUCTION

System Engineering involves the design, integration, and management of complex computer-based systems. It follows a structured hierarchy to ensure each system component functions cohesively within the overall framework. Business Process Engineering aims to analyze and improve organizational workflows to enhance efficiency and productivity. System modeling provides abstract representations of systems to visualize and understand their structure and behavior, facilitating better design, analysis, and communication. Together, these elements ensure the development of robust, efficient, and scalable systems that align with business objectives and user needs.

Software Engineering Practice encompasses a comprehensive set of methodologies and techniques aimed at developing high-quality software systems. It includes effective communication practices to ensure all stakeholders are aligned, detailed planning practices for structured project management, modeling practices to create abstract representations of the system, construction practices focused on coding and testing, and deployment practices to smoothly release and maintain software. Together, these practices facilitate the efficient and reliable delivery of software products that meet user requirements and maintain scalability and usability over time.

The chapter covers about the software engineering practices with the topics included in communication practices, Planning Practices, Modelling Practices, Construction Practices, and Deployment practices.

## 5.2 SOFTWARE ENGINEERING PRACTICE

Practice is a broad array of concepts, principles, methods, and tools that you must consider as software is planned and developed.
It represents the details

- the technical considerations and how to's
- that are below the surface of the software process
- the things that you'll need to actually build high-quality computer software

### ❖ The Essence of Practice

This section lists the generic framework (communication, planning, modeling, construction, and deployment) and umbrella (tracking, risk management, reviews, measurement, configuration management, reusability management, work product creation, and product) activities found in all software process models.

George Polya, in a book written in 1945 (!), describes the essence of software engineering practice:

- **Understand the problem** (communication and analysis).
- Who are the stakeholders?
- What are the unknowns? "Data, functions, features to solve the problem?"
- Can the problem be compartmentalized? "Smaller that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?
- **Plan a solution** (modeling and software design).
- Have you seen a similar problem before?
- Has a similar problem been solved? If so, is the solution reusable?
- Can sub-problems be defined?
- Can you represent a solution in a manner that leads to effective implementation?
- Carry out the plan (code generation).
- Does the solution conform to the plan?
- Is each component part of the solution probably correct?
- Examine the result for accuracy (testing and quality assurance).
- Is it possible to test each component part of the solution?
- Does the solution produce results that conform to the data, functions, features, and behavior that are required?

### ❖ Core Principles

1. **The Reason It All Exists: Provide Value to the Customer and the User** The ultimate purpose of any software system is to deliver value. Whether that value is in the form of efficiency, automation, convenience, or innovation, every design decision should align with improving the end-user's experience or fulfilling the customer's needs.
    - If a feature or task does not add measurable value, it should not be pursued.
    - Focus on solving *real problems*, not just building features for the sake of complexity.
    - Always ask: *Does this make the product better for the user?*
2. **KISS — Keep It Simple, Stupid!**

Simplicity is the cornerstone of good software design. Overly complex systems are difficult to understand, debug, test, and maintain.

- Strive for clarity over cleverness.
- Simple designs are more robust and adaptable to change.
- Remember Albert Einstein's advice: *"Everything should be made as simple as possible, but no simpler."*
- Avoid unnecessary abstractions and overengineering.

3. **Maintain the Product and Project "Vision"**
   A well-defined and consistently communicated vision guides all team members in the right direction.
   - The vision defines *what the product should achieve* and *how it aligns with user goals and organizational strategy*.
   - It ensures that day-to-day decisions support long-term objectives.
   - Without a clear vision, teams risk scope creep, conflicting priorities, and inconsistent results.
   - Revisit and refine the vision regularly as user needs and technology evolve.

4. **What You Produce, Others Will Consume**
   Software development is a collaborative process. The artifacts you produce — specifications, designs, code, and documentation — will be used by others, either within your team or by future maintainers.
   - Write clean, readable, and well-documented code.
   - Design with maintainability in mind.
   - Assume that the next person working on your code will not have the same context or assumptions as you.
   - Strive to make your work understandable and reusable.

5. **Be Open to the Future**
   Technology and requirements change rapidly. Good design anticipates change and avoids locking the system into a rigid structure.
   - Avoid hard-coded assumptions or solutions that only work for one scenario.
   - Design for extensibility and flexibility — for example, through modular architectures, interfaces, and abstraction layers.
   - Frequently ask "What if?" — *What if the requirements change? What if the scale increases? What if a new technology emerges?*
   - Build systems that solve the general problem, not just the immediate one.

**Additional Supporting Principles**
You may also include these to further strengthen your list:

6. **Think Before You Build**
   - A little planning saves a lot of rework.
   - Understand the problem domain thoroughly before jumping into coding.

7. **Iterate and Improve Continuously**
   - No design is perfect in the first attempt.
   - Use feedback loops, testing, and user evaluation to refine your design.

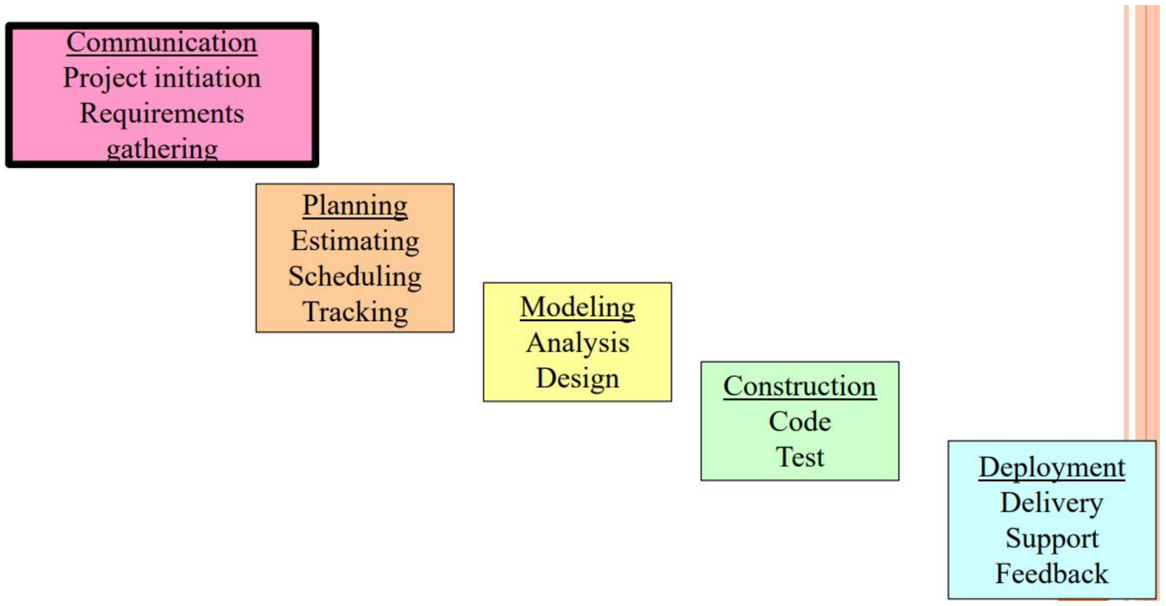
8. **Quality Is Everyone's Responsibility**
   - From requirements to deployment, every phase should emphasize quality.

- Quality isn't added later — it's built in from the start.

## 5.3 COMMUNICATION PRACTICES

Before customer requirements can be analyzed, modeled, or specified they must be gathered through a communication (also called requirement elicitation) activity.
Effective communication (among technical peers, with the customer and other stakeholders, and with project managers) is among the most challenging activities that confront a S/W engineer.
In this context, the following are communication **principles** and concepts that apply to customer communication:

❖ **Listen**: focus on the speaker's words, rather than formulating your response to those words. Be a polite listener.



❖ **Prepare before you communicate**: Spend the time to understand the problem before you meet with others "research".
❖ **Someone should facilitate the communication activity**. Have a leader "moderator" to keep the conversation moving in a productive direction.
❖ **Face-to-face communication is best**.
❖ **Take notes and document decisions**.
❖ **Collaborate with the customer**. Each small collaboration serves to build trust among team members and creates a common goal for the team.
❖ **Stay focused, modularize your discussion**. The facilitator should keep the conversation modular; leaving one topic only after it has been resolved.
❖ **Draw pictures when things are unclear**.
(a)Once you agree to something, move on;
(b) if you can't agree to something, move one;
(c) if a feature or function is unclear and can't be clarified at the moment, move on.

- **Negotiation is not a contest or a game. It works best when both parties win**.
Negotiation should not be viewed as a competition where one side wins and the other loses. Effective negotiation aims to reach a mutually beneficial agreement — a *win-win*

outcome — where both parties feel their needs and interests have been acknowledged and satisfied.

## 5.4 PLANNING PRACTICES

The planning activity encompasses a set of management and technical practices that enable the S/W team to define a road map as it travels toward its strategic goal and tactical objectives.
Regardless of the rigor with which planning is conducted, the following principles always apply:

- Understand the project scope. Scope provides the S/W team with a destination.
- Involve the customer (and other stakeholders) in the planning activity. The customer defines priorities and establishes project constraints. S/W engineers must often negotiate order of delivery, timelines, and other related issues.
- Recognize that planning is iterative. A plan must be adjusted to accommodate changes.
- Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.
- Consider risk as you define the plan.
- Be realistic. Even the best S/W engineers make mistakes.
- Adjust granularity as you plan. A fine granularity plan provides significant work task detail that is planned over relatively short time increments. A coarse granularity plan provides broader work tasks that are planned over longer time periods.
- Define how quality will be achieved.
- Define how you'll accommodate changes. "Can the customer request a change at any time?"
- Track what you've planned and make adjustments as required.

**Barry Boehm states:**

- "You need an organizing principle that scales down to provide simple plans for simple projects."
- Boehm suggests an approach that addresses project objectives, milestones, and schedules, responsibilities, management and technical approaches, and required resources.
- Boehm calls it W5HH principle, after a series of questions that lead to a definition of key project characteristics and the resultant project plan.
- Why is the system being developed? Does the business purpose justify the expenditure of people, time and money?
- What will be done? Identify the functionality to be built.
- When will it be accomplished? Establish a workflow and timeline for key project tasks and identify milestones required by the customer.
- Who is responsible for a function? Define members' roles and responsibilities.
- Where are they located (organizationally)? Customers also have responsibilities.

- How will the job be done technically and managerially? Once a scope is defined, a technical strategy must be defined.
- How much of each resource is needed? The answer is derived by developing estimates based on answers to earlier questions.

## 5.5 MODELING PRACTICES

The process of developing analysis and design models is described in this section. The emphasis is on describing how to gather the information needed to build reasonable models, but no specific modeling notations are presented in this chapter.

UML and other modeling notations are described in detail later in the text. In S/W Eng. work, two models are created: analysis models and design models.

➢ **Analysis models** represent the customer requirements by depicting the S/W in three different domains: the information domain, the functional domain, and the behavioral domain.

➢ **Design models** represent characteristics of the S/W that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

▪ **Analysis Modeling Principles**

➢ **The information domain of a problem must be represented and understood**. The information domain encompasses the data that flow into the system (end-users, other systems, or external devices), the data that flow out of the system and the data stores that collect and organize persistent data objects.

➢ **Represent software functions**. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

➢ **Represent software behavior.** The behavior of the S/W is driven by the interaction with the external environment.The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered fashion (or hierarchical).

➢ **The analysis task should move from essential information toward implementation detail**. Analysis begins by describing the problem from the end-user perspective. The"essence" of the problem is described without any consideration of how a solution will be implemented.

▪ **Design Modeling Principles**

➢ The software design model is the equivalent of an architect's plans for a house. Set of principles used:

➢ **Design must be traceable to the analysis model**. The analysis model describes the information domain of the problem, user visible functions, system behavior, and a set of analysis classes that package business objects with the methods that service them.

- The design model translates this information into an architecture: a set of subsystems that implement major functions, and a set of component-level designs that are the realization of analysis class.
- **Always consider architecture.** S/W architecture is the skeleton of the system to be built. Only after the architecture is built should the component-level issues should be considered.
- **Focus on the design of data as it is as important as a design**. Data design is an essential element of t architectural design.
- **Interfaces (both user and internal) must be designed**. A well designed interface makes integration easier and assists the tester in validating component functions.
- **User interface design should be tuned to the needs of the end-user.** "Ease of use."
- **Component-level design should exhibit functional independence**. The functionality that is delivered by a component should be cohesive- that is, it should focus on one and only one function.
- **Components should be loosely coupled to one another and to the external environment**.
- Coupling is achieved in many ways – via a component interface, by messaging through global data. Coupling should be kept as low as is reasonable. As the level of coupling increases, error propagation also increases and the overall maintainability of the system decreases.
- **Design representation (models) should be easily understood**. **The design model should be developed iteratively. With each iteration, the designer should strive for greater simplicity**.

## 5.6 CONSTRUCTION PRACTICES

In this text "construction" is defined as being composed of both coding and testing. The purpose of testing is to uncover defects. Exhaustive testing is not possible so processing a few test cases successfully does not guarantee that you have bug free program. Unit testing of components and integration testing will be discussed in greater later in the text along with software quality assurance activities.
Although testing has received increased attention over the past decade, it is the weakest part of software engineering practice for most organizations.

### 5.6.1 Coding Principles and Concepts Preparation Principles:

Before _writing one line of code, be sure of:

- Understand the problem you are trying to solve.
- Understand the basic design principles.
- Pick a programming language that meets the needs of the S/W to be built and the environment in which it will operate.
- Select a programming environment that provides tool that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

**Coding Principles**: As you begin writing code, be sure you

- Constrain your algorithm by following structured programming practice.
- Select the proper data structure.
- Understand the software architecture.

- Keep conditional logic as simple as possible.
- Create easily tested nested loops.
- _Write code that is self-documenting.
- Create a visual layout.

**Validation Principles**: After you've completed your first coding pass, be sure you
- Conduct a code walkthrough.
- Perform unit test and correct errors.
- Refactor the code.

### 5.6.2 Testing Principles

- Testing is a process of executing a program with the intent of finding errors.
- A good test is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

1. **Testing Is the Process of Executing a Program with the Intent of Finding Errors**
    - The primary goal of software testing is to identify defects before the system is delivered to users.
    - Testing is not about proving that the software works perfectly; rather, it is about finding situations where it does not work as expected.
    - Effective testing involves executing the program under various conditions — normal, boundary, and exceptional — to reveal hidden errors.
    - It is an essential step in verifying that the software meets its specified requirements and performs reliably in real-world environments.
    - As Glenford J. Myers famously stated:
      "The purpose of testing is to show that a program has bugs, not that it works."
2. **A Good Test Is One That Has a High Probability of Finding an As-Yet Undiscovered Error**
    - A "good" test is strategically designed to challenge the software's logic, structure, and assumptions.
    - It focuses on areas most likely to contain defects — complex logic, boundary conditions, and user input handling.
    - Characteristics of a good test:
        - It is effective, detecting previously unknown errors.
        - It is efficient, avoiding redundancy.
        - It is traceable, linking back to specific requirements.
        - It is repeatable and reliable, producing consistent results.
    - Good tests are created through careful test case design techniques, such as:
        - Boundary Value Analysis (BVA)
        - Equivalence Partitioning
        - Error Guessing
        - Cause-Effect Graphing
3. **A Successful Test Is One That Uncovers an As-Yet-Undiscovered Error**
    - Many people mistakenly believe that a test is successful if it passes without finding any defects.In reality, a test is truly successful if it reveals a problem that was previously unknown.
    - Each discovered defect provides valuable feedback for improving software quality.Every error found early prevents more serious failures later in production.

- Hence, finding bugs should be seen as a positive outcome — not a failure of the developer but a success of the testing process.

| Concept | Meaning | Desired Outcome |
|---|---|---|
| Testing | Executing software to find errors | Identify defects early |
| Good Test | Has a high chance of revealing hidden defects | Maximizes fault detection |
| Successful Test | Actually finds an undiscovered error | Improves software quality |

## 5.7 DEPLOYMENT PRACTICES

**Customer Expectations for the software must be managed.** "Don't promise more than you can deliver."

- A complete delivery package should be assembled and tested.
- A support regime must be established before the software is delivered.
- Appropriate instructional materials must be provided to end-users.
- Buggy software should be fixed first, delivered later.

**Principles for Effective Software Delivery and Deployment**

1. **A Complete Delivery Package Should Be Assembled and Tested**
    - Before releasing any software product, it is essential to assemble a **complete delivery package** that includes not just the executable code but also all related components such as configuration files, libraries, documentation, and installation scripts.
    - The entire package should be tested as a whole to ensure that:
        - All required files are included.
        - The installation and deployment process works smoothly in the target environment.
        - Dependencies, versions, and configurations are properly managed.
    - This process helps prevent *"it works on my machine"* problems and ensures that users receive a functional, ready-to-use system.
    - **Key takeaway:** Deliver a cohesive, thoroughly validated package — not a collection of unverified parts.

2. **A Support Regime Must Be Established Before the Software Is Delivered**
    - Delivery does not end when the software is handed over; it marks the beginning of the **support and maintenance phase**.
    - A well-defined support structure ensures users have access to timely help for installation, configuration, troubleshooting, and updates.
    - The support regime should include:
        - Clear procedures for reporting bugs and issues.
        - Defined response and resolution times (Service Level Agreements, SLAs).
        - Maintenance schedules for patches and updates.
        - User communication channels (help desk, chat, or ticketing system).
    - Establishing this system **before delivery** ensures a smooth transition from development to operational use.
    - **Key principle:** Support readiness is as critical as product readiness.

3. **Appropriate Instructional Materials Must Be Provided to End-Users**
   - Even the best-designed software can fail in practice if users do not understand how to use it effectively.
   - Comprehensive and easy-to-understand **instructional materials** should accompany the delivery package.
   - These materials may include:
     - User manuals and quick-start guides.
     - Online help systems and FAQs.
     - Video tutorials or interactive walkthroughs.
     - Administrator and maintenance documentation for advanced users.
   - Instructional content should be tailored to the audience's skill level and the complexity of the system.
   - **Key idea:** Good documentation enhances usability, reduces support load, and increases user satisfaction.

4. **Buggy Software Should Be Fixed First, Delivered Later**
   - Rushing a product to delivery with known defects undermines credibility and increases long-term maintenance costs.
   - The priority should always be **to ensure stability and correctness before release**.
   - Releasing buggy software can lead to:
     - User dissatisfaction and loss of trust.
     - Costly patches and emergency fixes.
     - Reputational damage to the organization.
   - It is better to delay a release slightly to correct critical issues than to deliver an unstable product.
   - **Key message:** Quality must never be sacrificed for schedule pressure.

| **Principle** | **Core Focus** | **Key Outcome** |
|---|---|---|
| Complete Delivery Package | Deliver all components tested together | Reliable installation and operation |
| Support Regime | Prepare user support before delivery | Smooth post-deployment assistance |
| Instructional Materials | Provide user guidance and training | Better usability and reduced errors |
| Fix Before Delivery | Prioritize quality over deadlines | Stable, trustworthy software |

## 5.8 SUMMARY

Software Engineering Practice is a vital discipline that underpins the successful development and delivery of high-quality software systems. It integrates a range of principles, methods, and activities that guide teams from concept to deployment. Effective communication practices foster clarity, coordination, and alignment among stakeholders, ensuring that project goals are well-understood and consistently pursued. Planning practices provide the necessary structure, direction, and control to manage complexity, allocate resources efficiently, and mitigate risks. Modeling practices enable engineers to visualize, analyze, and refine system architecture before implementation, reducing design flaws early in the process. Construction practices emphasize disciplined coding, thorough testing, and adherence to quality standards, resulting in robust and maintainable systems. Finally, deployment practices ensure smooth release, seamless integration, and sustained system support through effective delivery and maintenance strategies. Together, these practices form a cohesive framework that promotes efficiency, reliability, maintainability, and user satisfaction—the hallmarks of successful software engineering projects.

## 5.9 TECHNICAL TERMS

software engineering practices, Communication, Planning, Modelling, Construction, Deployment

## 5.10 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. Discuss the key components of Software Engineering Practice
2. explain how they contribute to the successful development and deployment of software systems.

**Short questions:**

1. What are the main goals of Software Engineering Practice?
2. Why is effective communication important in software engineering projects?
3. What techniques are used in requirements gathering?
4. How do planning practices impact the success of a software project?
5. What is the role of modeling practices in software engineering?
6. What are some common modeling techniques used in software engineering?

## 5.11 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Kampa Lavanya

**LESSON- 06**

# SYSTEM ENGINEERING

**AIMS AND OBJECTIVES**

To introduce the principles of **System Engineering** and its role in the development of complex, computer-based systems; to explain the hierarchy of system elements, business process engineering concepts, and system modeling techniques used to analyze and design large-scale systems.

**After completing this lesson, you will be able to:**

1. Explain what **System Engineering** is and its importance in software development.
2. Describe the **System Engineering Hierarchy** and the components of a system.
3. Identify characteristics of **Computer-Based Systems**.
4. Understand the concept and phases of **Business Process Engineering (BPE)**.
5. Explain different **System Modeling techniques** used for analysis and design.
6. Relate system engineering activities to the overall **software engineering process**.
7. Appreciate the need for interdisciplinary collaboration in complex system development.

**STRUCTURE**

## 6.1 INTRODUCTION

Modern software products are rarely developed in isolation—they operate as part of larger, computer-based systems that include hardware, software, networks, data, and people. System Engineering is the process of defining, developing, and integrating these diverse components into a single, unified system that fulfills user needs and organizational objectives.

System engineering provides the "big picture" perspective—it ensures that all components of a system work together harmoniously, meeting both functional and non-functional requirements such as reliability, safety, and performance.

In short, system engineering focuses on what the system should do, how it should interact with its environment, and how it will be realized and maintained.

## 6.2 COMPUTER-BASED SYSTEMS

A Computer-Based System (CBS) is a complex arrangement of hardware, software, data, and human elements that interact to perform a specific function or achieve a particular goal.

**Components of a Computer-Based System:**

1. Software: Programs and procedures that control system behavior.
2. Hardware: Physical devices—computers, sensors, controllers, communication lines.
3. Data: Information that the system uses and produces.
4. People: Users, operators, and maintainers interacting with the system.
5. Procedures: Policies, rules, and workflows that govern system operation.
6. Documentation: Manuals, specifications, and technical guides.

**Examples of Computer-Based Systems:**

- Air traffic control systems.
- Online banking networks.
- E-commerce and inventory management systems.
- Hospital information management systems.
- Smart transportation and IoT-based control systems.

Each system integrates multiple components, and the success of the overall product depends on the proper functioning and coordination of each part.

## 6.3 SYSTEM ENGINEERING: DEFINITION AND SCOPE

System Engineering is a multidisciplinary process concerned with defining customer needs and required functionality early in the development cycle, documenting requirements, and then proceeding with design synthesis and system validation.

**It encompasses:**

- Requirements definition.
- System analysis and modeling.
- Architecture design.
- Integration and testing.
- Operation, maintenance, and evolution.

System engineering is not limited to software—it integrates hardware, people, and processes into a complete operational system.

**Goals of System Engineering**

- Define the system’s overall structure and behavior.
- Balance performance, cost, schedule, and risk.
- Manage complexity through hierarchical decomposition.
- Ensure interoperability between subsystems.
- Provide a foundation for subsequent software engineering.

## 6.4 THE SYSTEM ENGINEERING HIERARCHY

System engineering organizes the development process into hierarchical levels to manage complexity effectively.

**Levels in the System Engineering Hierarchy:**

| **Level** | **Description** |
|---|---|
| 1. World View (Context Level) | Defines the global context in which the system operates. Identifies external systems, users, and environmental interactions. |
| 2. Domain Level | Focuses on a specific business area or domain (e.g., healthcare, banking, transport). Establishes system boundaries. |
| 3. Element Level | Defines subsystems, their interfaces, and communication among components. |
| 4. Component Level | Describes the individual hardware and software units. This is where software engineering begins. |
| 5. Detailed Level | Deals with the design and implementation of specific algorithms, programs, and data structures. |

**Hierarchy Characteristics:**

- Each level refines and elaborates the one above.
- Feedback loops allow corrections when inconsistencies appear.
- Integration occurs progressively as elements are developed.

This hierarchical view ensures that system-level issues are resolved early, avoiding costly redesign during later stages.

## 6.5 SYSTEM ENGINEERING PROCESS

The system engineering process consists of a set of structured activities that transform user needs into a fully functioning system.

**Typical Phases:**

1. System Requirements Engineering: Identify functional and performance needs.
2. System Design: Develop high-level architecture and interfaces.
3. System Integration: Combine subsystems into a working whole.
4. System Verification and Validation: Ensure the system meets all requirements.
5. System Deployment and Maintenance: Install, monitor, and evolve the system.

System engineering thus creates a roadmap for software development, ensuring the software fits within a well-defined hardware and human context.

## 6.6 BUSINESS PROCESS ENGINEERING (BPE)

Business Process Engineering (BPE) is the systematic analysis, design, and optimization of business processes within an organization to align them with technology and strategic objectives.

It is sometimes referred to as Business Process Reengineering (BPR) when major transformations are undertaken.

**Goals of BPE**

- Improve process efficiency and quality.
- Eliminate redundant or non-value-adding activities.
- Introduce automation and IT support.
- Align business processes with customer needs and company strategy.

**Phases of Business Process Engineering**

| **Phase** | **Activities** |
|---|---|
| 1. Business Definition | Identify business goals and processes to be improved. |
| 2. Process Identification | Map current processes and workflows. |
| 3. Process Evaluation | Analyze weaknesses, inefficiencies, and redundancies. |
| 4. Process Redesign | Reconstruct processes using technology and best practices. |
| 5.Implementation and Integration | Deploy redesigned processes using information systems. |
| 6. Continuous Improvement | Monitor and refine processes over time. |

**Example:**
In an online retail company, BPE might involve redesigning the order fulfillment process by integrating inventory management, customer service, and shipping systems to reduce delivery time and cost.

## 6.7 RELATIONSHIP BETWEEN SYSTEM ENGINEERING AND BUSINESS PROCESS ENGINEERING

System engineering and business process engineering are complementary disciplines.

- BPE focuses on what the business does—the workflows, policies, and procedures.
- System Engineering focuses on how technology supports those business processes.

BPE identifies the requirements and goals, while System Engineering develops the technical systems that implement and optimize them.

Together, they form the foundation for enterprise-wide information systems and digital transformation.

## 6.8 SYSTEM MODELING

System Modeling is the process of creating abstract representations (models) of a system to analyze its structure, behavior, and interactions before implementation.
Models help in understanding complex systems and serve as a communication bridge between stakeholders.

**Types of System Models**

| **Model Type** | **Description** |
|---|---|
| Functional Model | Describes system functions and data flow (e.g., DFDs – Data Flow Diagrams). |

| | |
|---|---|
| Behavioral Model | Represents dynamic aspects using state diagrams or sequence diagrams. |
| Object-Oriented Model | Uses UML diagrams to describe classes, relationships, and interactions. |
| Physical Model | Illustrates system hardware layout, network connections, and deployment. |
| Mathematical/Analytical Model | Uses formulas or simulations for performance and reliability analysis. |

**Functional Model**

A Functional Model describes what the system does — the functions, transformations, and data flows within the system.

It focuses on processes and operations, showing how inputs are converted into outputs.
Explanation:
The functional model breaks down a system into smaller, manageable functions or modules that together achieve the overall goal.

It answers the question “What does the system do?” rather than “How does it do it?”.
The most commonly used technique for functional modeling is the Data Flow Diagram (DFD).
DFDs represent processes, data stores, data flows, and external entities. Each process is shown as a transformation that takes data inputs and produces outputs. At higher levels, DFDs give a broad view; at lower levels, they depict detailed operations.
**Example:**
In an Online Shopping System, a functional model might include:

- Process 1: *Receive Order*
- Process 2: *Verify Payment*
- Process 3: *Update Inventory*
- Process4:*GenerateInvoice*
  Each process shows how data (like order details, payment info) flows through the system.

**Use:**
Functional models are used during requirements analysis and early system design to visualize functionality, detect missing processes, and ensure logical completeness.

**Behavioral Model**

A Behavioral Model represents the dynamic behavior of a system — how it reacts to external or internal events over time.

It shows states, transitions, and events that change the system’s condition.

Explanation:
While functional models describe *what* the system does, behavioral models describe *when* and *under what conditions* those actions occur.

They focus on system states, events, and responses.

Common techniques include:

- State Transition Diagrams (STDs) – to represent state changes.
- Sequence Diagrams – to show message flow among system components.
- Activity Diagrams – to model concurrent actions and workflows.

**Example:**
In a Train Ticket Booking System, the states could include:

- “Idle” → “Seat Selection” → “Payment Processing” → “Ticket Confirmed.” Events such as *payment success* or *seat unavailability* trigger state changes.

**Use:**
Behavioral models are essential for real-time systems, interactive systems, and control systems, where timing and sequence of operations are critical.

**Object-Oriented Model**
An Object-Oriented Model (OOM) represents a system in terms of objects, classes, and their relationships.

It integrates both structure and behavior, focusing on data encapsulation and interaction between objects.

In this model, the system is viewed as a collection of objects that collaborate to perform functions.
Each object contains data (attributes) and operations (methods).

Relationships such as inheritance, association, and aggregation define how objects interact and depend on each other.

The standard notation for object-oriented modeling is the Unified Modeling Language (UML).
**Common diagrams include:**

- Class Diagrams – show classes, attributes, methods, and relationships.
- Use Case Diagrams – show interactions between users (actors) and the system.
- Sequence Diagrams – show the order of interactions among objects.

**Example:**
In a Library Management System, objects include:

- *Book*, *Member*, *Librarian*, *LoanRecord*.
- Each has attributes (BookID, Title, DueDate) and methods (issueBook(), returnBook()).
  The system’s functionality emerges from object interactions.

**Use:**
Object-oriented models are widely used in modern software engineering for analysis, design, and implementation, ensuring modularity, reusability, and scalability.
**Physical Model**

A Physical Model depicts the physical structure and deployment of a system, showing how software components, hardware devices, and networks are interconnected in the real world.
Explanation:

This model focuses on implementation and environment details, answering the question "Where and how will the system components operate?"
It typically includes:

- Hardware configurations (servers, routers, sensors).
- Communication links (network topology, bandwidth).
- Deployment of software modules across hardware nodes.
- Interfaces between physical components.

In UML, Deployment Diagrams and Component Diagrams are used to create physical models.

**Example:**
For an Online Banking System, the physical model may show:

- Client nodes (customer mobile apps and browsers).
- Application servers (running web services).
- Database servers (storing customer records).
- Network connections and firewalls ensuring secure communication.

**Use:**
Physical models are essential for system implementation, testing, and deployment planning, ensuring performance, scalability, and reliability.

**Mathematical / Analytical Model**
A Mathematical or Analytical Model represents system behavior and performance using mathematical equations, formulas, or algorithms.

It provides a quantitative way to analyze system characteristics such as reliability, throughput, and response time.
Unlike visual or descriptive models, mathematical models use formal notations to describe relationships and interactions precisely.

These models are especially useful for predicting system performance, optimizing design, and evaluating trade-offs before physical implementation.
Common mathematical modeling techniques include:

- Queuing Models – to analyze performance under different loads.
- Reliability Models – to estimate system failure rates.
- Control Theory Models – for dynamic systems like robotics.
- Simulation Models – for complex real-world scenarios.

**Example:**
In a Networked E-Commerce System, queuing theory can be used to model:

- The number of requests arriving at a server per second.
- The average waiting time for customers during peak load.
- This helps determine how many servers are required for smooth operation.

**Use:**
Mathematical models are applied during system optimization, performance evaluation, and risk assessment, providing a scientific basis for design decisions.

**Summary Table – Types of System Models**

| **Model Type** | **Focus Area** | **Main Tool / Technique** | **Example Application** |
|---|---|---|---|
| **Functional Model** | What the system does | Data Flow Diagrams (DFDs) | Order Processing System |
| **Behavioral Model** | How the system behaves over time | State or Sequence Diagrams | Ticket Booking Flow |
| **Object-Oriented Model** | Structure and interaction of objects | UML Class / Use Case Diagrams | Library or Banking System |
| **Physical Model** | Physical layout and deployment | Component / Deployment Diagrams | Distributed Database or Network System |
| **Mathematical / Analytical Model** | Quantitative performance analysis | Equations, Simulation, Queuing Models | Network Throughput or Reliability Estimation |

**Benefits of System Modeling**

- Improves understanding of system behavior.
- Helps identify missing or conflicting requirements.
- Serves as a blueprint for design and implementation.
- Supports risk analysis and cost estimation.

**Example:**
In a hospital management system, models may include:

- DFDs showing patient data flow between departments.
- State diagrams representing patient admission and discharge processes.
- UML class diagrams for patient, doctor, and billing entities.

**Data Flow Diagram – DFD**

The functional model defines *what* the hospital management system does by showing the movement and transformation of data between processes and entities.
In the HMS, a Level-0 DFD might include the following processes:

- *Patient Registration* – captures patient details and creates a new record.
- *Appointment Scheduling* – connects patients with available doctors.
- *Laboratory Testing* – manages test requests and stores results.
- *Billing and Accounts* – processes payments and generates invoices.
- *Report Generation* – summarizes patient statistics and hospital performance.

Data Stores: Patient Database, Doctor Database, Test Results, Billing Records. External Entities: Patients, Doctors, Pharmacy, Insurance Company.
This model shows how input data (e.g., admission details) flows through processes and generates output data (e.g., discharge summaries, bills).

**State Diagram**
The behavioral model captures how the system changes states in response to events over time.
It helps to understand dynamic behavior such as patient flow, record management, or emergency handling.
A Patient State Diagram could include the following states:

- *Registered → Admitted → Under Diagnosis → Under Treatment → Ready for Discharge → Discharged → Archived.*
- Events:
- *Admit Request* triggers transition from Registered to Admitted.
- *Doctor's Approval* moves the patient from Treatment to Ready for Discharge.
- *Bill Clearance* moves from Ready for Discharge to Discharged.

This model is critical for ensuring that all system transitions occur correctly and that no invalid operations (e.g., billing before treatment) are allowed.

**UML Class Diagram**
**Key Classes:**

- Patient – Attributes: patientID, name, address, dateOfAdmission.
  Methods: register(), discharge(), updateRecord().
- Doctor – Attributes: doctorID, specialization, schedule.
  Methods: assignPatient(), prescribeTreatment().
- Appointment – Attributes: appointmentID, date, time, status.
  Methods: schedule(), cancel().
- Billing – Attributes: invoiceNo, totalAmount, paymentStatus.
  Methods: generateInvoice(), updatePayment().
- LabTest – Attributes: testID, testName, result, status.
  Methods: recordResult(), verifyResult().

**Relationships:**

- A *Doctor* treats multiple *Patients*.
- A *Patient* may have multiple *Appointments*.
- Each *Patient* has one *BillingRecord*.
- *LabTest* is associated with *Patient* and *Doctor*.

This UML model defines the logical architecture of the HMS and serves as a direct foundation for coding and database design.

**Deployment and Network Architecture)**
The physical model illustrates *where and how* the hospital management system is deployed — showing the hardware, network, and system configuration.
A typical HMS physical model may include:

**Components:**

- **Client Layer:**
  • Workstations at reception, pharmacy, billing, and doctor's rooms.
  • Mobile/tablet interfaces for doctors and nurses.
- **Application Layer:**
  • Central hospital server hosting the HMS software.
  • Application services like patient management, billing, and reporting modules.
- **Database Layer:**
  • Dedicated database server storing patient records, test results, and billing data.
  • Backup server for redundancy and data recovery.
- **Network Infrastructure:**
  • LAN/Wi-Fi connecting departments.
  • Secure VPN access for external doctors and administrators.
  • Firewall and encryption for data protection.

**Example:**
When a receptionist registers a new patient, the workstation sends the data over the hospital LAN to the application server, which stores it in the central database and updates all connected departments in real time.
This model ensures system scalability, security, and reliability for 24×7 hospital operations.

**Analytical Model**
The mathematical model provides a quantitative framework to analyze system performance and reliability, helping engineers make informed design and optimization decisions.

**Applications in the Hospital Management System:**

1. **Performance Modeling (QueuingTheory) :**
   To estimate waiting times for patients during registration, diagnosis, or billing. Example: If patients arrive at the registration counter at an average rate of 12 per hour ($\lambda = 12$), and the clerk can process 15 patients per hour ($\mu = 15$), the model helps predict average queue length and waiting time.

2. **Reliability Modeling:**
   To estimate system uptime and downtime probabilities, ensuring that the HMS servers maintain 99.9% availability for critical medical services.

3. **Resource Optimization:**
   Mathematical models can determine the optimal number of lab technicians or billing clerks needed during peak hours to maintain efficiency.

4. **Cost-Benefit Analysis:**
   Analytical models calculate trade-offs between adding new servers (cost) and reducing patient waiting time (benefit).

**Example Calculation:**

A queuing model might predict that adding one extra registration counter reduces average waiting time from 10 minutes to 4 minutes — a 60% improvement in service quality.

**Integrated View**
Each model offers a unique but complementary view of the hospital system:

| **Model Type** | **Focus** | **Example in HMS** |
|---|---|---|
| **Functional Model** | **Logical flow of data and processes** | **DFD showing patient data from registration to billing** |
| **Behavioral Model** | **Dynamic state transitions** | **State diagram for patient admission and discharge** |
| **Object-Oriented Model** | **Classes and relationships** | **UML diagram with Patient, Doctor, Billing** |
| **Physical Model** | **Hardware and deployment architecture** | **Network setup linking clients, servers, and databases** |
| **Mathematical Model** | **Quantitative analysis** | **Queuing model for registration counter performance** |

**By integrating these models, system engineers can:**

- Understand both functional and technical aspects of the system.
- Validate design decisions before implementation.
- Anticipate performance issues and optimize workflows.
- Ensure alignment with real-world hospital operations.

## 6.9 CHALLENGES IN SYSTEM ENGINEERING

- Managing complexity across disciplines.
- Ensuring interoperability among heterogeneous systems.
- Handling evolving requirements and change management.
- Achieving balance between performance, cost, and reliability.
- Maintaining traceability from business goals to system components.
- Coordinating communication among large, multidisciplinary teams.

Effective system engineering requires strong project management, technical expertise, and collaboration between business analysts, engineers, and stakeholders.

## 6.10 SUMMARY

System Engineering provides the framework for developing complex computer-based systems that integrate hardware, software, people, and processes.
It defines the hierarchy of system elements and coordinates activities from requirements through operation.
Business Process Engineering complements this by aligning organizational workflows with technology to improve efficiency and effectiveness.
System Modeling, in turn, provides visual and analytical representations that help engineers understand, communicate, and design systems systematically.
Together, these disciplines ensure that modern information systems are technically sound, economically viable, and aligned with business goals.

## 6.11 TECHNICAL TERMS

1. System Engineering
2. Computer-Based System
3. System Engineering Hierarchy
4. Business Process Engineering (BPE)
5. System Modeling
6. Functional Model
7. Behavioral Model
8. Object-Oriented Model
9. Integration and Verification
10. Continuous Improvement

## 6.12 SELF-ASSESSMENT QUESTIONS

**Essay Questions**

1. Define System Engineering and explain its importance in modern software development.
2. Discuss the key components of a Computer-Based System with examples.
3. Explain the System Engineering Hierarchy and its significance.
4. What are the major phases of the System Engineering process?

5. Describe Business Process Engineering (BPE) and its stages.
6. How are System Engineering and Business Process Engineering related?
7. Define System Modeling and describe its various types.
8. Explain how modeling helps in analyzing and designing complex systems.
9. What challenges are commonly faced in System Engineering?
10. Discuss the role of System Engineering in the development of an e-commerce platform.

**Short Questions**

1. World View Level
2. Element Level
3. Process Redesign in BPE
4. Data Flow Diagrams (DFDs)
5. Behavioral Modeling
6. System Integration
7. Continuous Process Improvement
8. System Verification and Validation
9. System Hierarchy Feedback Loops
10. Role of People in Computer-Based Systems

## 6.13 SUGGESTED READINGS

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 7th Ed., 2014.
2. Sommerville, Ian, *Software Engineering*, 10th Ed., Pearson Education, 2015.
3. Blanchard, Benjamin S. & Fabrycky, Wolter J., *Systems Engineering and Analysis*, Prentice Hall, 2011.
4. Dennis, Alan et al., *Systems Analysis and Design*, Wiley, 7th Ed., 2019.
5. Hoffer, George, & Valacich, *Modern Systems Analysis and Design*, Pearson, 2017.
6. Marakas, George, *Systems Analysis and Design: An Active Approach*, McGraw-Hill, 2006.
7. Checkland, Peter, *Systems Thinking, Systems Practice*, Wiley, 1999.
8. Yourdon, Edward, *Modern Structured Analysis*, Prentice Hall, 1998.
9. Avison, David & Fitzgerald, Guy, *Information Systems Development*, McGraw-Hill, 2006.
10. Martin, James, *Systems Analysis for Business Data Processing*, Prentice Hall, 1995.

Dr. Kampa Lavanya

LESSON- 07

# BUILDING THE ANALYSIS MODEL

## AIMS AND OBJECTIVES

The primary goal of this chapter is to understand Building the Analysis Model. The chapter began with understand of Building the Analysis Model: Requirement Analysis, Analysis Modeling Approaches, Data Modeling Concepts, Object Oriented Analysis, Scenario Based Modeling, Flow Oriented Modeling, Class Based Modeling, Creating a Behavioral Model. After completing this chapter, the student will understand the complete knowledge about Building the Analysis Model.

## STRUCTURE

## 7.1 INTRODUCTION

Building the Analysis Model is a critical phase in software engineering that bridges the gap between user requirements and system design. It begins with requirement analysis, where user needs are meticulously gathered and documented. Various analysis modeling approaches are employed to create abstract representations of the system. Data modeling concepts focus on structuring and organizing data, while object-oriented analysis emphasizes defining system objects and their interactions. Scenario-based modeling uses real-world scenarios to illustrate system functions, and flow-oriented modeling maps out data movement within the system. Class-based modeling organizes system components into classes with shared attributes and behaviors. Finally, creating a behavioral model captures the dynamic aspects of the system, detailing how it responds to events and interacts with users and other systems. This comprehensive analysis ensures a clear, detailed understanding of the system's requirements and functionalities, guiding the subsequent design and development phases.

## 7.2 REQUIREMENT ANALYSIS

Requirements analysis results in the specification of software’s operational characteristics, indicates software’s interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

- Scenario-based models of requirements from the point of view of various system "actors"
- Data models that depict the information domain for the problem
- Class-oriented models that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- Flow-oriented models that represent the functional elements of the system and how they transform data as it moves through the system
- Behavioral models that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

Throughout requirements modeling, primary focus is on what, not how. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

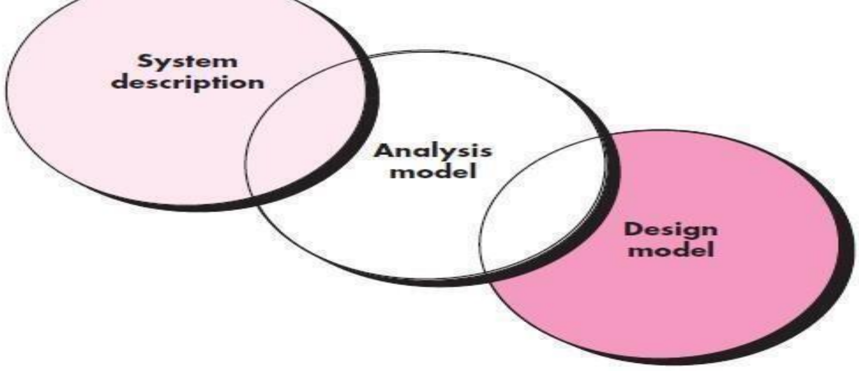


Fig 7.1 The requirements model as a bridge between the system description and the design model

**The requirements model must achieve three primary objectives:**

- To describe what the customer requires,
- to establish a basis for the creation of a software design, and
- to define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure.

**Analysis Rules of Thumb**

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- Minimize coupling throughout the system. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.
- Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model
- Keep the model as simple as it can be. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

**Domain Analysis**

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides.

The "specific application domain" can range from avionics to banking, frommultimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to identify common problem solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

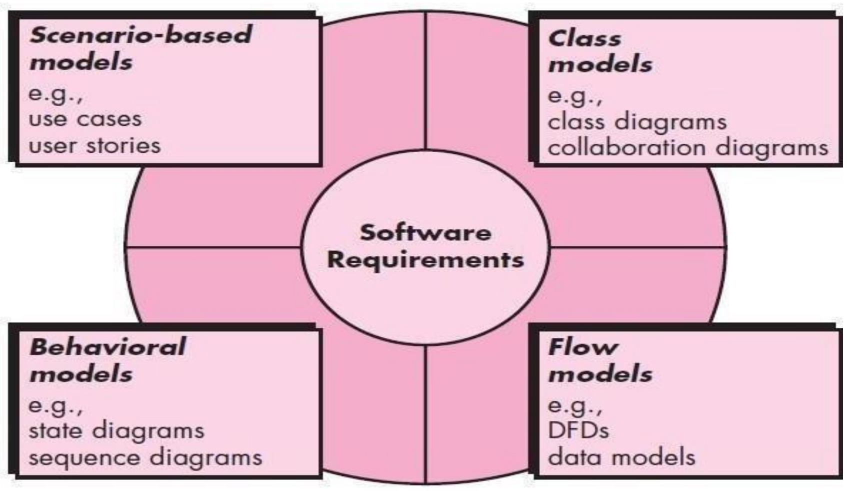


Fig 7.2 The requirements model as a bridge between the system description and the design model

**Requirements Modeling Approaches**

One view of requirements modeling, called structured analysis, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their attributes and relationships.

A second approach to analysis modeling, called object-oriented analysis, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements. UML and the Unified Process are predominantly objectoriented.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

**Scenario-based elements** depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

**Class-based elements** model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.

**Behavioral elements** depict how external events change the state of the system or the classes that reside within it. Finally,

**Flow-oriented elements** represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

## 7.3 SCENARIO-BASED MODELING

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

### Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a "contract for behavior", the "contract" defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself.
A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor.

These are the questions that must be answered if use cases are to provide value as a requirements modeling tool.

1) what to write about,
2) how much to write about it,
3) (3)how detailed to make your description, and
4) how to organize the description?

To begin developing a set of use cases, list the functions or activities performed by a specific actor.

### Refining a Preliminary Use Case

Each step in the primary scenario is evaluated by asking the following questions:

- Can the actor take some other action at this point?
- Is it possible that the actor will encounter some error condition at this point? If so, what might it be?
- Is it possible that the actor will encounter some other behavior at this point (e.g.,behavior that is invoked by some event outside the actor's control)? If so, what might it be?

Cockburn recommends using a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the three generic questions suggested earlier in this section, the following issues should also be explored:

- Are there cases in which some “validation function” occurs during this use case? This implies that validation function is invoked and a potential error condition might occur.
- Are there cases in which a supporting function (or actor) will fail to respond appropriately? For example, a user action awaits a response but the function that is to respond times out.
- Can poor system performance result in unexpected or improper user actions? For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

**Writing a Formal Use Case**

The typical outline for formal use cases can be in following manner

- The goal in context identifies the overall scope of the use case.
- The precondition describes what is known to be true before the use case is initiated.
- The trigger identifies the event or condition that “gets the use case started”
- The scenario lists the specific actions that are required by the actor and the appropriate system responses.
- Exceptions identify the situations uncovered as the preliminary use case is refined Additional headings may or may not be included and are reasonably self-explanatory.

Every modeling notation has limitations, and the use case is no exception. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements.

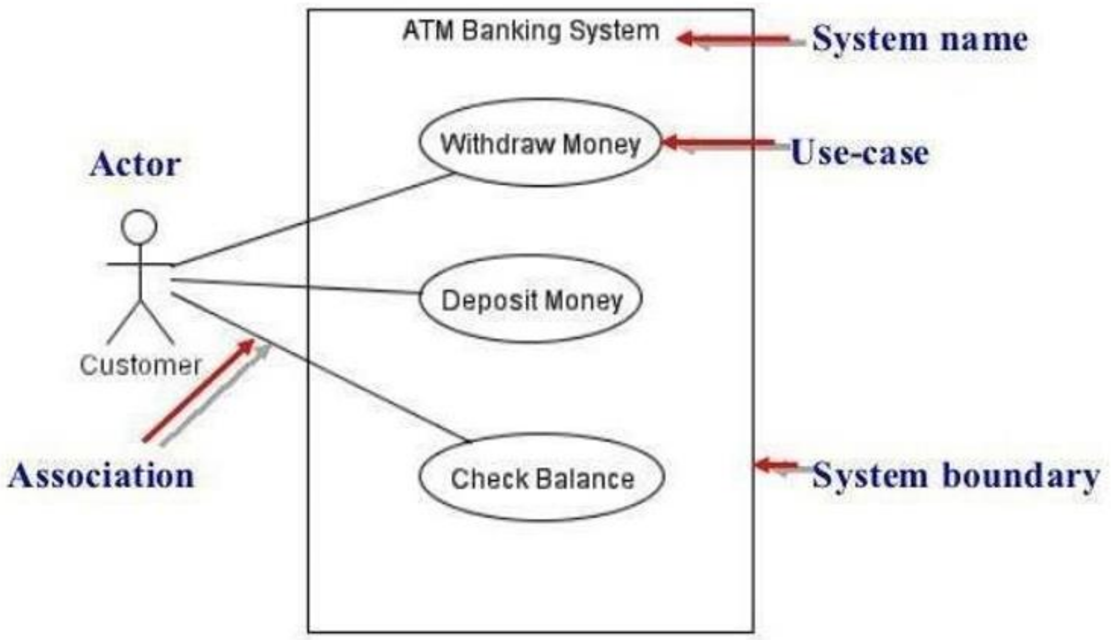


**Fig 7.3 Simple Use-case Diagram**

However, scenario-based modeling is appropriate for a significant majority of all situationsthat you will encounter as a software engineer.

**UML MODELS THAT SUPPLEMENT THE USE CASE**

- **Developing an Activity Diagram**

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through thesystem, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. i.e A UML activity diagram represents the actions and decisions that occur as some function is performed.

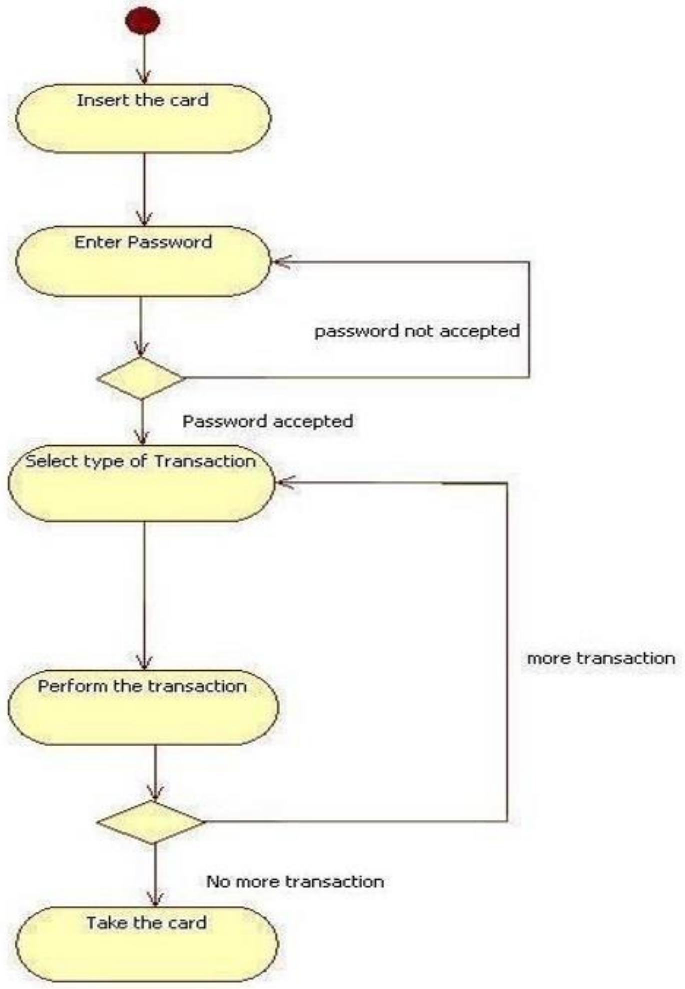


Fig 7.4 Activity Diagram for ATM

❖ **Swimlane Diagrams**

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle. Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

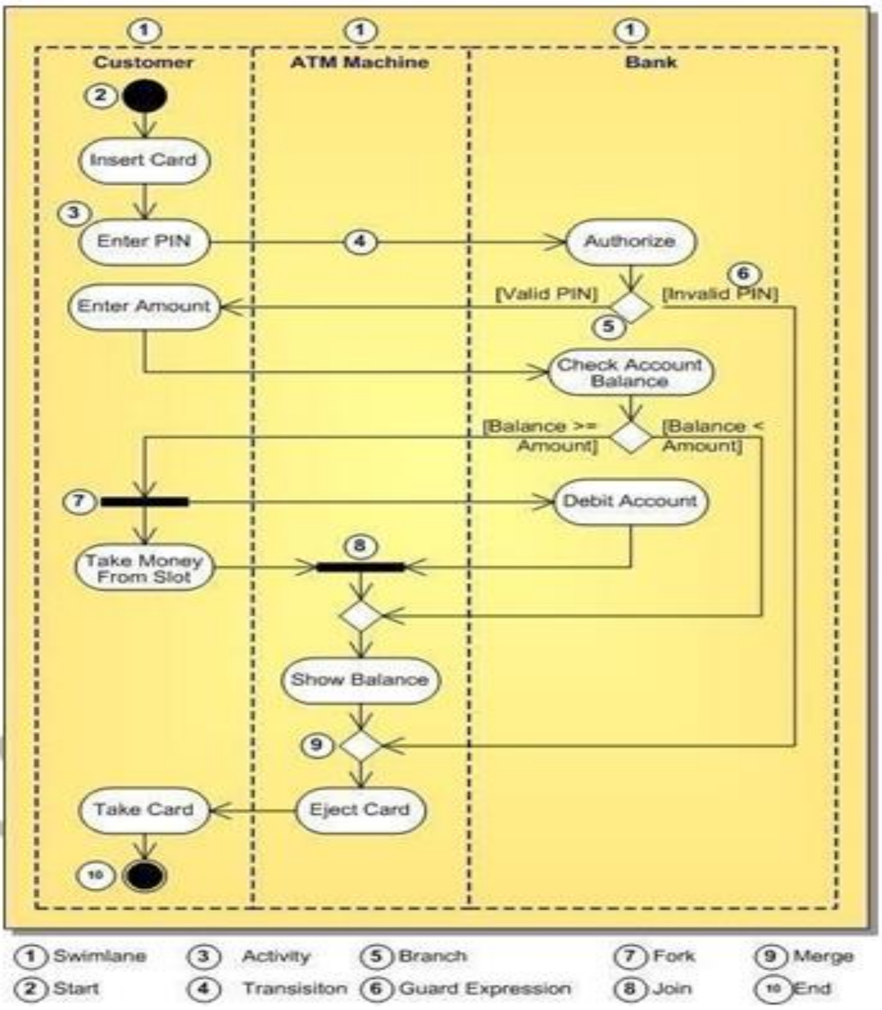


Fig 7.5 *swimlane diagram for ATM*

## 7.4 DATA MODELING CONCEPTS

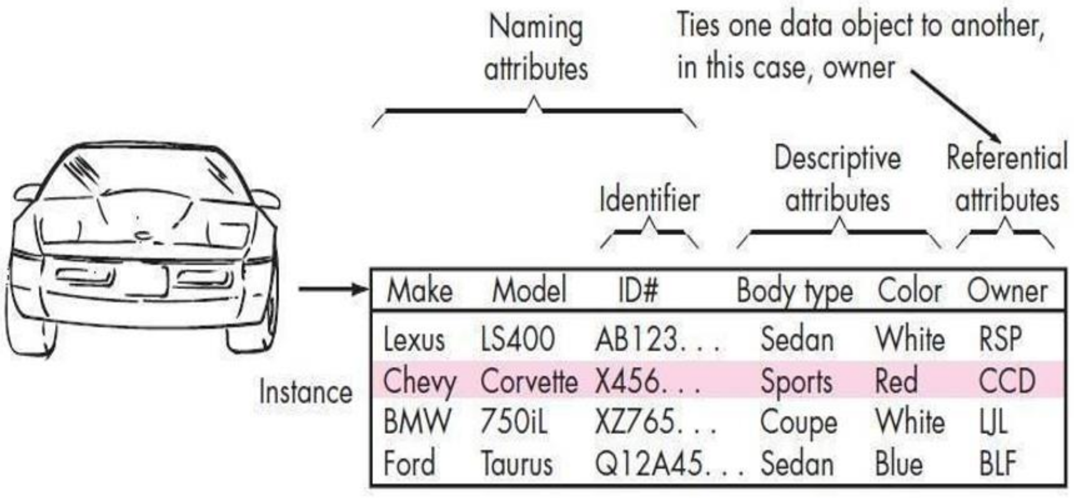
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most widely used data Model by the Software engineers is **E*ntity-Relationship Diagram*** **(ERD)**, it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

### Data Objects

A *data object* is a representation of composite information that must be understood by software. A data object can be an **external entity** (e.g., anything that produces or consumes information), **a thing** (e.g., a report or a display), **an occurrence** (e.g., a telephone call) or **event** (e.g., an alarm), **a role** (e.g., salesperson), **an organizational unit** (e.g., accounting department), **a place** (e.g., a warehouse), **or a structure** (e.g., afile).

For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.

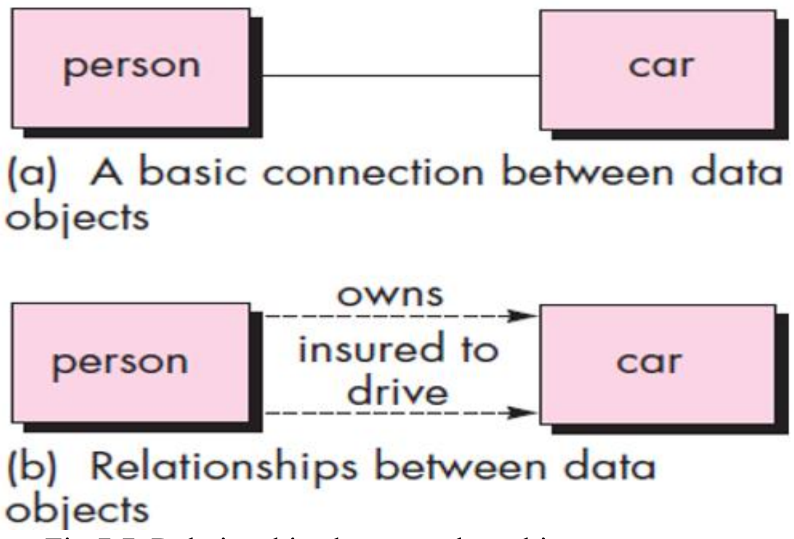


| Make | Model | ID# | Body type | Color | Owner |
|---|---|---|---|---|---|
| Lexus | LS400 | AB123. . . | Sedan | White | RSP |
| Chevy | Corvette | X456. . . | Sports | Red | CCD |
| BMW | 750iL | XZ765. . . | Coupe | White | LJL |
| Ford | Taurus | Q12A45. . . | Sedan | Blue | BLF |

Fig 7.6 Tabular representation of data objects

**Data Attributes**
*Data attributes* define the properties of a data object and take on one of **three** different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table.

**Relationships**
Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the following simple notation and relationships are 1) A person *owns* a car, 2) A person *is insured to drive* a car


person
car
(a) A basic connection between data objects
owns
person
insured to drive
car
(b) Relationships between data objects
Fig 7.7 Relationships between data objects

**7.5 CLASS-BASED MODELING**
Class-based modeling represents the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between theobjects,

and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class responsibility-collaborator (CRC) models, collaboration diagrams, and packages.

**Identifying Analysis Classes**

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "**grammatical parse**" on the use cases developed for the system to be built.
*Analysis classes* manifest themselves in one of the following ways:

- ***External entities*** (e.g., other systems, devices, people) that produce orconsume information to be used by a computer-based system.
- ***Things*** (e.g., reports, displays, letters, signals) that are part of the information domain forthe problem.
- ***Occurrences or events*** (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- ***Roles*** (e.g., manager, engineer, salesperson) played by people who interact withthe system.
- ***Organizational units*** (e.g., division, group, team) that are relevant to an application.
- ***Places*** (e.g., manufacturing floor or loading dock) that establish the context ofthe problem and the overall function of the system.
- ***Structures*** (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Coad and Yourdon suggest **six** selection characteristics that should be used as you consider each potential class for inclusion in the **analysis model**:

1. ***Retained information***. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. ***Needed services***. The potential class must have a set of identifiable operations that can changethe value of its attributes in some way.
3. ***Multiple attributes***. During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. ***Common attributes***. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
5. ***Common operations***. A set of operations can be defined for the potential class and these operations apply to all instances of the class.
6. ***Essential requirements***. External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirements model.

**Specifying Attributes**

*Attributes* describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

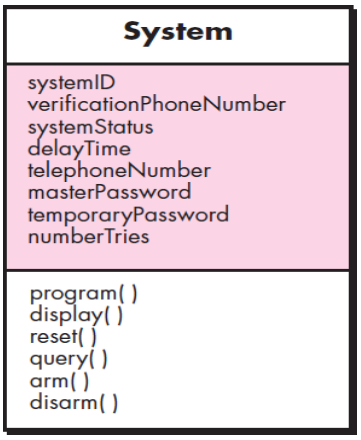
To develop a meaningful set of attributes for an analysis class, you should study each usecase and select those "things" that reasonably "belong" to the class.

**Defining Operations**

*Operations* define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event

**Fig : Class diagram for the system class**

**Class-Responsibility-Collaborator (CRC) Modeling**

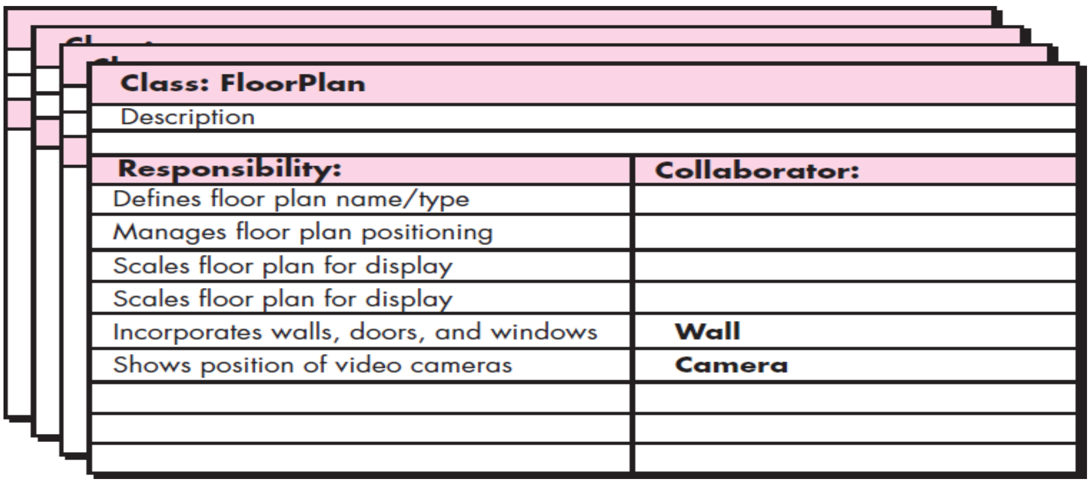


*Class-responsibility-collaborator (CRC) modeling* provides a simple means for identifying and organizing the classes that are relevant to system or product requirements Ambler describes CRC modeling in the following way :

A CRC model is really a collection of standard **index cards** that represent classes. The cards are divided into **three** sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the **left** and the collaborators on the **right**.

The CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. ***Responsibilities*** are the attributes and operations that are relevant for the class. i.e., a responsibility is "anything the class knows or does" ***Collaborators*** are those classes that are required to provide a class with the information needed to complete a responsibility.

**Fig : A CRC model index card**


Class: FloorPlan
Description
Responsibility:
Collaborator:
Defines floor plan name/type
Manages floor plan positioning
Scales floor plan for display
Scales floor plan for display
Incorporates walls, doors, and windows
Wall
Shows position of video cameras
Camera

**Classes:** The taxonomy of class types can be extended by considering the following categories:

- ***Entity classes***, also called ***model* or *business*** classes, are extracted directly from the statement of the problem. These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.
- ***Boundary classes*** are used to create the interface that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- ***Controller classes*** manage a "unit of work" from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

**Responsibilities:** Wirfs-Brock and her colleagues suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do.
2. **Each responsibility should be stated as generally as possible.** This guidelineimplies that general responsibilities should reside high in the class hierarchy
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes thatmanipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

**Collaborations.** Classes fulfill their responsibilities in one of **two** ways:

1. A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
2. A class can collaborate with other classes.
3. When a complete CRC model has been developed, stakeholders can review the model using the following approach :
4. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have twocards that collaborate).
5. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
6. The review leader reads the use case deliberately. As the review leader comes to a namedobject, she passes a token to the person holding the corresponding class index card.

7. When the token is passed, the holder of the card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
8. If the responsibilities and collaborations noted on the index cards cannot accommodate the use case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

**Associations and Dependencies**

An ***association*** defines a relationship between classes. An association may be further defined by indicating ***multiplicity***. **Multiplicity** defines how many of one class are related to how many of another class.

A client-server relationship exists between two analysis classes. In such cases, a client class depends on the server class in some way and a ***dependency relationship*** is established. Dependencies are defined by a **stereotype**. A ***stereotype*** is an "**extensibility mechanism**" within UML that allows you to define a special modeling element whose semantics are custom defined. In UML. Stereotypes are represented in double angle brackets (e.g., **<<stereotype>>**).

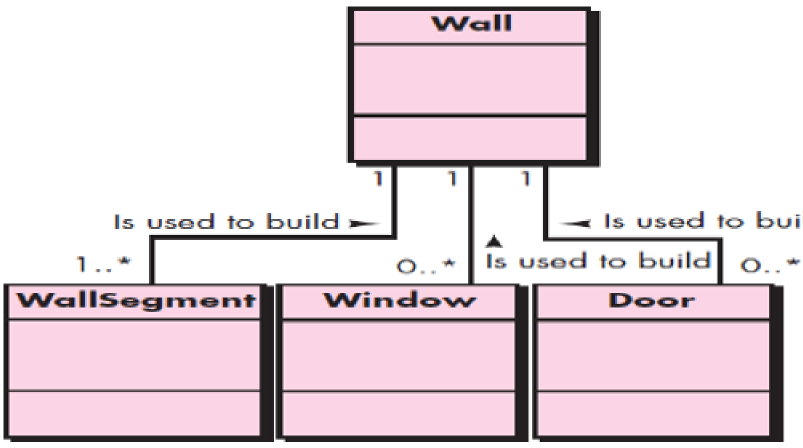


**Fig : Multiplicity**

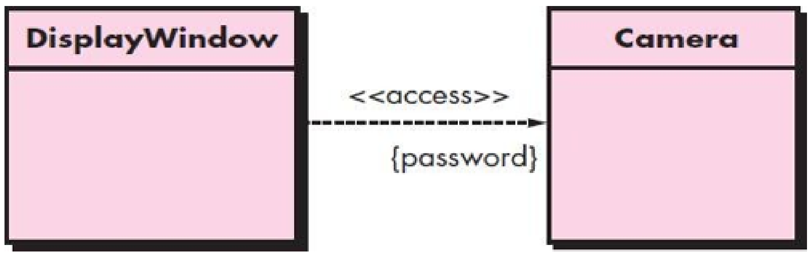
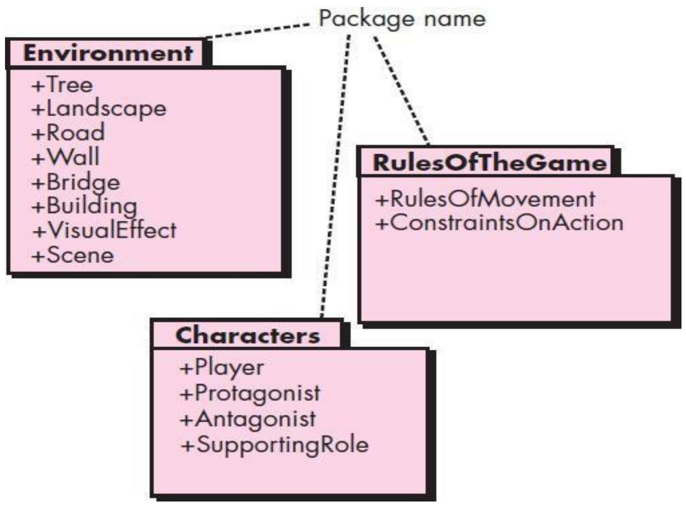


**Fig : Dependencies**

**Analysis Packages:**

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

## 7.6 FLOW-ORIENTED MODELING

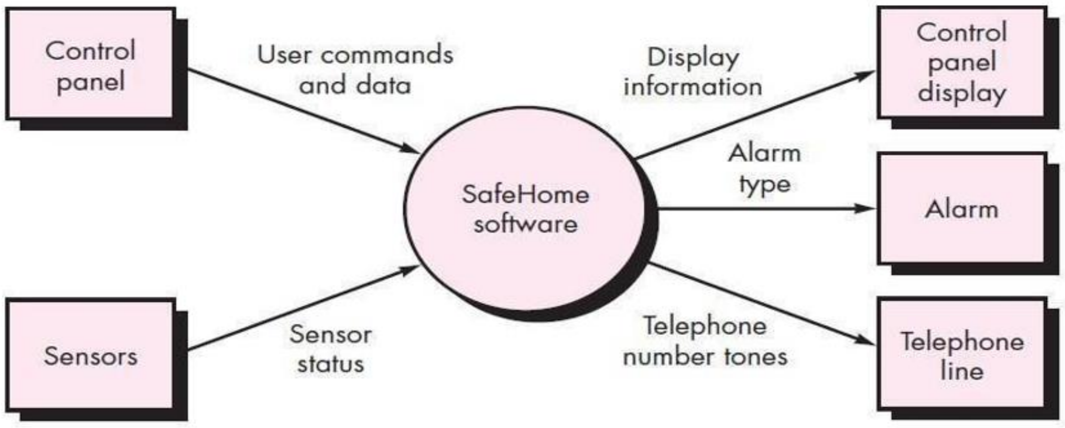


One view of requirements modeling, called ***structured analysis***,. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow throughthe system.

A second approach to analysis modeled, called ***object-oriented analysis,*** focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

Flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today. The data flow diagram (DFD) is the representation of Flow-oriented modeling. The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers."

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled arrows, and transformations are represented by circles (also called bubbles). The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level 0 DFD or context diagram) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.



**Creating a Data Flow Model**

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram:

1. The level 0 data flow diagram should depict the software/system as a single bubble;
2. Primary input and output should be carefully noted;
3. Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level;
4. All arrows and bubbles should be labeled with meaningful names;
5. Information flow continuity must be maintained from level to level,2 and
6. One bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

A level 0 DFD for the security function is shown in above figure. The primary external entities (boxes) produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies.

The level 0 DFD must now be expanded into a level 1 data flow model. you should apply a "grammatical parse" to the use case narrative that describes the context-level bubble. That is, isolate all nouns (and noun phrases) and verbs (and verb phrases). The grammatical parse is not foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

The processes represented at DFD level 1 can be further refined into lower levels. The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. a concept, Cohesion can be used to assess the processing focus of a given function. i.e refine DFDs until each bubble is "single-minded."

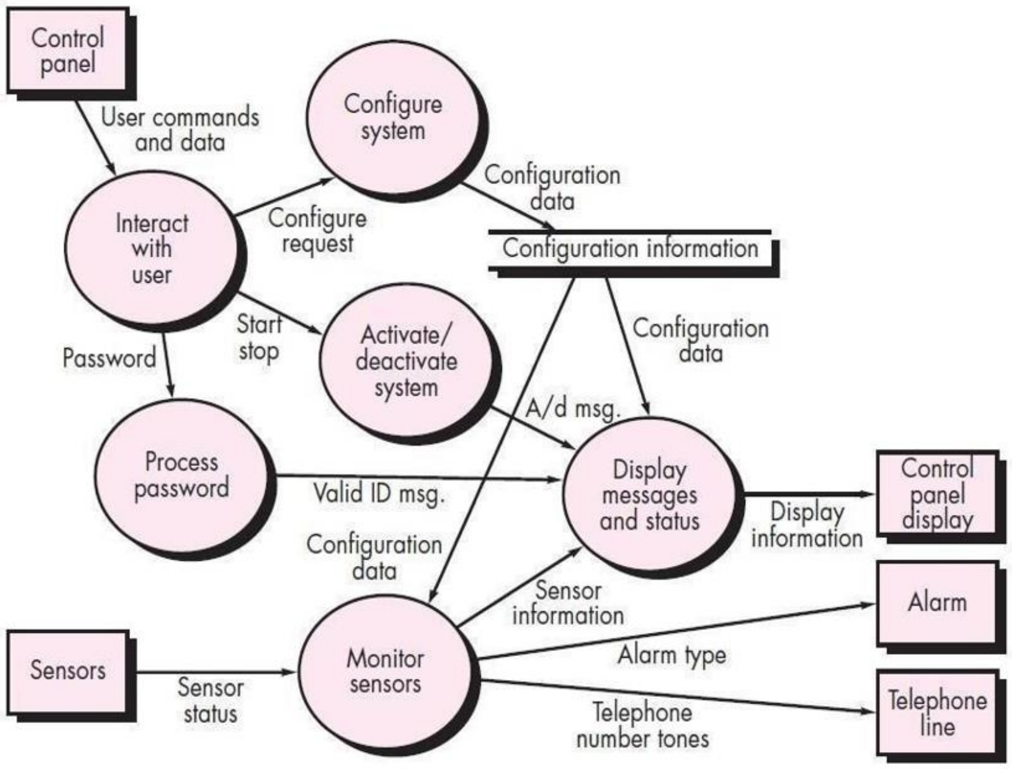


**Fig: Level 1 DFD for SafeHome security function**

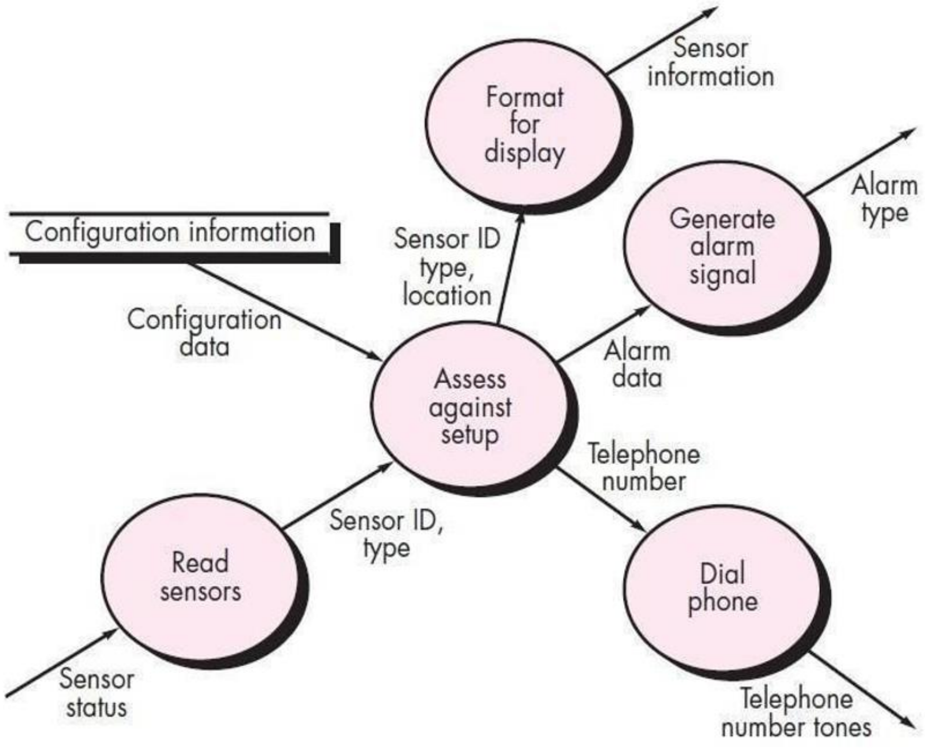


**Fig : Level 2 DFD that refines the monitor sensors process**

**Creating a Control Flow Model**

The data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements.

The following guidelines are suggested for creating a Control Flow Model

- List all sensors that are "read" by the software.
- List all interrupt conditions.
- List all "switches" that are actuated by an operator.
- List all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible control specification inputs/outputs.
- Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
- Focus on possible omissions—a very common error in specifying control;

**The Control Specification**

A *control specification* (CSPEC) represents the behavior of the system in **two** different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. The following figure depicts a preliminary state diagram for the level 1 control flow model for *SafeHome.* The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, we can determine the behavior of the system and, more important, ascertain whether there are "holes" in the specifiedbehavior.
The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.

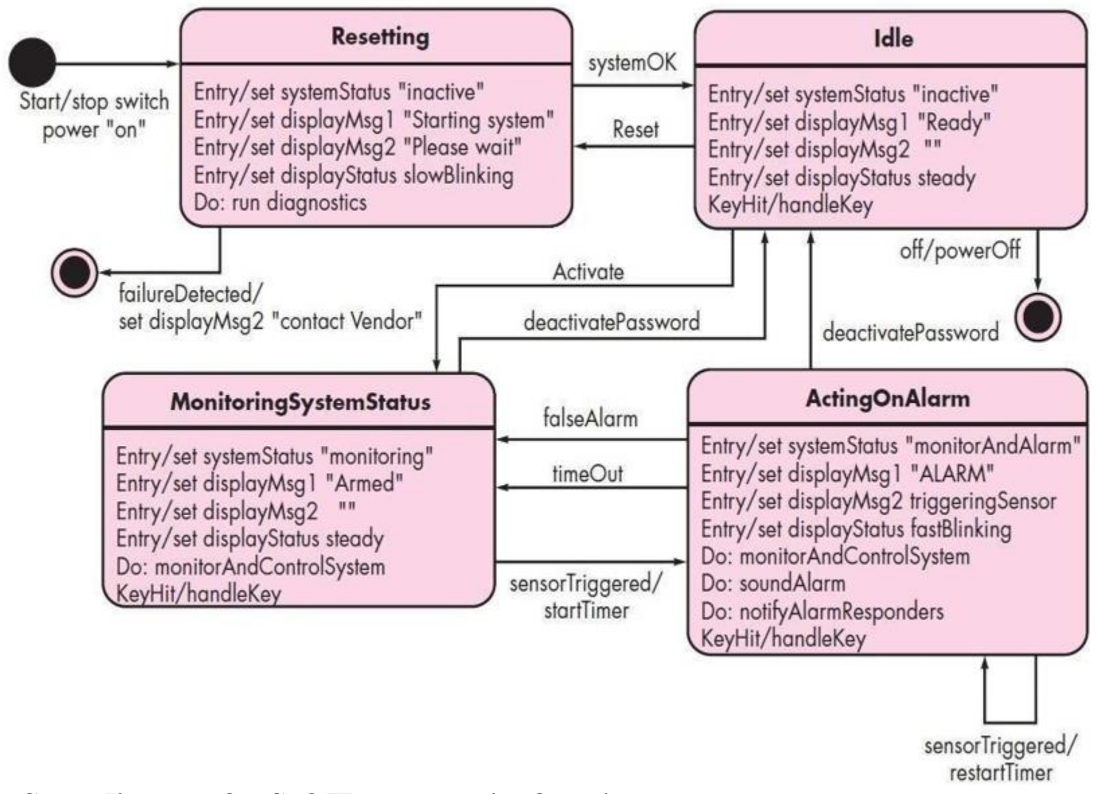

Resetting
Entry/set systemStatus "inactive"
Entry/set displayMsg1 "Starting system"
Entry/set displayMsg2 "Please wait"
Entry/set displayStatus slowBlinking
Do: run diagnostics
Start/stop switch power "on"
systemOK
Reset
Idle
Entry/set systemStatus "inactive"
Entry/set displayMsg1 "Ready"
Entry/set displayMsg2 ""
Entry/set displayStatus steady
KeyHit/handleKey
failureDetected/ set displayMsg2 "contact Vendor"
Activate
deactivatePassword
off/powerOff
deactivatePassword
MonitoringSystemStatus
Entry/set systemStatus "monitoring"
Entry/set displayMsg1 "Armed"
Entry/set displayMsg2 ""
Entry/set displayStatus steady
Do: monitorAndControlSystem
KeyHit/handleKey
falseAlarm
timeOut
sensorTriggered/ startTimer
ActingOnAlarm
Entry/set systemStatus "monitorAndAlarm"
Entry/set displayMsg1 "ALARM"
Entry/set displayMsg2 triggeringSensor
Entry/set displayStatus fastBlinking
Do: monitorAndControlSystem
Do: soundAlarm
Do: notifyAlarmResponders
KeyHit/handleKey
sensorTriggered/ restartTimer

**Fig : State diagram for SafeHome security function**

**The Process Specification**

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm, mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a "mini-spec" that serves as a guide for design of the software component that will implement the bubble.

## 7.7 CREATING A BEHAVIORAL MODEL

The *behavioral model* indicates how software will respond to external events or stimuli.
To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction withinthesystem.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create a sequence for each use case.
4. Build a state diagram for the system.
5. Review the behavioral model to verify accuracy and consistency.

**Identifying Events with the Use Case**

The use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function. The homeowner uses the keypad to key in a four-digitpassword. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events .

**State Representations**

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its Function **Two** different behavioral representations are discussed in the paragraphs that follow. The **first** indicates how an individual class changes state based on external events and the second shows the behavior of the software as a function of time.

**State diagrams for analysis classes.** One component of a behavioral model is a UML state diagram that represents active states for each class and the events (triggers) that cause changes between these active states. The following figure illustrates a state diagram for the **ControlPanel** object in the *SafeHome* security function. Each arrow shown in figure represents a transition from one active state of an object to another. The labels shown for each arrow represent the eventthat triggers the transition

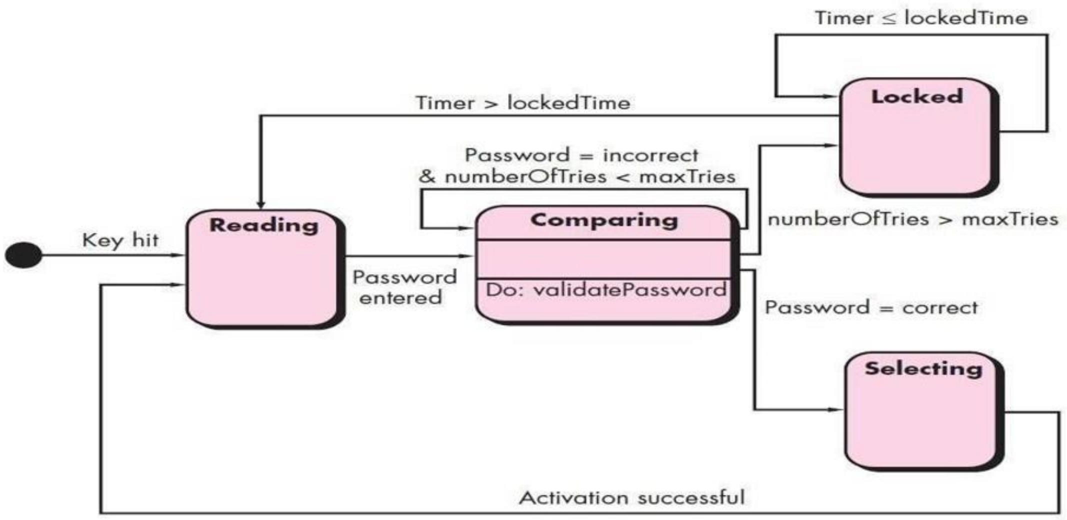


Fig : State diagram for the Control Panel class

Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, thesequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

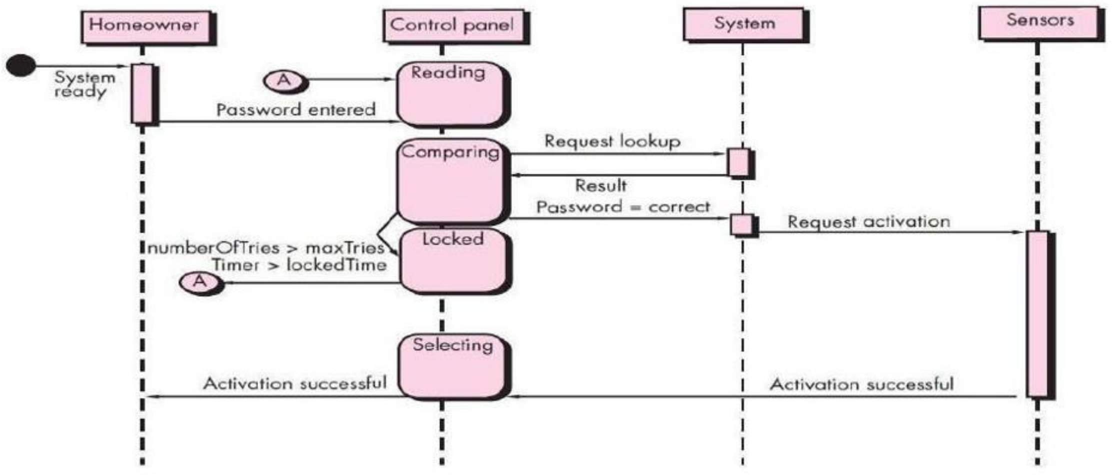


Fig : Sequence diagram (partial) for the *SafeHome* security function

## 7.8 SUMMARY

In conclusion, building the analysis model is a pivotal step in software development that transforms user requirements into a detailed blueprint for system design. Through requirement analysis, the foundational needs of users are identified and articulated. Diverse analysis modeling approaches provide structured frameworks to represent system components and their relationships. Data modeling concepts ensure that data is logically organized and accessible. Object-oriented analysis delineates system objects and their interactions, fostering a modular and reusable design. Scenario-based modeling leverages real-world examples to clarify system behavior, while flow-oriented modeling illustrates the movement of data within the system. Class-based modeling structures the system into cohesive classes with shared attributes and methods. Finally, creating a behavioral model captures the dynamic behavior of the system, detailing how it responds to various events and interacts with users and other systems. Collectively, these modeling techniques create a comprehensive and precise analysis model, laying a solid foundation for the design and implementation phases of software development.

## 7.9 TECHNICAL TERMS

Building Analysis Mode,Object Oriented Analysis,Flow Oriented Analysis,Clas bases analysis,scenario based analysis.

## 7.10 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. What is the primary objective of requirement analysis in building the analysis model?
2. How do analysis modeling approaches facilitate the transition from user requirements to system design?
3. What are data modeling concepts and why are they important in software engineering?
4. Describe the main focus of object-oriented analysis.

**Short questions:**

1. What is scenario-based modeling and how does it help in understanding system requirements?
2. How does flow-oriented modeling differ from scenario-based modeling?
3. Explain the role of class-based modeling in organizing system components.
4. What is a behavioral model and why is it crucial in the analysis phase?
5. How do these various modeling techniques interrelate to form a comprehensive analysis model?
6. Why is it important to have a detailed analysis model before proceeding to the design and implementation phases of software development?

## 7.11 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Kampa Lavanya

# LESSON- 09
# TESTING STRATEGIES

## AIMS AND OBJECTIVES

The primary goal of this chapter is to understand testing stragegies. The chapter began with understand of **Testing Strategies:** A strategic Approach to Software Testing, Strategic Issues, and Test Strategies for conventional Software, Testing Strategies for Object Oriented Software, Validation Testing, System Testing, the Art of Debugging.. After completing this chapter, the student will understand the complete knowledge about testing strategies.

## STRUCTURE

## 9.1 INTRODUCTION

Testing strategies are essential for ensuring the quality and reliability of software systems. A strategic approach to software testing involves planned and systematic testing activities that aim to identify defects and verify that the software meets its requirements. Strategic issues include determining the scope, objectives, resources, and schedule for testing. Test strategies for conventional software typically involve unit testing, integration testing, system testing, and acceptance testing, while testing strategies for object-oriented software focus on testing objects, classes, and their interactions. Validation testing ensures that the software fulfills user needs and requirements, whereas system testing evaluates the system's compliance with specified requirements. The art of debugging involves identifying, isolating, and fixing defects in the software. Together, these strategies and practices help in delivering high-quality software that performs reliably in real-world scenarios.

**Principles of Testing:-**

(i) All the test should meet the customer requirements
(ii) To make our software testing should be performed by third party
(iii) Exhaustive testing is not possible.As we need the optimal amount of testing based on the risk assessment of the application.
(iv) All the test to be conducted should be planned before implementing it

(v) It follows pareto rule(80/20 rule) which states that 80% of errors comes from 20% of program components.
(vi) Start testing with small parts and extend it to large parts.

## 9.2 TESTING STRATEGIES

### 1. Unit Testing

It focuses on smallest unit of software design. In this we test an individual unit or group of inter related units.It is often done by programmer by using sample input and observing its corresponding outputs.
Example:

a) In a program we are checking if loop, method or function is working fine
b) Misunderstood or incorrect, arithmetic precedence.
c) Incorrect initialization

### 2. Integration Testing

The objective is to take unit tested components and build a program structure that has been dictated by design.Integration testing is testing in which a group of components are combined to produce output.
Integration testing is of four types: (i) Top down (ii) Bottom up (iii) Sandwich (iv) Big-Bang
Example

a) **Black Box testing:-** It is used for validation. In this we ignores internal working mechanism and focuses on what is the output?.
b) **White Box testing:-** It is used for verification. In this we focus on internal mechanism i.e.how the output is achieved?

### 3. Regression Testing

Every time new module is added leads to changes in program. This type of testing make sure that whole component works properly even after adding components to the complete program.
Example In school record suppose we have module staff, students and finance combining these modules and checking if on integration these module works fine is regression testing

### 4. Smoke Testing

This test is done to make sure that software under testing is ready or stable for further testing It is called smoke test as testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on.
Example:
If project has 2 modules so before going to module make sure that module 1 works properly

### 5. Alpha Testing

This is a type of validation testing.It is a type of acceptance testing which is done before the product is released to customers. It is typically done by QA people.
Example:
When software testing is performed internally within the organization

### 6. Beta Testing

The beta test is conducted at one or more customer sites by the end-user of the software. This version is released for the limited number of users for testing in real time environment
Example: When software testing is performed for the limited number of people

**7. System Testing**
In this software is tested such that it works fine for different operating system.It is covered under the black box testing technique. In this we just focus on required input and output without focusing on internal working.
In this we have security testing, recovery testing , stress testing and performance testing
Example:
This include functional as well as non functional testing

**8. Stress Testing**
In this we gives unfavorable conditions to the system and check how they perform in those condition.

Example:
(a) Test cases that require maximum memory or other resources are executed
(b) Test cases that may cause thrashing in a virtual operating system
(c) Test cases that may cause excessive disk requirement

**9. Performance Testing**
It is designed to test the run-time performance of software within the context of an integrated system.It is used to test speed and effectiveness of program.
Example:
Checking number of processor cycles.

| **Type of Testing** | **Focus Area** | **Objective** | **Performed By** | **Example** |
|---|---|---|---|---|
| Unit Testing | Individual components or functions | Verify correctness of smallest code units | Developer | Checking function logic or variable initialization |
| Integration Testing | Interaction between modules | Verify module interfaces and data flow | Developer / Tester | Black-box and white-box integration tests |
| Regression Testing | Entire system after changes | Ensure modifications don’t break existing features | QA Team | Retesting after adding a new module |
| Smoke Testing | Initial software build | Verify build stability for further testing | QA / Tester | Basic module verification before deeper testing |
| Alpha Testing | Pre-release internal testing | Validate system before customer release | QA Team | Internal organizational testing |
| Beta Testing | External, real-world testing | Gather user feedback before full release | End-users | Beta version testing at customer sites |
| System Testing | Whole integrated software | Verify end-to-end system compliance | QA Team | Functional and non-functional testing |
| Stress Testing | Extreme or adverse conditions | Check system behavior under stress | QA / Performance Engineer | Memory or resource exhaustion tests |
| Performance Testing | Runtime behavior | Measure speed, efficiency, scalability | QA / Developer | Evaluating response time and throughput |

Software testing is not just about finding bugs — it ensures that software meets quality, reliability, and performance standards. Each testing type plays a unique role in ensuring that the final product is robust, user-friendly, and ready for deployment.

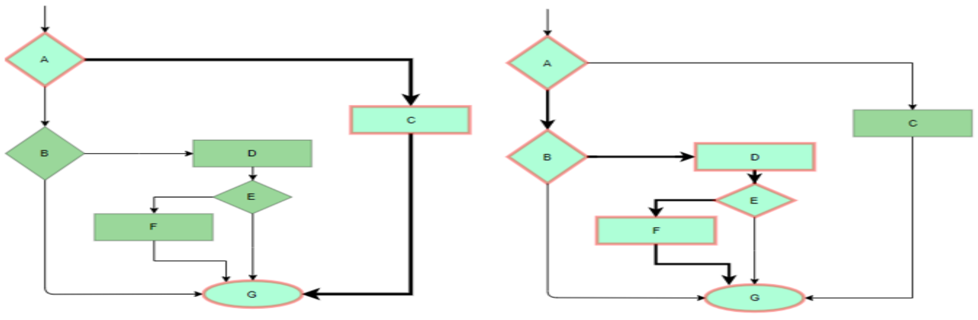
## 9.3 WHITE BOX TESTING

White box testing techniques analyze the internal structures the used data structures, internal design, code structure and the working of the software rather than just the functionality as in black box testing. It is also called glass box testing or clear box testing or structural testing.
Working process of white box testing:

- Input: Requirements, Functional specifications, design documents, source code.
- Processing: Performing risk analysis for guiding through the entire process.
- Proper test planning: Designing test cases so as to cover entire code. Execute rinse-repeat until error-free software is reached. Also, the results are communicated.
- Output: Preparing final report of the entire testing process.

**Testing techniques:**

- Statement coverage: In this technique, the aim is to traverse all statement at least once. Hence, each line of code is tested. In case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, helps in pointing out faulty code.



**Statement Coverage Example**

•**Branch Converge:** In this technique, test cases are designed so that each branch from all decision points are traversed at least once. In a flowchart, all edges must be traversed at least once.

**Purpose**

The main purpose of branch coverage is to:

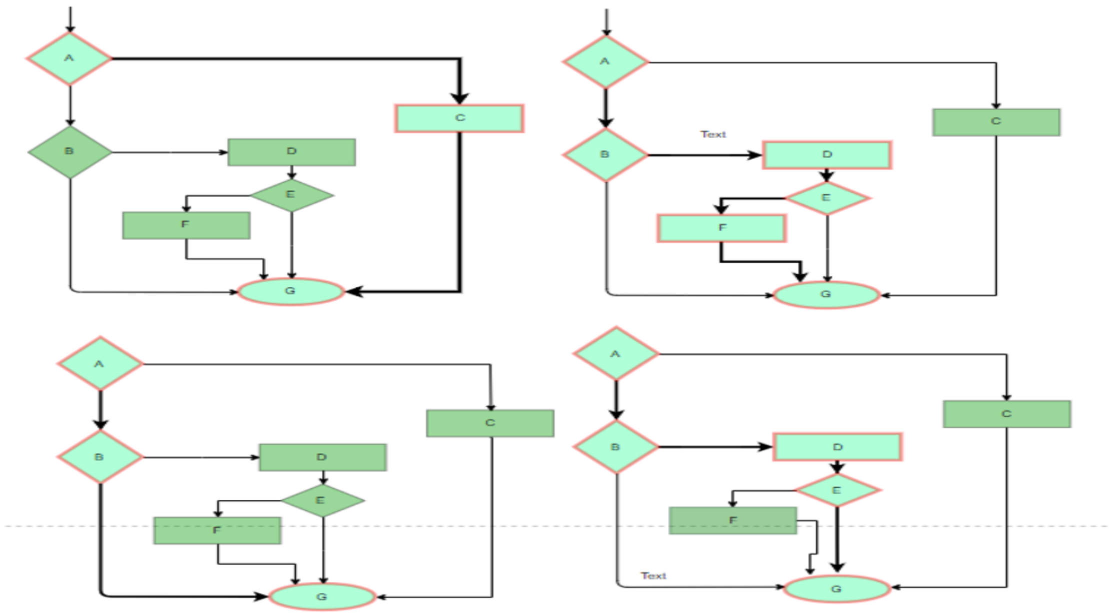
- Ensure that all decisions in the code are tested.
- Detect errors caused by missing or incorrect decision logic.
- Increase the likelihood of uncovering logic-related bugs that may not be revealed by simple statement coverage.

**Advantages of Branch Coverage**

- Ensures that all logical decisions in the program have been tested.
- Helps detect unreachable code and logical errors in control structures.
- Provides a stronger measure of test completeness than statement coverage.
- Useful in critical software systems where logic validation is essential (e.g., banking, medical, or safety systems).

**Disadvantages**

- Does not necessarily ensure that all possible conditions within complex expressions are tested (that is done through Condition Coverage or MC/DC Coverage).
- May require a large number of test cases for programs with many decision points.
- Time-consuming for large or deeply nested code structures.



4 test cases required such that all branches of all decisions are covered, i.e, all edges of flowchart are covered

**Condition Coverage**: In this technique, all individual conditions must be covered as shown in the following example:

1. READ X, Y
2. IF(X == 0 || Y == 0)
3. PRINT ‘0’

In this example, there are 2 conditions: X == 0 and Y == 0. Now, test these conditions get TRUE and FALSE as their values. One possible example would be:

- #TC1 – X = 0, Y = 55
- #TC2 – X = 5, Y = 0

**Multiple Condition Coverage:** In this technique, all the possible combinations of the possible outcomes of conditions are tested at least once. Let’s consider the following example:

0. READ X, Y
1. IF(X == 0 || Y == 0)
2. PRINT ‘0’

- #TC1: X = 0, Y = 0
- #TC2: X = 0, Y = 5
- #TC3: X = 55, Y = 0
- #TC4: X = 55, Y = 5

Hence, four test cases required for two individual conditions.
Similarly, if there are n conditions then 2n test cases would be required.

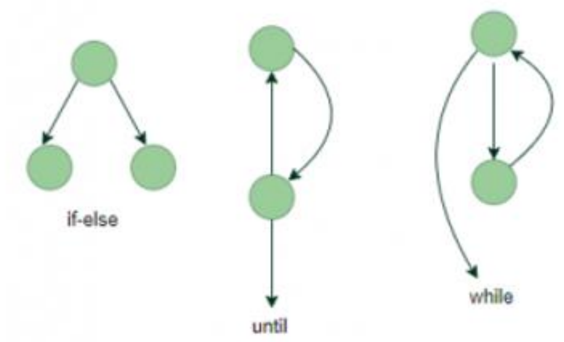
## 9.4 BASIS PATH TESTING

In this technique, control flow graphs are made from code or flowchart and then Cyclomatic complexity is calculated which defines the number of independent paths so that the minimal number of test cases can be designed for each independent path.

Steps:

0. Make the corresponding control flow graph
1. Calculate the cyclomatic complexity
2. Find the independent paths
3. Design test cases corresponding to each independent path

**Flow graph notation:** It is a directed graph consisting of nodes and edges. Each node represents a sequence of statements, or a decision point. A predicate node is the one that represents a decision point that contains a condition after which the graph splits. Regions are bounded by nodes and edges.



**Cyclomatic Complexity:** It is a measure of the logical complexity of the software and is used to define the number of independent paths. For a graph G, V(G) is its cyclomatic complexity.

**Purpose**

- To measure the structural complexity of the program.
- To estimate the effort required for testing and maintenance.
- To identify risky, error-prone, or hard-to-maintain modules.
- To determine the minimum number of test cases needed for full path coverage.

**Key Concept**

Cyclomatic Complexity is based on the Control Flow Graph (CFG) of the program:

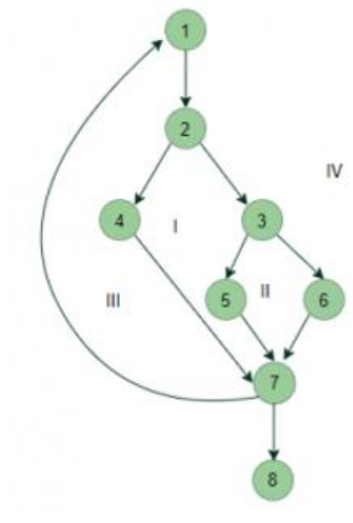
- Nodes represent program statements or decision points.
- Edges represent control flow between those statements.
- Regions represent bounded areas (including the outer region)

**Cyclomatic Complexity (V(G))** is a key software metric that helps developers and testers understand the logical structure of a program. By quantifying the number of independent paths, it guides both testing efforts and maintenance planning. Keeping cyclomatic complexity low leads to simpler, more reliable, and easier-to-maintain software.

Calculating V(G):

1. V(G) = P + 1, where P is the number of predicate nodes in the flow graph
2. V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes
3. V(G) = Number of non-overlapping regions in the graph

Example:



V(G) = 4 (Using any of the above formulae)
No of independent paths = 4

- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

**Loop Testing:** Loops are widely used and these are fundamental to many algorithms hence, their testing is very important. Errors often occur at the beginnings and ends of loops.

- Simple loops: For simple loops of size n, test cases are designed that:
- Skip the loop entirely
- Only one pass through the loop
- 2 passes
- m passes, where m < n
- n-1 ans n+1 passes

**Nested loops:** For nested loops, all the loops are set to their minimum count and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.
**Concatenated loops:** Independent loops, one after another. Simple loop tests are applied for each.
If they're not independent, treat them like nesting.

**Advantages:**

1. White box testing is very thorough as the entire code and structures are tested.
2. It results in the optimization of code removing error and helps in removing extra lines of code.
3. It can start at an earlier stage as it doesn't require any interface as in case of black box testing.
4. Easy to automate.

**Disadvantages:**

1. Main disadvantage is that it is very expensive.
2. Redesign of code and rewriting code needs test cases to be written again.

3. Testers are required to have in-depth knowledge of the code and programming language as opposed to black box testing.
4. Missing functionalities cannot be detected as the code that exists is tested.
5. Very complex and at times not realistic.

## 9.5 BLACK BOX TESTING

Black box testing is a type of software testing in which the functionality of the software is not known. The testing is done without the internal knowledge of the products.
Black box testing can be done in following ways:

1. **Syntax Driven Testing** – This type of testing is applied to systems that can be syntactically represented by some language. For example- compilers,language that can be represented by context free grammar. In this, the test cases are generated so that each grammar rule is used at least once.
2. **Equivalence partitioning** – It is often seen that many type of inputs work similarly so instead of giving all of them separately we can group them together and test only one input of each group. The idea is to partition the input domain of the system into a number of equivalence classes such that each member of class works in a similar way, i.e., if a test case in one class results in some error, other members of class would also result into same error.

The technique involves two steps:

1. **Identification of equivalence class** – Partition any input domain into minimum two sets: valid values and invalid values. For example, if the valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.

2. **Generating test cases** –

(i) To each valid and invalid class of input assign unique identification number.
(ii) Write test case covering all valid and invalid test case considering that no two invalid inputs mask each other.

To calculate the square root of a number, the equivalence classes will be:

(a) Valid inputs:
- Whole number which is a perfect square- output will be an integer.
- Whole number which is not a perfect square- output will be decimal number.
- Positive decimals

(b) Invalid inputs:
- Negative numbers(integer or decimal).
- Characters other that numbers like “a”,”!”,”;”,etc.

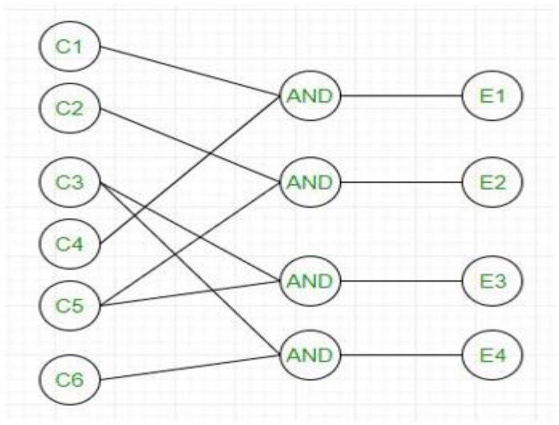
3. **Boundary value analysis** – Boundaries are very good places for errors to occur. Hence if test cases are designed for boundary values of input domain then the efficiency of testing improves and probability of finding errors also increase. For example – If valid range is 10 to 100 then test for 10,100 also apart from valid and invalid inputs.

4. **Cause effect Graphing** – This technique establishes relationship between logical input called causes with corresponding actions called effect. The causes and effects are represented using Boolean graphs.

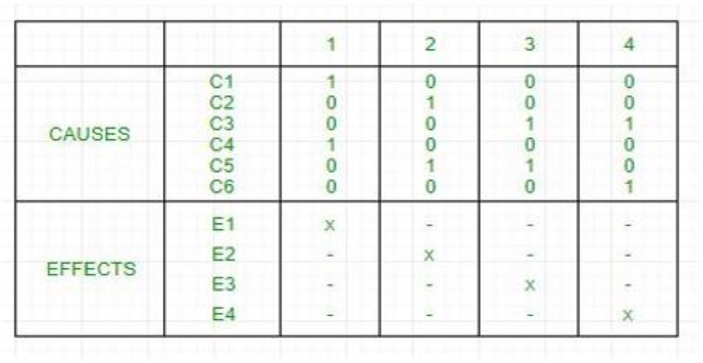
The following steps are followed:

1. Identify inputs (causes) and outputs (effect).
2. Develop cause effect graph.
3. Transform the graph into decision table.
4. Convert decision table rules to test cases.

For example, in the following cause effect graph:



It can be converted into decision table like:



| | | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| CAUSES | C1 | 1 | 0 | 0 | 0 |
| | C2 | 0 | 1 | 0 | 0 |
| | C3 | 0 | 0 | 1 | 1 |
| | C4 | 1 | 0 | 0 | 0 |
| | C5 | 0 | 1 | 1 | 0 |
| | C6 | 0 | 0 | 0 | 1 |
| EFFECTS | E1 | x | - | - | - |
| | E2 | - | x | - | - |
| | E3 | - | - | x | - |
| | E4 | - | - | - | x |

Each column corresponds to a rule which will become a test case for testing. So there will be test cases.

5. Requirement based testing – It includes validating the requirements given in SRS of software system.
6. Compatibility testing – The test case result not only depend on product but also infrastructure for delivering functionality. When the infrastructure parameters are changed it is still expected to work properly. Some parameters that generally affect compatibility of software are:

1. Processor (Pentium 3,Pentium 4) and number of processors.
2. Architecture and characteristic of machine (32 bit or 64 bit).
3. Back-end components such as database servers.
4. Operating System (Windows, Linux, etc)

## 9.6 OBJECT ORIENTED TESTING

As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message passing, polymorphism, concurrency, etc.

Testing classes is a fundamentally different issue than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these approaches.

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variable and its types.

But in Object-Oriented systems, there are the following additional dependencies:

- Class-to-class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

1. **Class-to-Class Dependencies**
   - Definition:
     A *class-to-class dependency* occurs when one class relies on another for its functionality, typically through inheritance, composition, aggregation, or association.
   - Example:
     - A Car class depends on an Engine class for its operations.
     - A Student class depends on a Course class to represent enrollment.
   - Impact:
     - High class-to-class dependency increases coupling and makes systems harder to maintain.
     - Should be managed using interfaces, abstraction, and dependency injection.

2. **Class-to-Method Dependencies**
   - Definition:
     A *class-to-method dependency* exists when a class depends on specific methods of another class to perform its tasks.
   - Example:
     A PaymentProcessor class depends on the validateCard() method of a CreditCard class.
   - Impact:
     - Changes to the called method (e.g., parameters or behavior) may affect all dependent classes.
     - Encourages careful API design and use of interfaces.

3. **Class-to-Message Dependencies**
- Definition:
  This dependency occurs when a class sends a message (method invocation) to another class to request an action or service.
- Example:
  A Customer class sends a message like order.placeOrder() to an Order class.
- Impact:
  - Represents runtime communication between objects.
  - Excessive message dependencies can indicate tight coupling and poor encapsulation.

4. **Class-to-Variable Dependencies**
- Definition:
  A *class-to-variable dependency* exists when one class directly accesses or modifies the member variables (attributes) of another class.
- Example:
  A ReportGenerator class accessing employee.salary from an Employee class.
- Impact:
  - Violates encapsulation if variables are not properly protected.
  - Should be replaced by getter and setter methods to maintain data hiding.

5. **Method-to-Variable Dependencies**
- Definition:
  This dependency arises when a method depends on variables (attributes) — either within the same class or another class — for performing its logic.
- Example:
  The calculateInterest() method depends on balance and rate variables.
- Impact:
  - Encourages encapsulation; internal variable dependencies are expected, but cross-class variable access should be minimized.

6. **Method-to-Message Dependencies**
- Definition:
  A *method-to-message dependency* occurs when a method sends a message (invokes another method) within the same class or across classes.
- Example:
  The checkout() method of a ShoppingCart class calling payment.processPayment().
- Impact:
  - Indicates control flow and collaboration between methods.
  - Should be designed to minimize tight coupling between methods.

**7. Method-to-Method Dependencies**
- Definition:
  A *method-to-method dependency* exists when one method directly invokes another method, either within the same class (internal dependency) or across classes (external dependency).
- Example:
  The calculateTotal() method calling applyDiscount() within the same class.
- Impact:
  - Common and natural in modular design.

- Excessive inter-method dependencies can reduce modularity and increase testing complexity.

**Summary Table: Dependencies in Object-Oriented Systems**

| **Type of Dependency** | **Description** | **Example** | **Impact / Concern** |
|---|---|---|---|
| Class-to-Class | One class depends on another for behavior or structure. | Car depends on Engine. | May increase coupling. |
| Class-to-Method | A class depends on a method in another class. | PaymentProcessor → validateCard() | Sensitive to method changes. |
| Class-to-Message | A class sends a message (method call) to another class. | Customer → order.placeOrder() | Represents runtime interaction. |
| Class-to-Variable | A class accesses variables of another class. | ReportGenerator → employee.salary | Can violate encapsulation. |
| Method-to-Variable | A method uses class variables for operations. | calculateInterest() uses balance. | Encourages encapsulation if internal. |
| Method-to-Message | A method invokes another method via message passing. | checkout() → processPayment() | Defines control flow. |
| Method-to-Method | A method calls another method directly. | calculateTotal() → applyDiscount() | Common, but may increase complexity if excessive. |

Understanding and managing dependencies is crucial in object-oriented software design.Well-structured dependencies promote low coupling, high cohesion, and better maintainability. By carefully analyzing relationships between classes and methods, developers can design systems that are modular, extensible, and easier to test and evolve.

## 9.7 SUMMARY

In conclusion, effective testing strategies are essential for delivering high-quality and dependable software systems. A well-structured and strategic approach to software testing facilitates systematic planning, prioritization, and execution, addressing critical aspects such as test scope, resource allocation, and risk management. Testing strategies for conventional software and object-oriented software differ in their focus — the former emphasizes functional decomposition and control flow, while the latter targets object interactions, inheritance, and encapsulation. Both require tailored methodologies to ensure comprehensive test coverage. Validation testing plays a vital role in confirming that the developed software meets all user needs and business requirements, whereas system testing verifies that the integrated system complies with its overall specifications and environmental constraints.
Equally important is the art of debugging, which involves systematic identification, analysis, and correction of defects discovered during testing. Effective debugging enhances software reliability and stability, reducing future maintenance costs. Together, these testing strategies, validation practices, and debugging techniques form a comprehensive framework that ensures software robustness, functionality, performance, and user satisfaction—the ultimate goals of any successful software engineering effort.

## 9.8 TECHNICAL TERMS

Testing, Strategies, Black box, object, white box, Boundary, Planning, Excution, Validation Testing, Software testing.

## 9.9 SELF ASSESSMENT QUESTIONS

**Essay questions:**

1. What is the primary goal of a strategic approach to software testing?
2. Identify and explain three strategic issues in software testing.
3. Describe the main differences between test strategies for conventional software and object-oriented software.
4. What is the purpose of validation testing in the software development lifecycle?

**Short questions:**

1. 6How does system testing ensure that a software system meets its specified requirements?
2. What are the key activities involved in the art of debugging?
3. Why is it important to plan and execute testing activities systematically?
4. Explain the role of unit testing and integration testing in conventional software testing strategies.
5. What specific challenges are addressed by testing strategies for object-oriented software?
6. How do validation testing and system testing complement each other in ensuring software quality?

## 9.10 SUGGESTED READINGS

1. "Software Engineering: A Practitioner's Approach" by Roger S. Pressman
2. "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
3. "Refactoring: Improving the Design of Existing Code" by Martin Fowler
4. "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (often referred to as the Gang of Four, or GoF)

Dr. Vasantha Rudrarnalla

LESSON- 08

# DESIGN ENGINEERING

## AIMS AND OBJECTIVES

After completing this lesson, the learner will be able to:

- Explain the role of **design** in the software engineering process.
- Identify and describe **key elements of the design model**.
- Understand the **design process** and **factors influencing design quality**.
- Apply **design concepts** such as abstraction, modularity, and information hiding.
- Describe **architectural, data, interface, component, and deployment design elements**.
- Understand the importance of **patterns, refactoring, and frameworks** in modern design.
- Recognize how **pattern-based software design** enhances reusability and scalability.

## STRUCTURE

## 8.1 DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design is the technical kernel of the engineering process. It begins once the requirements model has been validated and acts as the foundation for all later phases — coding, testing, and maintenance.
Pressman emphasizes that design is not coding, but it provides the structural framework that makes coding systematic and efficient.
Design transforms:

- Information (from analysis) into data structures.
- Functional requirements into software components.
- Behavioral models into architectural and procedural representations.

A good design must satisfy both functional requirements (what the system should do) and non-functional requirements (performance, reliability, usability, and maintainability).

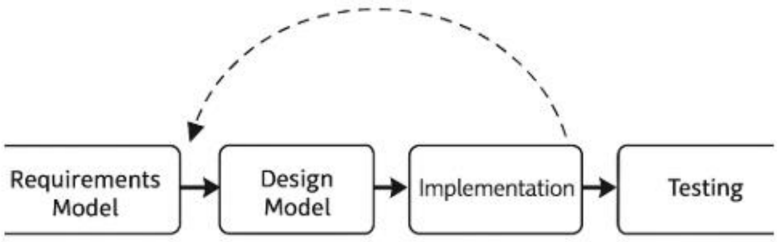


**Figure 8.1 – The Design Process Context**
***(Depicts flow: Requirements Model → Design Model → Implementation → Testing)***

## 8.2 DESIGN PROCESS AND DESIGN QUALITY

**The Design Process**
The design process follows a logical progression:

1. Understand the problem domain.
2. Identify design objectives and constraints.
3. Develop representations — data, architecture, interface, and components.
4. Refine these representations into implementable structures.

Each design phase feeds into the next, ensuring traceability from requirement to implementation.

**Design Quality Attributes**
A quality design exhibits the following attributes:

- Correctness: The design satisfies all specified requirements.
- Understandability: Easy for developers and maintainers to comprehend.
- Efficiency: Promotes optimal use of resources.
- Maintainability: Supports future enhancements with minimal rework.
- Reusability: Encourages reuse of components or design patterns.

Design quality = structure + clarity + flexibility.

**Quality Guidelines**

- A design should exhibit an architecture which
- has been created using recognizable architectural styles or patterns,

- is composed of components that exhibit is composed composed of components components that exhibit exhibit good design characteristics good design characteristics characteristics
- can be implemented in an evolutionary fashion
- A design should be modular
- A design should contain distinct representations A design should contain contain distinct distinct representations representations of data architecture of data, architecture architecture,
- interfaces, and components.
- A design should lead to data structures that are
- appropriate for the classes to be implemented appropriate appropriate for the classes classes to be implemented implemented
- drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional
- characteristics.
- A design should lead to interfaces that reduce the complexity of connections
- between components and with the external environment.
- A design should be represented effectively for communicating its meaning.

## 8.3 DESIGN CONCEPTS

Pressman identifies key concepts that form the foundation of sound software design.

### 8.3.1 Abstraction

Abstraction allows designers to focus on essential features without being distracted by implementation details.

It provides multiple layers of design — from general system architecture to specific algorithms.

Types of Abstraction:
- Procedural Abstraction: Represents a sequence of instructions.
- Data Abstraction: Encapsulates data with associated operations.
- Control Abstraction: Describes control mechanisms without details.

*Example:*
A "Library Account" object abstracts data (member details) and behavior (borrow, return).

**Design Pattern Template (as per Pressman, 6th Edition)**

A Design Pattern Template provides a standard structure for documenting reusable design solutions. It allows developers to describe a recurring design problem, its context, and its proven solution in a consistent, easy-to-understand format.
Each pattern entry in this template captures what the pattern does, when to use it, and how it works, ensuring that teams can communicate design knowledge effectively.

**Design Pattern Template**

| **Section** | **Description** |
|---|---|
| Pattern Name | Provides a short, descriptive, and expressive name for the pattern. The name should capture the *essence of the pattern* and make it easy to refer to and discuss. *Example:* Singleton, Observer, Factory Method. |
| Intent | Describes the goal or purpose of the pattern — *what it does* and *why it exists*. It summarizes the problem and the pattern's solution in one or two sentences. *Example:* "Ensure that a class has only one instance and provide a global access point to it." |
| Motivation | Provides a real-world example or scenario that illustrates the *problem context* and how the pattern provides a solution. This section helps the reader understand *why* the pattern is useful. *Example:* In a logging system, multiple instances can cause inconsistent log data; the Singleton pattern ensures only one instance handles all logs. |
| Applicability | Specifies situations or contexts where the pattern can be applied. It describes design problems and conditions that indicate when this pattern is suitable. *Example:* Use the Factory Method pattern when a class cannot anticipate the class of objects it must create. |
| Structure | Describes the classes and objects that participate in the pattern, typically illustrated using UML class or interaction diagrams. This section visually shows the static relationships among components. |
| Participants | Lists and describes the responsibilities of each class or object in the pattern. It explains what role each participant plays in realizing the pattern. *Example:* In the Observer pattern, the "Subject" maintains a list of "Observers" and notifies them of changes. |
| Collaborations | Describes how the participants interact to carry out their responsibilities. It explains the flow of control and data between objects in the pattern. *Example:* The Subject calls an update() method on each Observer when its state changes. |
| Consequences (optional) | Explains the results and trade-offs of using the pattern — such as benefits, drawbacks, or system impacts (e.g., performance, flexibility). *Example:* Singleton improves control but reduces testability. |
| Implementation | Provides guidelines or steps for implementing the pattern. May include sample code or pseudocode showing how classes are defined and interact. |
| Known Uses (optional) | Gives real-world examples or systems where the pattern has been applied successfully. *Example:* The MVC architecture uses the Observer pattern to synchronize views with the model. |
| Related Patterns | Cross-references other patterns that are related in intent or structure. Helps identify complementary or alternative solutions. *Example:* The Abstract Factory pattern is often used with the Factory Method pattern. |

**Example (Extract – Singleton Pattern)**

| | |
|---|---|
| Field | Example Entry |
| Pattern Name: | Singleton |

| Field | Example Entry |
|---|---|
| Intent: | Ensure that only one instance of a class exists and provide a global access point to it. |
| Motivation: | Logging, configuration, or driver management requires a single control object. |
| Applicability: | Use when a single instance is needed to coordinate actions across the system. |
| Structure: | One class with a private constructor and a static instance variable. |
| Participants: | Singleton class manages its own instance creation and access. |
| Collaborations: | Clients access the single instance via a static method (e.g., getInstance()). |
| Related Patterns: | Abstract Factory, Builder. |

This Design Pattern Template ensures uniform documentation and understanding of reusable solutions across teams.
By defining sections such as Intent, Motivation, Structure, and Collaborations, Pressman's approach encourages clarity, consistency, and reusability — key aspects of high-quality software design.

### 8.3.2 Modularity

Modularity divides a system into manageable, logically distinct units called modules. Each module performs a specific function and communicates with others through defined interfaces.
Advantages:

- Simplifies development and debugging.
- Supports parallel work among teams.
- Improves reusability and testing.
- Facilitates maintenance.

### 8.3.3 Information Hiding

Proposed by David Parnas, this principle suggests that modules should hide internal details and expose only necessary interfaces.

Changes inside a module should not affect others — leading to low coupling and high cohesion.
Example:
A banking module hides interest calculation logic while exposing only public methods for account operations.

**Why Information Hiding?**

Information hiding is essential because it:

1. **Reduces the likelihood of side effects:**
    - When one module changes, hidden internal details ensure that other modules remain unaffected.

- This prevents unintended consequences — also known as *side effects* — in other parts of the software.

2. **Limits the global impact of local design decisions:**
   - Internal modifications (like changing data representation or algorithm logic) can be done **without impacting** the rest of the system.
   - This supports **maintainability** and **evolution** of software.
3. **Emphasizes communication through controlled interfaces:**
   - Modules interact via **well-defined interfaces** (public methods, APIs) rather than through direct access to internal data.
   - This enforces **discipline** in communication between components.
4. **Discourages the use of global data:**
   - Global variables make programs difficult to debug and maintain because changes can have widespread effects.
   - By hiding data within modules, each component becomes **self-contained** and **robust**.
5. **Leads to Encapsulation:**
   - **Encapsulation** is a design attribute where **data and the operations that manipulate it** are bundled together.
   - It is a direct result of information hiding and is the **core of object-oriented design**.
   - Example: A BankAccount class hides its balance attribute and provides controlled access via methods like deposit() and withdraw().
6. **Results in Higher Quality Software:**
   - Designs based on information hiding are **easier to understand, modify, test, and reuse**.
   - They improve **modularity**, **flexibility**, and **reliability**, leading to high-quality software products.

### 8.3.4 Functional Independence

A design achieves functional independence when modules:

- Perform unique, cohesive tasks, and
- Minimize dependencies (coupling) with other modules.

Cohesion: Measures how closely related the functions within a module are.
Coupling: Measures how much one module depends on others.
The goal is high cohesion, low coupling — a hallmark of good design.

### 8.3.5 Refinement

Refinement is a top-down process of elaborating design details.
High-level functions are decomposed into more specific operations until each is detailed enough for coding.

Example:
"Manage Library Accounts" → "Add Member," "Update Record," "Deactivate Account."

## 8.4 THE DESIGN MODEL

The Design Model represents the structure of the system, its components, interfaces, and deployment configuration. Pressman identifies five key design elements, each describing a different aspect of the system.

### 8.4.1 Data Design Elements

Define how data structures are organized, stored, and accessed. Data design ensures consistency with the data model created during analysis.
Includes:

- Data objects, attributes, and relationships.
- Database schema design and normalization.
- Data structures used by algorithms.

### 8.4.2 Architectural Design Elements

The software architecture defines the overall structure — how major components interact. It provides a blueprint for system organization, capturing both static structure and dynamic behavior.
Common Architectural Styles:

- Layered Architecture (e.g., Presentation–Business–Data)
- Client–Server Architecture
- Object-Oriented Architecture
- Component-Based and Service-Oriented Architectures (SOA)

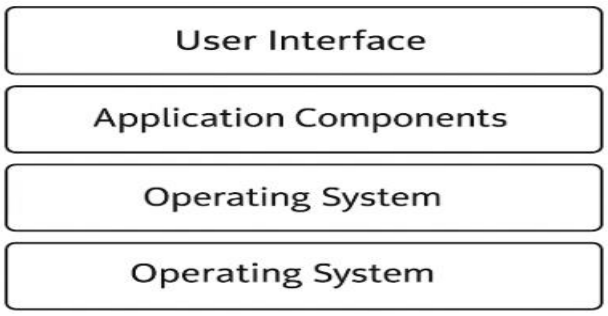


Figure 8.2 – Example of a Layered Software Architecture

Figure 8.2 illustrates the Layered Software Architecture, one of the most widely used and fundamental architectural styles in software engineering. A layered architecture organizes software into hierarchical layers, where each layer provides services to the layer above it and depends on the layer below it.
This design promotes separation of concerns, modularity, and scalability, making it easier to maintain and evolve large systems.

**Advantages of Layered Architecture**

| **Feature** | **Benefit** |
|---|---|
| Separation of Concerns | Simplifies understanding and maintenance by isolating functionalities. |
| Ease of Maintenance | Changes in one layer (e.g., UI redesign) do not affect others. |
| Reusability | Each layer's services can be reused by other applications. |
| Scalability | Additional features (e.g., new services, APIs) can be added with minimal changes. |
| Testability | Each layer can be unit tested independently. |

**Example – Library Management System (Layered View)**

| **Layer** | Example Components |
|---|---|
| **Presentation Layer** | User interface for librarians and members, web portal |
| **Business Logic Layer** | Book issue/return management, fine calculation, membership validation |
| **Data Layer** | Database storing book records, member information, transaction history |

The Layered Software Architecture embodies Pressman's principle of structured design — ensuring that systems are built in well-organized, stable, and evolvable layers, leading to high-quality software.

### 8.4.3 Interface Design Elements

Defines how modules and users interact with the system.
It includes both user interfaces (UI) and inter-module interfaces (APIs).
Principles:

- Consistency in layout and behavior.
- Minimal user effort.
- Clear navigation and feedback.
- Standardization of data exchange protocols.

#### 1. User Interface (UI) Design

The **User Interface (UI)** represents the **visible and interactive part** of a software system — what users see and how they perform tasks.
It involves not only visual appearance but also **interaction design**, **workflow**, and **user experience (UX)** principles.
**Key Goals of UI Design:**

- To make the system **easy to learn**, **efficient to use**, and **pleasant to interact with**.
- To translate system functionality into **clear visual metaphors** (buttons, menus, icons, forms).
- To ensure that the system supports **user goals** and reduces cognitive load.

**UI Design Principles (as per Pressman)**

| **Principle** | **Description** |
|---|---|
| **Consistency** | Maintain uniformity in layout, color schemes, controls, and terminology throughout the application. Consistency reduces user confusion and enhances predictability. |
| **Minimal User Effort** | The interface should minimize the number of steps required to perform a task. Avoid redundant confirmations and simplify workflows. |
| **Clarity and Feedback** | Every user action should trigger immediate feedback — visual (loading indicators), auditory, or textual (error/success messages). |
| **Visibility of System Status** | Users should always know what the system is doing — through progress indicators, status bars, or confirmation messages. |
| **Error Prevention and Recovery** | Design should minimize user errors and provide clear guidance for correction. Example: Confirm before deleting data. |
| **Flexibility and Efficiency** | Support novice users through guidance while allowing experts to use shortcuts or advanced commands. |
| **Aesthetic and Minimal Design** | Avoid unnecessary visual clutter. Present information in a clean, organized layout aligned with usability goals. |

**Example – Library Management System (UI Interaction)**

- The user logs in via a **login form** (input validation ensures correct credentials).
- After successful login, the **dashboard interface** displays menus for book search, issue, and return operations.
- The system provides **real-time feedback** — e.g., “Book successfully issued” or “Book not available.”
- The interface maintains a consistent look (same colors, typography, and navigation structure).

**2. Inter-Module Interface Design (API Design)**

Beyond the visual interface, software systems also rely on **inter-module communication** — internal and external connections that allow components to exchange data or trigger operations.

An **Application Programming Interface (API)** defines *how modules communicate* through **function calls**, **messages**, or **data streams**.

**Objectives of API Design:**

- Enable modules to interact **independently** without exposing internal details.
- Ensure **standardized data exchange protocols** (e.g., JSON, XML, REST).
- Support **extensibility** — new modules can be added without modifying existing ones.
- Promote **reuse** across applications or systems.

**API Design Principles**

| **Principle** | **Description** |
|---|---|
| **Encapsulation** | APIs expose only what is necessary, hiding implementation details. |
| **Simplicity** | Keep method signatures and data formats simple and intuitive. |
| **Consistency** | Follow consistent naming conventions, parameter orders, and error-handling mechanisms. |
| **Statelessness (for Web APIs)** | Each request should be independent to ensure scalability and simplicity (as in REST APIs). |
| **Version Control** | Maintain backward compatibility and version identifiers (v1, v2) for evolving systems. |
| **Security** | Protect data exchange through authentication, authorization, and encryption mechanisms. |

**Example – Library Management System (API Interface)**

| **API Endpoint** | **Purpose** | **Request/Response Format** |
|---|---|---|
| /api/books/search | Retrieve books by title or author | Request: GET /api/books/search?title=AI → Response: { "bookID": 501, "title": "AI Concepts", "status": "Available" } |
| /api/books/issue | Issue a book to a member | Request: POST /api/books/issue → Response: { "status": "Issued", "dueDate": "2025-11-15" } |
| /api/members/register | Add new library members | Request: POST /api/members/register → Response: { "memberID": 1023, "status": "Active" } |

These APIs ensure smooth interaction between the front-end (Presentation Layer) and the back-end (Database Layer).

### 8.4.4 Component-Level Design Elements

Each software component is designed in detail — defining its classes, methods, attributes, and relationships.

This is where object-oriented design principles like encapsulation, inheritance, and polymorphism are applied.
Example:
Class Book with attributes (*title, author, ISBN*) and methods (*issue(), return()*).

**Objectives of Component-Level Design**
The key objectives are to:

1. Define the structure and behavior of each software component.
2. Specify the internal logic and interactions of classes and functions.
3. Apply object-oriented principles such as encapsulation, inheritance, and polymorphism.
4. Ensure that each component supports reusability, testability, and maintainability.
5. Align detailed component design with the architectural framework.

**Key Elements of Component-Level Design**

| **Element** | **Description** |
|---|---|
| Classes | Represent real-world entities or logical abstractions. Each class defines data (attributes) and behavior (methods). |
| Attributes | Define the data held by a class or component. Attributes represent the *state* of an object. |
| Methods / Operations | Define the *behavior* of the class — how it manipulates its data and interacts with other classes. |
| Interfaces | Define how other classes or components can access the functionality of the component. |
| Relationships | Describe how classes are connected — through association, aggregation, composition, or inheritance. |
| Packages / Modules | Logical groupings of related classes that form larger components or subsystems. |

**Component-Level Design Workflow**

1. Identify Components: Based on analysis and architecture models, determine logical building blocks (e.g., Book, Member, Transaction).
2. Define Class Hierarchies: Establish inheritance and composition relationships.
3. Specify Interfaces: Clearly define the entry points and methods available to other components.
4. Describe Internal Logic: Use pseudocode, flowcharts, or structured English to describe behavior.
5. Verify and Refine: Ensure that component logic satisfies design constraints and aligns with architectural decisions.

**Advantages of Component-Level Design**

| **Aspect** | **Benefit** |
|---|---|
| Reusability | Components can be reused across projects with minimal modification. |
| Maintainability | Encapsulated components can be updated independently. |
| Scalability | New components can be integrated easily without affecting others. |
| Testability | Each component can be tested in isolation. |
| Reliability | Independent design reduces the risk of failure propagation. |

**Component-Level Design and UML**

Component-level design is often represented using UML diagrams:

- Class Diagrams: Show structure, attributes, methods, and relationships.
- Sequence Diagrams: Illustrate dynamic interactions among components.
- Component Diagrams: Depict system-level organization and dependencies.

## 8.4.5 Deployment-Level Design Elements

Defines how software components are physically distributed across hardware and networks. It ensures scalability, reliability, and performance.
Deployment design includes:

- Mapping software elements to physical devices.
- Network topology.
- Load balancing and redundancy.

**Example – Library Management System (Deployment Design)**
**Scenario: The university library system is web-based and must serve both local and remote users.**

| Deployment Element | Physical Configuration | Purpose |
|---|---|---|
| Web Interface | Hosted on Web Server | Handles user requests and displays results. |
| Business Logic (Application Layer) | Deployed on Application Server | Processes user inputs and executes core functions. |
| Database | Centralized on Database Server | Stores book records, user accounts, and transactions. |
| Backup Server | Remote Site / Cloud | Maintains backup copies for disaster recovery. |
| Network | Secure LAN + Internet Gateway | Provides connectivity between users and servers. |

**Key Characteristics of Deployment Design**

| Aspect | Description |
|---|---|
| Distribution | Defines how components are spread across different physical or virtual machines. |
| Concurrency | Handles multiple requests simultaneously without performance degradation. |
| Fault Tolerance | Ensures continuity during hardware, software, or network failures. |
| Security | Implements encryption, authentication, and firewalls to protect data and services. |
| Scalability | Enables the addition of new servers or services as demand grows. |
| Performance | Optimizes response time through caching, compression, and efficient routing. |

**Advantages of Deployment-Level Design**

| Benefit | Explanation |
|---|---|
| Improved Reliability | Redundancy and fault tolerance minimize downtime. |
| Enhanced Performance | Proper load distribution improves system responsiveness. |

| | |
|---|---|
| Ease of Maintenance | Modular deployment allows isolated upgrades. |
| Scalability | Supports growth without redesigning the entire system. |
| Security and Control | Enables secure communication and controlled access. |

**Best Practices for Deployment Design**

1. Use Layered Deployment: Align with layered architecture (UI → Business → Data).
2. Automate Deployment: Use tools like Docker, Kubernetes, Jenkins, or Ansible.
3. Monitor System Health: Implement real-time monitoring for resource utilization and service uptime.
4. Plan for Disaster Recovery: Maintain offsite backups and failover mechanisms.
5. Document the Configuration: Include diagrams and mapping tables for clarity and reproducibility.

The Deployment-Level Design is where the logical software model becomes a physical reality.
It defines how and where each software component will operate, ensuring that the system delivers scalability, reliability, and performance under real-world conditions.

**Example: Object-Oriented Design for Library Management System**
**Classes:**

- Book – Represents each library book.
- Member – Represents library members.
- Librarian – Manages book records and membership.
- Transaction – Handles book issue and return operations.

**Relationships:**

- Member *borrows* Book.
- Librarian *manages* Book and Member.
- Transaction *uses* both Member and Book.

Object-Oriented Design (OOD) transforms a problem into a network of collaborating objects, promoting modularity, reusability, and clarity.
It is grounded on the four foundational principles — Encapsulation, Inheritance, Polymorphism, and Abstraction — and supported by additional concepts like composition, association, and low coupling with high cohesion.

## 8.5 PATTERN-BASED SOFTWARE DESIGN

Modern design uses patterns to capture proven solutions to recurring problems.

### 8.5.1 Describing a Design Pattern

A design pattern is a reusable template describing how to solve a class of problems in a particular context.
Each pattern typically includes:

- Name – A meaningful label (e.g., Singleton, Observer, Factory).
- Intent – What the pattern accomplishes.
- Motivation – The context or situation where it applies.
- Structure – Class and interaction diagrams.
- Consequences – Trade-offs and results of applying the pattern.

### 8.5.2 Using Patterns in Design

Patterns improve consistency and reuse. They help teams:

- Apply standard solutions for known challenges.
- Communicate effectively using a common design vocabulary.
- Reduce design time through reuse of existing approaches.

Categories of Patterns:

- Creational: Deal with object creation (e.g., Singleton, Factory Method).
- Structural: Define composition of classes (e.g., Adapter, Decorator).
- Behavioral: Manage communication (e.g., Observer, Strategy).

### 8.5.3 Refactoring and Frameworks

- Refactoring is the process of improving internal code structure without changing its external behavior.
- It enhances readability, performance, and maintainability.
- *Example:* Reorganizing methods in a class to reduce redundancy.
- Frameworks are semi-complete software architectures that provide reusable designs for specific domains (e.g., web, GUI, data processing). Frameworks combine design patterns into a cohesive environment for rapid application development.

## 8.6 SUMMARY

- Software design transforms analysis models into an implementation blueprint.
- It involves data, architecture, interface, component, and deployment design.
- Design concepts such as abstraction, modularity, and information hiding ensure quality and maintainability.
- Patterns, frameworks, and refactoring enhance reuse and long-term adaptability.
- A well-structured design leads directly to a robust, reliable, and maintainable software product.

## 8.7 TECHNICAL TERMS

Design Model, Architecture, Abstraction, Modularity, Information Hiding, Functional Independence, Cohesion, Coupling, Design Pattern, Refactoring, Framework, Component, Interface, Deployment Diagram**.**

## 8.8 SELF-ASSESSMENT QUESTIONS

**Essay Questions**

1. Explain the importance of design within the context of software engineering.
2. Describe the key design concepts introduced by Pressman.
3. Discuss the elements of the design model in detail.
4. What are design patterns? Explain their types and advantages.
5. Explain how modularity and information hiding contribute to software quality.

**Short Notes**

1. What is Abstraction
2. Describe brief Layered Architecture
3. How Functional Independence works.
4. Explain Refactoring.
5. Write about Frameworks

## 8.9 SUGGESTED READINGS

1. Roger S. Pressman, *Software Engineering – A Practitioner's Approach*, 6th Edition, TMH International.
2. Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.
3. Ian Sommerville, *Software Engineering*, Pearson Education.
4. Steve McConnell, *Code Complete*, Microsoft Press.
5. Craig Larman, *Applying UML and Patterns*, Prentice Hall

Dr. Kampa Lavanya

LESSON- 10

# TESTING TACTICS

## AIMS AND OBJECTIVES

To understand the principles, strategies, and techniques of software testing, including both conventional and object-oriented methods, and to explore advanced testing strategies for specialized environments and reusable testing patterns that enhance software quality and reliability.

**After completing this lesson, you will be able to:**

- Explain the **fundamentals and objectives** of software testing.
- Distinguish between **white-box** and **black-box** testing techniques.
- Apply structural testing methods such as **basis path**, **data flow**, and **loop testing**.
- Design effective black-box tests using **equivalence partitioning**, **boundary value analysis**, and **orthogonal array testing**.
- Recognize the testing implications of **object-oriented concepts** such as encapsulation, inheritance, and polymorphism.
- Apply testing methods to **specialized environments**, including GUIs, client–server, real-time, and documentation testing.
- Identify and use **software testing patterns** to standardize and reuse proven testing strategies.
- Appreciate the importance of systematic testing throughout the **software development life cycle (SDLC)** to ensure software quality and dependability.

## STRUCTURE

## 10.1 SOFTWARE TESTING FUNDAMENTALS

Software testing is a critical component of software quality assurance (SQA). It provides a **systematic approach to uncovering defects**, ensuring that the software system performs as intended.
The fundamental goal of testing is to **detect errors before the software is delivered** to the end user.
Testing can never prove that a program is completely correct — it can only show that **defects are present** under certain conditions.

As Dijkstra famously stated:
"Testing can show the presence of bugs, but never their absence."

**Objectives of Testing**

1. To **find and correct errors** in the software.
2. To ensure that the **software performs its required functions**.
3. To verify that the **software meets both functional and non-functional requirements**.
1. To build **confidence in the software's reliability** before deployment.
2. To support **maintenance and regression activities** after updates or enhancements.

**Verification and Validation (V&V)**

| Aspect | Verification | Validation |
|---|---|---|
| **Definition** | Ensures the product is built correctly as per design. | Ensures the right product has been built to meet user needs. |
| **Focus** | Process compliance. | Product correctness. |
| **Performed by** | Developers and quality engineers. | End-users or testers. |
| **Example** | Code inspection, design review. | Acceptance testing, usability testing. |

Testing contributes to both verification and validation — it verifies software conformance to specifications and validates the system's functionality from the user's perspective.

**Fundamental Principles of Testing**

1. **Testing shows the presence of defects, not their absence.**
2. **Exhaustive testing is impossible.** Only representative tests can be done.
3. **Early testing saves time and cost.** Start testing activities in the requirements and design phases.
4. **Defects cluster together.** A small number of modules contain the majority of defects (Pareto principle).

5. **Pesticide paradox.** Reusing the same test cases repeatedly finds fewer new defects — test cases must evolve.
6. **Testing is context dependent.** Different systems require different testing approaches.
7. **Absence of errors is a fallacy.** A program that passes tests may still not meet user needs.

**Testing Process**
Software testing follows a defined set of activities:

| **Phase** | **Description** |
|---|---|
| **Test Planning** | Identify scope, objectives, strategy, and resources. |
| **Test Design** | Develop test cases, data, and expected results. |
| **Test Execution** | Run tests, record outcomes, and compare with expectations. |
| **Defect Reporting** | Log detected defects for resolution. |
| **Test Evaluation** | Assess results and determine readiness for release. |

**Levels of Testing**

| **Level** | **Purpose** |
|---|---|
| **Unit Testing** | Verifies individual components or classes. |
| **Integration Testing** | Validates interaction between integrated units. |
| **System Testing** | Tests the complete system for functionality and performance. |
| **Acceptance Testing** | Confirms software’s readiness for delivery to the user. |

**Types of Testing**

| **Category** | **Examples** |
|---|---|
| **Functional Testing** | Unit, integration, system, acceptance. |
| **Non-Functional Testing** | Performance, reliability, security, usability. |
| **Maintenance Testing** | Regression and re-testing after modifications. |

**Testing vs Debugging**
Testing and debugging are closely related but distinct activities:

| **Activity** | **Purpose** | **Performed By** |
|---|---|---|
| **Testing** | Identify the presence of defects. | Testers / QA team. |
| **Debugging** | Locate and correct the defects. | Developers. |

Testing is about **detection**, while debugging is about **correction**.

## 10.2 INTERNAL AND EXTERNAL VIEWS OF TESTING

Software can be tested using two complementary perspectives — **internal (white-box)** and **external (black-box)** testing.

### 10.2.1 Internal View (White-Box Testing)

White-box testing (also called *structural testing*) examines the **internal logic and structure** of the code.

It is based on the knowledge of the program’s control flow, data flow, and algorithms.

**Objective**
To ensure that all **independent paths**, **loops**, and **conditions** are executed at least once.

**Typical Techniques**

- Basis path testing
- Control structure testing
- Loop testing
- Condition and data flow testing

**Advantages**

- Identifies hidden logic errors and boundary issues.
- Ensures high code coverage.
- Facilitates early defect detection at the developer level.

**Disadvantages**

- Requires access to source code.
- Time-consuming for large programs.
- Cannot detect missing functionalities.

### 10.2.2 External View (Black-Box Testing)

Black-box testing (also known as *functional testing*) treats the program as a “black box,” focusing only on **inputs and outputs** without considering internal logic.

**Objective**

To validate that the software’s **observable behavior** matches its specification.

**Typical Techniques**

- Equivalence partitioning
- Boundary value analysis
- Graph-based testing
- Orthogonal array testing

**Advantages**

- Does not require knowledge of code.
- Tests from the user’s perspective.
- Useful for validation and acceptance testing.

**Disadvantages**

- Cannot detect internal logic errors.
- Redundant testing possible for overlapping inputs.
- Coverage depends on test case design quality.

### 10.2.3 Combined Approach

The most effective testing strategy combines both **internal (white-box)** and **external (black-box)** views — a practice often called **gray-box testing**.
This ensures both **structural integrity** and **functional correctness**.

## 10.3 WHITE-BOX TESTING

**Introduction**

White-box testing (also called **structural**, **glass-box**, or **logic-driven testing**) involves examining the **internal workings of a program**.

The tester has full visibility into source code, algorithms, and data structures. This method ensures that **every line of code, decision point, and logical path** is verified for correctness.

**Objectives of White-Box Testing**

1. Ensure that **all independent paths** in a module are executed at least once.
2. Verify **logical conditions** (true/false branches) for accuracy.
3. Check **loops**, **data structures**, and **internal boundaries**.
4. Validate **error-handling and exception mechanisms**.
5. Confirm that **unused or dead code** is identified and removed.

**Steps in White-Box Testing**

| **Step** | **Activity** |
|---|---|
| 1. Code Review | Examine code for logic and style compliance. |
| 2. Flow Graph Creation | Represent control structure visually. |
| 3. Path Identification | List independent control paths. |
| 4. Test Case Design | Create tests to cover all identified paths. |
| 5. Test Execution | Run and observe behavior for expected outcomes. |
| 6. Result Analysis | Verify outputs, coverage, and performance. |

**Advantages**

- Detects logical and computational errors early.
- Provides code coverage measurement.
- Ensures thorough testing of loops and decisions.

**Limitations**

- Requires knowledge of programming logic.
- Ineffective for missing functionalities.
- Can be time-consuming for large systems.

## 10.4 BASIS PATH TESTING

**Definition**

**Basis Path Testing** (developed by *Tom McCabe*) is a **systematic white-box technique** used to derive a **logical complexity measure** of a program and to design a minimal set of test cases that ensure coverage of all independent paths.

**Concept Overview**

Every program can be represented as a **control flow graph (CFG)**, where nodes represent processing statements, and edges represent control flow between statements. By analyzing this graph, testers can identify **independent paths** and derive test cases that guarantee full coverage of decision structures.

**Steps in Basis Path Testing**

1. **Construct a flow graph** of the program.
2. **Calculate cyclomatic complexity (V(G))** to determine the number of independent paths.
3. **Identify independent paths** through the program.
4. **Develop test cases** to execute each path at least once.
5. **Execute and verify** program correctness for each path.

**Advantages of Basis Path Testing**

- Provides a quantitative measure of program complexity.
- Ensures thorough path coverage.
- Helps in identifying unreachable or redundant code.

### 10.4.1 Flow Graph Notation

A **flow graph** (or control flow graph) uses the following basic symbols:

| **Symbol** | **Meaning** |
|---|---|
| **Node** | A sequence of one or more procedural statements. |
| **Edge / Link** | Represents the flow of control between nodes. |
| **Decision Node** | A point where control can branch (e.g., if, while). |
| **Region** | Area bounded by edges and nodes; represents logical partitions. |

**Example**

```
1: Read A, B
2: If A > B then
3: Print "A greater"
4: Else
5: Print "B greater"
6: Endif
7: Stop
```

**Flow Graph Description:**

- **Nodes:** 1 to 7
- **Edges:** Represent transitions between statements
- **Decisions:** Occur at Node 2

### 10.4.2 Independent Program Paths

An **independent path** is any path through the program that introduces at least **one new set of processing statements or decisions** not covered by previously tested paths.

**Cyclomatic Complexity**

Cyclomatic complexity (V(G)) provides a measure of the **logical complexity** of a program and indicates the **minimum number of test cases** required for full path coverage.
It can be computed by:

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

Where:

- **E** = number of edges
- **N** = number of nodes

or equivalently:

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

where **P** is the number of decision nodes.

**Example**

For the earlier flow graph:

| **Nodes (N)** | **Edges (E)** | **Decisions (P)** | **Cyclomatic Complexity (V(G))** |
|---|---|---|---|
| 7 | 8 | 1 | 2 |

**Therefore:**
→ At least **2 independent test paths** are needed to ensure coverage.

### 10.4.3 Deriving Test Cases

After calculating complexity, derive test cases so that each independent path is executed at least once.
**Steps:**
1. List all paths through the program.
2. Identify independent paths.
3. Assign input data to force execution of each path.
4. Document expected outputs.
5. Execute tests and compare results.

**Example**
For the "Compare A and B" example:

| Path | Description | Input Example | Expected Output |
|---|---|---|---|
| P1 | A > B path | A = 8, B = 5 | "A greater" |
| P2 | A ≤ B path | A = 4, B = 7 | "B greater" |

### 10.4.4 Graph Matrices
A **graph matrix** (also called a **connectivity matrix**) is a tabular representation of the flow graph showing which nodes are connected by edges.

| From / To | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **1** | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **4** | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **5** | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

A '1' indicates the presence of a control link between nodes.
**Uses of Graph Matrices**
- Visually identify control flow paths.
- Validate logical completeness and reachability.
- Support automated path analysis tools.

**Benefits of Basis Path Testing**

| Advantage | Explanation |
|---|---|
| Quantitative control over test design | Cyclomatic complexity provides an exact measure. |
| High coverage | Ensures all paths and decisions are tested. |
| Error detection | Uncovers untested branches or unreachable code. |
| Process improvement | Simplifies maintenance and regression test planning. |

**Limitations**
- Not suitable for very large programs with numerous paths.
- Focuses only on control flow; data flow errors may remain.
- Assumes deterministic behavior (no randomness).

## 10.5 CONTROL STRUCTURE TESTING

**Introduction**
Control structure testing focuses on the **logical control structures** of the program — decisions, conditions, loops, and data flows.

It is a **white-box testing technique** that supplements basis path testing by providing **targeted tests** for specific constructs in the code.
This method helps ensure that all **possible control paths** and **logical branches** behave as expected under various conditions.

**Types of Control Structure Testing**
1. **Condition Testing**
2. **Data Flow Testing**
3. **Loop Testing**

Each of these is explained below.

### 10.5.1 Condition Testing

**Definition**
Condition testing focuses on the **logical conditions** (Boolean expressions) that control the execution of program statements.

The goal is to ensure that all possible outcomes of each decision condition are tested at least once.
**Objectives**
- Detect logical errors in conditions.
- Verify that all parts of a composite condition are evaluated correctly.
- Identify incorrect relational or logical operators.

**Example**
```
if ((A > B) && (C == D))
   printf("Valid");
```

**Test Cases Should Cover:**
1. Both conditions true → print "Valid".
2. A > B true, C == D false.
3. A > B false, C == D true.
4. Both false.

This ensures every component of the decision expression is exercised.

**Common Techniques**

| **Technique** | **Description** |
|---|---|
| **Simple Condition Testing** | Each condition tested independently. |
| **Compound Condition Testing** | All combinations of conditions evaluated. |
| **Relational Condition Testing** | Tests boundary values in relational operators. |
| **Boolean Operator Testing** | Tests each AND, OR, NOT operation outcome. |

### 10.5.2 Data Flow Testing

**Definition**

Data flow testing examines the **lifecycle of data variables** — from their definition to their use.

It helps detect **uninitialized variables**, **incorrect data usage**, and **redundant definitions**.

**Basic Idea**

Each variable has:

- **Definition (def):** where it is assigned a value.
- **Use (use):** where its value is accessed.
- **Kill (kill):** where its lifetime ends.

Data flow testing ensures that all valid *def-use* pairs are tested.

**Example**

```
1: int x;
2: x = 10;      // def(x)
3: if (x > 0)
4:     y = x + 2; // use(x)
```

**Objective:** Verify that variable x is defined before it is used, and not redefined or killed prematurely.

**Common Anomalies Detected**

| Anomaly | Description |
|---|---|
| **DU anomaly** | Variable used before definition. |
| **DD anomaly** | Variable defined twice before being used. |
| **UK anomaly** | Variable used after being killed or out of scope. |

### 10.5.3 Loop Testing

**Definition**

Loop testing validates the **correctness of iterative constructs**, such as for, while, or do-while loops.

It ensures that the loop executes the correct number of times under different conditions.

**Objectives**

- Validate initialization and termination conditions.
- Test off-by-one errors and boundary conditions.
- Detect infinite or skipped loops.

**Typical Loop Test Cases**

| Type | Test Case Example |
|---|---|
| **Zero Iterations** | Verify loop skipped correctly when condition false initially. |
| **One Iteration** | Ensure loop executes once correctly. |
| **Multiple Iterations** | Test normal operation (e.g., 3–5 times). |
| **Maximum Iterations** | Validate loop limit conditions. |
| **Beyond Maximum** | Ensure termination when exceeding limit. |

**Example**

```
for i in range(1, 6):
   print(i)
```

Test with:

- Start > End (loop never executes).
- Start = End (one iteration).
- Normal range (five iterations).

**Advantages of Control Structure Testing**

- High defect detection in logic and flow.
- Ensures all program paths are evaluated.
- Supports automated tools (e.g., static analyzers).

**Limitations**

- Requires access to source code.
- Complex for large, nested control structures.
- Focuses on control logic, not functionality.

## 10.6 BLACK-BOX TESTING

**Introduction**

Black-box testing, also known as **behavioral or functional testing**, focuses on the **external behavior** of software without considering its internal logic or structure.
It answers the question:
"Does the software perform what it is supposed to do?"

**Objectives**

1. Verify functional correctness according to requirements.
2. Validate input/output behavior.
3. Identify missing or incorrect functionalities.
4. Ensure proper error and boundary handling.

**Common Black-Box Testing Techniques**

1. **Graph-Based Testing Methods**
2. **Equivalence Partitioning**
3. **Boundary Value Analysis**
4. **Orthogonal Array Testing**

### 10.6.1 Graph-Based Testing Methods

These methods model program behavior as a **graph of nodes and edges**, representing inputs, states, and transitions.

**Example:**

In a menu-driven system, each menu is a node, and user actions are edges.
Graph-based testing ensures that **all menu combinations and transitions** are validated.

### 10.6.2 Equivalence Partitioning

**Concept**

Input data is divided into **equivalence classes** (partitions) such that all values in a class are treated similarly by the program.
Testing just one value from each partition is sufficient to represent all values.

**Example**

For a function that accepts integers between **1 and 100**:

| **Partition** | **Representative Value** | **Expected Result** |
|---|---|---|
| Less than 1 | 0 | Invalid |

| | | |
|---|---|---|
| 1 to 100 | 50 | Valid |
| Greater than 100 | 150 | Invalid |

### 10.6.3 Boundary Value Analysis (BVA)

**Concept**

Since most errors occur at boundaries, testing should focus on values at, below, and above boundaries.

**Example**

For valid input range 1–100, test values:

0, 1, 2, 99, 100, 101.

### 10.6.4 Orthogonal Array Testing (OAT)

Used when the number of input combinations is large.
OAT uses a **mathematical matrix** (orthogonal array) to systematically select a **small but representative set** of combinations for testing.

**Example**

In a system with 3 parameters (each having 3 possible values), instead of 27 combinations, OAT may require only 9.

**Advantages of Black-Box Testing**

- Applicable early (before code is available).
- Based on user requirements, not design.
- Effective for detecting missing or incorrect functionalities.

**Limitations**

- Limited coverage of internal logic.
- Redundant test cases possible.
- Difficult to determine internal error causes.

## 10.7 OBJECT ORIENTED TESTING METHODS

Object-oriented (OO) software development introduces a paradigm shift in both program design and testing. Unlike procedural systems where functions and data are separate, OO systems encapsulate data and behavior within classes and objects. This encapsulation fundamentally alters the way software is tested.
Traditional testing techniques focus on procedural control flow — verifying the correctness of input–output transformations in individual functions. However, OO software emphasizes interacting objects, class relationships, and dynamic behaviors that emerge only during runtime.

**The Nature of Testing in OO Systems**

In OO systems, testing must validate not only:

- Individual methods and operations,
- But also, how objects interact, inherit behavior, and respond to polymorphic calls.

Testing, therefore, needs to address both static structure (class hierarchy and relationships) and dynamic behavior (object states and message passing).

**Unique Characteristics of OO Testing**

| OO Feature | Testing Challenge |
|---|---|
| **Encapsulation** | Limits direct access to internal data — testers must rely on public interfaces. |
| **Inheritance** | Changes in superclass behavior can affect all derived classes, requiring regression testing. |
| **Polymorphism** | The same message can invoke different methods at runtime; dynamic binding complicates coverage. |
| **Dynamic Interaction** | Objects may collaborate in complex ways that cannot be predicted from static code inspection. |

**Objectives of OO Testing**

1. Validate object states and method behaviors.
2. Ensure inter-object communication works correctly.
3. Detect faults caused by inheritance and dynamic binding.
4. Guarantee consistency between class specifications and actual implementations.
5. Build reusable and automated test cases for evolving OO architectures.

**Why Specialized Testing Methods Are Needed**

Conventional testing techniques (like control-flow testing) are insufficient for OO software because:

- They focus on functions, not objects.
- They do not consider class hierarchies or message passing.
- They cannot easily trace dynamic object interactions during runtime.

OO testing methods therefore extend and adapt traditional strategies, introducing fault-based, scenario-based, and hierarchical test approaches.

### 10.7.1 The Test-Case Design Implications of Object-Oriented Concepts

Object-oriented concepts influence how test cases are selected, designed, and executed. Because classes encapsulate both data (attributes) and behavior (methods), the test designer must ensure that test cases cover all relevant combinations of object states and interactions.

#### 10.7.1 . (i) Key Object-Oriented Concepts Affecting Testing

**1. Encapsulation**

Encapsulation hides implementation details.
This means that internal data cannot be directly tested — tests must access it indirectly through the class's public interface.
Implication:
Test cases should:

- Verify that each method correctly manipulates internal data.
- Ensure that the class invariant remains valid after every public operation.

**2. Inheritance**

Inheritance allows subclasses to reuse and extend the behavior of parent classes.
Implication:

- Changes in a superclass require retesting of all derived classes (regression testing).
- Test cases for parent classes must be reapplied to subclasses.
- Subclass extensions need new test cases for overridden or additional behavior.

**3. Polymorphism**

Polymorphism allows the same operation name to invoke different methods depending on the object type.

Implication:

- All binding variations of a polymorphic message must be tested.
- Testers must verify that each implementation correctly fulfills the intended behavior.
- Additional runtime tests are required since method binding occurs dynamically**.**

**4. Dynamic Binding**

Dynamic (late) binding defers method resolution until runtime.

Implication:

- Complete path coverage cannot be determined statically.
- Test cases should simulate different runtime configurations to observe binding behavior.
- Automated test frameworks (e.g., JUnit, NUnit) can help capture runtime execution paths.

**5. Object State and Identity**

Each object maintains a unique identity and a set of states.

Implication:

- Test cases must verify state transitions as defined by state diagrams.
- Multiple test cases may be required to cover all valid and invalid transitions.
- Object identity testing ensures that distinct objects maintain independent states.

**10.7.1.(ii) General Guidelines for OO Test Design**

| **Guideline** | **Description** |
|---|---|
| Design tests from class specifications | Use contracts, preconditions, and postconditions as the test basis. |
| Ensure coverage of all methods | Each operation must be tested for all relevant input combinations. |
| Test interactions and collaborations | Focus on message passing between cooperating classes. |
| Test state-dependent behavior | Validate transitions using class/state diagrams. |
| Maintain traceability | Map test cases to class design elements and use cases. |

**10.7.1. (iii) Example – Implications Illustrated**

Consider a Shape superclass with subclasses Circle, Square, and Triangle, each implementing a draw() method.

| **Concept** | **Implication for Testing** | **Example Test Case** |
|---|---|---|
| Inheritance | Verify that subclass overrides maintain superclass contract. | Test draw() in each subclass to ensure correct rendering. |
| Polymorphism | Confirm correct method dispatch for runtime type. | Call shape.draw() where shape points to different subclass objects. |
| Encapsulation | Ensure internal coordinates are updated correctly. | After move(x, y), validate position using public getters. |

### 10.7.1. (iv). OO Test Case Design Example

**Let's design test cases for a class BankAccount:**

| **Method** | **Test Objective** | **Example Test Input** | **Expected Output** |
|---|---|---|---|
| deposit(amount) | Validate correct balance update. | deposit(100) | Balance increases by 100. |
| withdraw(amount) | Test boundary conditions (zero, overdraft). | withdraw(0), withdraw(2000) | Error or rejection if invalid. |
| transfer(targetAccount, amount) | Test collaboration with another object. | transfer(acc2, 500) | Balances updated in both accounts. |

Here, the last test involves inter-object collaboration, illustrating how OO testing extends beyond single-function verification.

### 10.7.2. Applicability of Conventional Test-Case Design Methods

**Introduction**

Although object-oriented software introduces new design paradigms, many traditional test-case design methods remain useful and relevant.
Methods such as equivalence partitioning, boundary value analysis, and state transition testing can be adapted to the OO context with modifications to account for class structure and object interactions.

### 10.7.2.(i) Conventional Techniques Still Applicable

| **Traditional Method** | **OO Adaptation** | **Example Application** |
|---|---|---|
| Equivalence Partitioning | Define partitions for method inputs and object states. | Divide valid/invalid ranges for method parameters or attributes. |
| Boundary Value Analysis (BVA) | Identify boundary conditions for class attributes and inherited variables. | Test withdraw(amount) at balance limit boundaries. |
| Decision Table Testing | Use decision logic embedded within methods or event handlers. | Verify response of calculateInterest() under various conditions. |
| State-Based Testing | Apply to object states and lifecycle transitions. | Test transitions between "Active", "Suspended", and "Closed" states. |
| Use-Case Testing | Derive from interactions among collaborating objects. | Test entire user scenario such as "Process Online Order". |

### 10.7.2.(ii) Applying Conventional Methods to Classes

At the Method Level

Each method within a class can be treated as a small functional unit.
For example:

- Apply equivalence partitioning to method parameters.
- Use BVA for numerical or range-based inputs.
- Apply decision testing for conditional logic.

At the Class Level

When testing the class as a whole:

- Consider the class as a state machine.
- Identify transitions between object states.

- Apply state transition diagrams to derive tests for valid/invalid transitions.

At the Integration Level

Conventional data flow and control flow testing can be extended to message passing:

- Replace function calls with object messages.
- Track the sequence of method invocations across collaborating objects.

### 10.7.2.(iii) Enhancements Needed for OO Systems

| **Aspect** | **Enhancement Required** |
|---|---|
| Data Access | Due to encapsulation, tests must access state indirectly through accessor methods. |
| Inheritance | Regression testing needed when parent classes change. |
| Polymorphism | Ensure all runtime bindings of a polymorphic message are tested. |
| Dynamic Behavior | Use sequence diagrams and interaction diagrams for dynamic test generation. |

### 10.7.2.(iv) Example – Adaptation of BVA to OO Class

**Class: TemperatureSensor**

| **Attribute** | **Range** | **Test Values** |
|---|---|---|
| temperature | -50°C to 150°C | -50, -49, 0, 149, 150, 151 |
| sensorID | Must be positive integer | 0, 1, 2, -1 |

**Each attribute's boundaries become targets for BVA test cases.**

Conventional test methods remain foundational for OO testing but must be augmented with OO-specific considerations such as class relationships, message sequences, and inheritance-based dependencies.

This hybrid approach leverages the strength of traditional methods while embracing the complexity of OO architectures.

## 10.8 Testing for Specialized Environments, Architectures, and Applications

Modern software systems increasingly operate in **heterogeneous and specialized environments**.Unlike standalone applications, these systems interact with users, hardware, databases, and networks in **real time**.

As a result, **traditional testing techniques**—which assume sequential execution and single-threaded control—are no longer sufficient.

The diversity of environments has given rise to **environment-specific testing strategies**, each addressing unique challenges such as:

- Complex **event-driven behaviors** (in GUIs),
- **Distributed processing and network reliability** (in client–server systems),
- **Documentation accuracy** and user guidance validation, and
- **Timing, concurrency, and synchronization** (in real-time systems).

**Testing for specialized environments thus requires:**

1. A deep understanding of the **architecture** and **interaction patterns**.
2. The use of **simulation**, **automation**, and **monitoring tools**.
3. Integration of **functional**, **performance**, and **usability** metrics.

In this lesson, we explore the four major specialized testing areas discussed in Pressman's *Software Engineering – A Practitioner's Approach* (Section 5.10):

- Testing Graphical User Interfaces (GUIs)
- Testing Client–Server Architectures
- Testing Documentation and Help Facilities
- Testing Real-Time Systems

Each of these environments imposes **unique requirements** on test design, execution, and evaluation.

### 10.8.1 Testing GUIs (Graphical User Interfaces)

#### (i) Overview

Graphical user interfaces are **event-driven systems**.

User interactions trigger sequences of events that invoke underlying application logic. Testing GUIs ensures that every visual component and user action results in the correct response.

#### (ii) Objectives of GUI Testing

- Verify the correctness of all visual elements and controls.
- Ensure proper navigation between screens and dialogs.
- Validate event handling and input validation mechanisms.
- Test cross-platform consistency and layout rendering.
- Evaluate usability, accessibility, and responsiveness.

#### (iii) GUI Testing Challenges

| **Challenge** | **Description** |
|---|---|
| **Event Explosion** | Thousands of possible event sequences. |
| **Platform Diversity** | Different browsers, resolutions, and devices. |
| **Dynamic Layouts** | Responsive design and localization changes. |
| **Human-Centered Usability** | Subjective perception of design and flow. |

❖ **Event Explosion**

A major challenge in GUI testing is the event explosion problem—the exponential growth in the number of possible event sequences that can occur during user interaction. Unlike procedural programs with predictable control flow, GUI-based systems respond to a virtually infinite combination of user actions such as clicks, drags, gestures, menu selections, or keyboard inputs.Each sequence of events can produce a different application state, making exhaustive testing practically impossible.To address this, testers must use finite-state modeling, event-pair coverage, and model-based testing techniques to prioritize the most critical and high-risk event paths while maintaining reasonable test effort and coverage.

❖ **Platform Diversity**

Modern GUI applications are expected to run seamlessly on multiple platforms—different browsers, operating systems, screen resolutions, and device types.
This diversity leads to challenges such as inconsistent rendering of visual components, varying font sizes, missing controls, or differences in event-handling behavior across platforms.For example, a web button may function properly in Chrome but fail to display or respond correctly in Safari or Edge.Effective testing in such environments requires the use of cross-platform automation frameworks (e.g., Selenium Grid, BrowserStack) and responsive design validation to ensure visual and functional consistency across all supported configurations.

❖ **Dynamic Layouts**
Dynamic layouts refer to interfaces that adjust in real time to screen size, user preferences, or content changes — a key feature of responsive web and mobile applications.Such flexibility complicates testing because UI components may shift position, resize, or change visibility dynamically, making traditional coordinate-based or object-recognition tests unreliable.Moreover, real-time content loading using AJAX or asynchronous APIs further adds to the complexity.To manage this, GUI test scripts must employ object identifiers (XPath, CSS locators) that adapt to layout variations and should be reinforced with visual validation tools capable of detecting layout shifts, missing elements, and design regressions automatically.

❖ **Human-Centered Usability**
GUI testing extends beyond functional validation to include human-centered usability evaluation, which assesses how intuitively and efficiently users can interact with the software.This aspect is inherently subjective and focuses on factors like layout clarity, feedback quality, color contrast, accessibility, and cognitive load.A functionally correct interface can still fail usability tests if it causes user confusion or fatigue. Therefore, usability testing combines automated validation with manual, heuristic-based reviews and user experience (UX) studies to ensure the GUI aligns with human expectations, accessibility standards (WCAG 2.1), and overall user satisfaction.

**(iv) Finite-State Model for GUI Testing**
Each interface screen can be treated as a **state**, and each user action as a **transition**. Finite-State Modeling (FSM) systematically enumerates possible sequences.
**Example:**
[Login] → [Home] → [Settings] → [Logout]
Each transition path defines a distinct test case (e.g., Login → Home → Logout).

**(v) GUI Testing Process**

| **Step** | **Description** |
|---|---|
| 1. Identify GUI Elements | Buttons, menus, forms, icons, dialogs. |
| 2. Model Event Flow | Create event-driven state diagrams. |
| 3. Design Test Cases | Include valid/invalid sequences. |
| 4. Automate Tests | Record/replay actions using test scripts. |
| 5. Execute Regression Tests | Re-run after GUI changes. |

**(vi) Tools for GUI Testing**

| **Tool** | **Environment** | **Functionality** |
|---|---|---|
| Selenium WebDriver | Web | Scripted automation. |
| Appium | Mobile | Cross-platform testing. |
| Ranorex / TestComplete | Desktop | Object-based event testing. |
| QTP/UFT | Enterprise | Record-and-playback testing. |
| Sikuli | All | Image-based testing for GUIs. |

**1. Selenium WebDriver**
**Selenium WebDriver** is an open-source framework designed for automating web-based application testing across browsers and operating systems.

It controls the browser directly using native automation APIs instead of relying on JavaScript execution.

**Key Features**

- Cross-browser testing (Chrome, Firefox, Edge, Safari).
- Supports multiple programming languages (Java, Python, C#, Ruby, JavaScript).
- Integration with CI/CD tools such as Jenkins, Maven, and Docker.
- Ability to handle dynamic web elements and AJAX-based content.
- Parallel test execution via Selenium Grid.

**Architecture**

- **Client Libraries:** Language bindings for different programming languages.
- **JSON Wire Protocol / W3C Protocol:** Facilitates communication between client and browser driver.
- **Browser Drivers:** ChromeDriver, GeckoDriver, EdgeDriver, etc., that control specific browsers.
- **Browser Instance:** Executes commands and returns results.

**Use Cases**

- Functional and regression testing of web applications.
- Data-driven and keyword-driven testing frameworks.
- Integration with TestNG or JUnit for test orchestration.

**Advantages**

- Free and community-supported.
- Highly extensible and customizable.
- Works with major browsers and operating systems.

**Limitations**

- Supports only web applications (no desktop or mobile).
- No built-in object repository or reporting.
- Requires skilled programming knowledge.

**2. Appium**

**Appium** is an open-source test automation framework for **mobile applications**. It allows testing of **native, hybrid, and mobile web apps** on both **Android** and **iOS** platforms using the WebDriver protocol.

**Key Features**

- Cross-platform support using a single API.
- No need to recompile or modify the app under test.
- Supports testing in multiple languages (Java, Python, C#, Ruby, JS).
- Compatible with mobile browsers (e.g., Chrome, Safari).
- Integrates with CI/CD and cloud-based test services (BrowserStack, Sauce Labs).

**Architecture**

- **Appium Server:** Acts as a REST server written in Node.js.
- **Appium Client:** Sends automation commands.
- **Mobile JSON Wire Protocol:** Communicates between client and device.
- **Device Drivers:** UIAutomator2 (Android), XCUITest (iOS).

**Use Cases**

- Testing login flows, gesture interactions, and mobile UI consistency.
- Regression testing across multiple OS versions.
- Cloud-based parallel execution of mobile test suites.

**Advantages**

- Unified automation for Android and iOS.
- Supports native device features (GPS, camera, notifications).

- Easily integrates with Selenium frameworks.

**Limitations**

- Slower execution compared to device-native frameworks.
- Complex setup for real device testing.
- Limited support for older OS versions.

**3. Ranorex / TestComplete**

**Ranorex** and **TestComplete** are commercial GUI testing tools designed for desktop, web, and mobile applications.

They provide record-and-playback functionality, scripting, and built-in object recognition engines.

**Key Features**

- Supports Windows, macOS, web, and mobile apps.
- Codeless test creation with advanced scripting (C#, VB.NET, Python).
- Robust object identification using XPath and image recognition.
- Data-driven and keyword-driven testing capabilities.
- Built-in reporting and test analytics dashboards.

**Architecture**

- **Object Repository:** Centralized storage of UI elements.
- **Recording Engine:** Captures user actions for playback.
- **Test Executor:** Executes automated scripts.
- **Reporting Module:** Generates test execution reports.

**Use Cases**

- Desktop GUI validation (e.g., accounting or ERP systems).
- Regression testing of web portals.
- Multi-environment test automation with minimal coding.

**Advantages**

- User-friendly interface for non-programmers.
- Integrated debugging and reporting features.
- Strong support for visual verification testing.

**Limitations**

- Commercial licensing cost.
- Resource-intensive on large test suites.
- Platform-dependent features may vary.

**4. QTP/UFT (QuickTest Professional / Unified Functional Testing)**

**QTP/UFT**, developed by **Micro Focus**, is a widely used commercial tool for functional and regression testing of desktop, web, and packaged applications.

It supports both **keyword-driven** and **VBScript-based** test automation.

**Key Features**

- Supports multiple application technologies (web, Java, SAP, Oracle, .NET).
- Record-and-playback for quick test development.
- Integration with HP ALM (Application Lifecycle Management).
- Advanced object repository for GUI components.
- Built-in test reporting and debugging tools.

**Architecture**

- **Test Design Interface:** Allows test case creation using keywords or VBScript.
- **Object Repository:** Stores application objects for reuse.
- **Test Execution Engine:** Controls application and logs results.
- **Result Viewer:** Generates detailed HTML reports.

**Use Cases**

- Functional testing of enterprise web and desktop systems.
- End-to-end regression testing in corporate environments.
- Integration testing with databases and APIs.

**Advantages**

- Easy to learn for testers with minimal coding experience.
- Strong integration with enterprise testing ecosystems.
- Comprehensive object recognition engine.

**Limitations**

- High licensing cost.
- Limited cross-platform support (mainly Windows).
- Slower execution for large web applications.

**5. Sikuli**

**Sikuli** is an open-source visual automation tool that uses **image recognition** to automate GUI operations.
It can interact with any visible element on the screen — web, desktop, or even within remote desktop sessions.

**Key Features**

- Uses screenshots as references for UI elements.
- Supports all types of graphical environments (no DOM dependency).
- Built on Java; supports scripting in Jython and Java.
- Capable of automating legacy or Flash-based systems.

**Architecture**

- **Sikuli IDE:** Visual editor for writing and executing scripts.
- **Image Matching Engine:** Uses OpenCV for pattern recognition.
- **Script Runner:** Executes visual actions (click, type, drag).
- **API Library:** For integration with Python and Java applications.

**Use Cases**

- Automating image-based workflows and legacy systems.
- Testing games, Flash, or video-based interfaces.
- Automating repetitive desktop operations.

**Advantages**

- Works with any visible GUI element regardless of technology.
- No access to application source code is required.
- Simple to learn and use for quick automation.

**Limitations**

- Dependent on image accuracy; sensitive to resolution changes.
- Not ideal for large-scale or complex test suites.
- Limited reporting and debugging facilities.

**Summary Comparison Table**

| **Tool** | **Type** | **Supported Platforms** | **Key Strength** | **Limitation** |
|---|---|---|---|---|
| **Selenium WebDriver** | Open-source | Web | Cross-browser automation | Web only |
| **Appium** | Open-source | Mobile (Android, iOS) | Unified API for mobile | Slower, complex setup |
| **Ranorex / TestComplete** | Commercial | Desktop, Web, Mobile | Record–playback, strong reporting | Licensing cost |

| | | | | |
|---|---|---|---|---|
| **QTP/UFT** | Commercial | Desktop, Web | Enterprise integration | High cost, Windows-centric |
| **Sikuli** | Open-source | All visual UIs | Image-based automation | Sensitive to UI changes |

**(viii) Example – Login Page Testing**

| **Test Case** | **Input** | **Expected Result** |
|---|---|---|
| Valid credentials | Correct username & password | Dashboard opens. |
| Invalid credentials | Wrong username/password | Error message. |
| Blank fields | Empty inputs | “Required fields” alert. |
| Browser compatibility | Chrome, Firefox | Consistent layout. |

❖ **Valid Credentials**
This test verifies that when a user enters the **correct username and password**, the system successfully logs in and displays the main application screen or dashboard. It confirms that the authentication process works properly for legitimate users.

❖ **Invalid Credentials**
This test checks how the system handles **wrong username or password** entries. The expected result is that access is denied and an **error message** (such as “Invalid Login” or “Incorrect Password”) is displayed, without revealing sensitive information.

❖ **Blank Fields**
This case tests the scenario where the user **does not enter any input** in the username or password fields and clicks the login button. The system should display a **validation message** like “Username and Password are required,” ensuring input completeness before submission.

❖ **Browser Compatibility**
This test ensures that the login page behaves **consistently across different web browsers** (e.g., Chrome, Firefox, Edge, Safari). It checks that all visual elements (buttons, forms, labels) and functionality (form submission, navigation) work correctly on each platform.

**(viii) Best Practices**

- Use model-based testing for event sequences.
- Test accessibility (keyboard navigation, color contrast).
- Combine automated regression with exploratory testing.
- Prioritize high-frequency user actions.

### 10.8.2 Testing Client–Server Architectures

**(i) Overview**

Client–server software distributes functionality across clients and servers connected via a network.
Testing must confirm that both ends operate correctly — independently and together — while ensuring data consistency, reliability, and performance.

**(ii) Levels of Client–Server Testing**

| **Level** | **Objective** |
|---|---|
| 1. Client Application Testing | Validate client behavior and user interface independently. |
| 2. Integration Testing | Test client–server communication and API correctness. |
| 3. System Testing | Evaluate complete architecture, including network behavior. |

**(iii) Common Testing Types**

- **Functional Testing:** Verifies client requests and server responses.
- **Database Testing:** Ensures transaction consistency and integrity.
- **Performance Testing:** Measures response time and throughput.
- **Security Testing:** Validates encryption, authentication, and access control.
- **Recovery Testing:** Assesses fault tolerance under failures.

**(iv) Tools and Frameworks**

| **Tool** | **Purpose** |
|---|---|
| Apache JMeter | Load and performance testing. |
| Postman | REST API testing. |
| Wireshark | Network traffic analysis. |
| SoapUI | Web service validation. |
| DbUnit | Database verification. |

**1. Apache JMeter**

Apache JMeter is an open-source tool developed by the Apache Software Foundation, primarily used for performance, load, and stress testing of client–server applications. It simulates multiple concurrent users sending requests to a target server, thereby evaluating the system’s scalability and reliability.

**Key Features**

- Supports multiple protocols: HTTP, HTTPS, FTP, JDBC, SOAP, REST, and JMS.
- Provides thread groups to simulate virtual users.
- Offers detailed graphs and reports on response time, throughput, and error rates.
- Enables parameterization and scripting through JSR223 and Groovy.
- Integration with CI/CD tools such as Jenkins for automated performance testing.

**Use Cases**

- Load testing of web servers, APIs, and databases.
- Stress testing to determine system breaking points.
- Baseline performance comparison before and after updates.

**Advantages**

- 100% free and open source.
- Highly extensible via plugins.
- Supports distributed testing across multiple machines.

**Limitations**

- Requires technical knowledge for complex scenarios.
- GUI mode consumes memory during high-load tests.

**2. Postman**

Postman is a powerful API testing platform used to develop, send, and verify API requests and responses.

It provides a user-friendly interface for testing RESTful and SOAP-based web services.

**Key Features**

- Supports HTTP methods (GET, POST, PUT, DELETE, PATCH).
- Allows creating collections of API requests.
- Provides built-in JavaScript scripting for pre- and post-test validation.
- Enables automated testing via the Newman CLI.
- Integration with CI/CD tools and version control (Git).

**Use Cases**
- Functional and regression testing of REST APIs.
- Validating server response codes, headers, and payload data.
- Testing authentication mechanisms (OAuth 2.0, JWT, API keys).

**Advantages**
- Intuitive GUI suitable for developers and testers.
- Supports both manual and automated testing.
- Provides real-time API documentation and mock servers.

**Limitations**
- Primarily limited to API-level testing (not end-to-end).
- Limited performance/load-testing capabilities.

### 3. Wireshark

Wireshark is an open-source network packet analyzer used to inspect data flowing between clients and servers in real time.It is essential for diagnosing network-level issues, verifying protocol compliance, and analyzing traffic security.

**Key Features**
- Captures and decodes live network packets.
- Supports hundreds of network protocols (TCP, IP, HTTP, SSL/TLS, DNS, etc.).
- Provides real-time filtering and color coding for packet types.
- Enables decryption and inspection of SSL/TLS sessions (with proper keys).
- Allows export of captured data for further analysis.

**Use Cases**
- Network debugging and troubleshooting.
- Identifying performance bottlenecks due to packet loss or latency.
- Verifying secure communication in client–server applications.

**Advantages**
- Free and open source.
- Deep inspection of network traffic.
- Suitable for both network administrators and QA engineers.

**Limitations**
- Steep learning curve for beginners.
- Generates very large log files for long capture sessions.

### 4. SoapUI

SoapUI is a dedicated testing tool for SOAP and REST web services.
It provides both open-source and commercial (ReadyAPI) versions for functional, security, and performance testing of APIs**.**

**Key Features**
- Supports SOAP, REST, GraphQL, and JMS protocols.
- Offers drag-and-drop test case creation.
- Facilitates data-driven testing using external files (CSV, Excel, databases).
- Includes built-in assertions for validating API responses.
- Supports security tests such as SQL injection, XML bombs, and fuzzing.

**Use Cases**
- Functional and regression testing of web services.
- API contract and schema validation.
- Security and load testing of endpoints.

**Advantages**

- Comprehensive API testing capabilities.
- Graphical interface for complex API workflows.
- Easy integration with CI/CD tools and Jenkins.

**Limitations**

- GUI can be resource-intensive.
- Learning curve for advanced scripting.

**5. DbUnit**

DbUnit is a Java-based testing framework designed for database testing. It integrates with JUnit and helps manage test data consistency by comparing database contents before and after test execution.

**Key Features**

- Supports importing and exporting data sets (XML, CSV, Excel).
- Automates validation of database state during integration testing.
- Enables rollback of changes after test completion to maintain a clean environment.
- Easily integrates with Java-based build tools like Maven and Ant.

**Use Cases**

- Verifying database CRUD (Create, Read, Update, Delete) operations.
- Ensuring data integrity and consistency after transactions.
- Validating stored procedures and triggers.

**Advantages**

- Automates database verification tasks.
- Works well with continuous integration pipelines.
- Ensures database remains in a known state for each test.

**Limitations**

- Designed primarily for Java ecosystems.
- Limited GUI support (script-based configuration).

| **Tool** | **Category** | **Primary Use** | **Key Strength** | **Limitation** |
|---|---|---|---|---|
| **Apache JMeter** | Performance & Load Testing | Simulate user load and analyze performance | Scalable and extensible | Complex for beginners |
| **Postman** | API Testing | Validate RESTful & SOAP APIs | User-friendly, automated validation | No load testing |
| **Wireshark** | Network Analysis | Monitor and debug client–server traffic | Deep protocol inspection | Complex packet data |
| **SoapUI** | Web Service Testing | Test and secure SOAP/REST services | Security & data-driven testing | Resource-heavy GUI |
| **DbUnit** | Database Testing | Validate database integrity and consistency | JUnit integration, data rollback | Java-specific |

In client–server testing, no single tool covers all aspects of validation. JMeter ensures performance under load, Postman validates functional correctness of APIs, Wireshark inspects network-level data flow, SoapUI ensures web service integrity, and DbUnit maintains database consistency.

### 10.3.5 Case Study – Online Reservation System

A client–server application was tested at three levels:

- **Client-side:** UI validation of booking form.
- **Integration:** API correctness for payment gateway.
- **System:** 500 concurrent users simulated via JMeter.

Outcome: 98% success rate with average response time < 1.5 seconds.

**Testing Objectives**

The primary goals of testing were to:

1. Validate the **functional accuracy** of both client and server components.
2. Ensure **secure and reliable communication** between the client, application server, and database.
3. Measure **system performance** under heavy concurrent user load.
4. Verify the **integration** of the external payment gateway and transaction logging.

**Testing Approach**

The application was tested at three hierarchical levels, consistent with client–server testing methodology:

**1. Client-Side Testing**

The **user interface (UI)** of the reservation module was validated on multiple browsers (Chrome, Edge, and Firefox).

Tests included:

- Verification of mandatory input fields (origin, destination, travel date).
- Validation of date pickers, seat selection, and confirmation dialogs.
- Error-handling tests for invalid or incomplete entries.
- Cross-browser layout and rendering checks for consistent appearance.

Automated scripts created using **Selenium WebDriver** executed 120 test cases covering navigation, input validation, and responsive behavior.
Minor UI issues, such as misaligned icons and inconsistent font sizes, were detected and corrected during early iterations.

**2. Integration Testing**

Integration testing focused on validating the communication between the **client interface, application server, and payment gateway APIs**.

Using **Postman** and **SoapUI**, testers verified that API calls correctly transmitted booking data and payment details.

Assertions were defined to confirm:

- Accurate HTTP status codes and response payloads.
- Correct encryption and authentication using OAuth 2.0 tokens.
- Proper handling of failed payment responses (timeouts, invalid card). Transaction integrity was further checked by comparing client logs with corresponding database entries through **DbUnit** scripts.

**3. System and Performance Testing**

For full-scale performance evaluation, **Apache JMeter** simulated **500 concurrent virtual users** executing typical reservation workflows.

Key metrics observed included:

- Average response time per transaction < 1.5 seconds.
- 98 percent overall success rate for all simulated transactions.
- Peak throughput of 2,300 requests per minute.
- CPU usage < 75 percent and database connection pooling efficiency within expected thresholds.

**Wireshark** traces were analyzed concurrently to ensure that all network communications were properly encrypted (TLS 1.3) and free from retransmission or packet-loss anomalies.

**(vi) Best Practices**

- Create separate test environments for client and server.
- Simulate peak user loads.
- Monitor real-time network statistics.
- Perform regression after each backend update.

### 10.8.3 Testing Documentation and Help Facilities

**(i) Importance**

- Documentation is part of the software configuration.
- Inaccurate documentation can lead to user frustration and operational failure. Testing ensures alignment between actual system behavior and written instructions.

**(ii) Phases of Documentation Testing**

1. **Technical Review (Static Testing)**
   - Check grammar, structure, and completeness.
   - Ensure consistent terminology.
2. **Live Testing (Dynamic Testing)**
   - Follow documentation while using the system.
   - Verify that expected outcomes match actual system responses.

**(iii) Methods**

| **Method** | **Purpose** |
|---|---|
| **Graph-Based Testing** | Trace navigation through help topics. |
| **Equivalence Partitioning** | Group valid and invalid commands. |
| **Boundary Value Analysis** | Verify example input limits. |
| **Model-Based Testing** | Compare documentation models to program execution. |

**(iv) Example**

Manual: *“Select ‘Save As’ → Enter file name → Click Save.”* Observed: System prompts “File Exists” even on new names → documentation error. Result: Updated user guide to reflect confirmation behavior.

**(v) Tools**

- Acrolinx – Language consistency analysis.
- Selenium Docs – Automated help link validation.
- MadCap Analyzer – Documentation cross-link testing.

**(vi) Best Practices**

- Maintain documentation under version control.
- Update documentation in sync with software releases.
- Perform periodic usability reviews.

### 10.8.4 Testing for Real-Time Systems

**(i) Definition**

A real-time system must process input and deliver responses **within a defined time frame**. Failure to meet timing deadlines can cause critical failures in control, aviation, or healthcare systems.

**(ii) Four-Step Real-Time Testing Strategy**

| **Step** | **Description** |
|---|---|
| **1. Task Testing** | Verify each independent task for logic and computation accuracy. |
| **2. Behavioral Testing** | Simulate events to observe system response. |
| **3. Intertask Testing** | Examine synchronization among concurrent tasks. |
| **4. System Testing** | Integrate hardware and software to validate end-to-end timing. |

**(iii) Timing and Performance Analysis**

**Metrics:**

- Response latency
- Deadline hit ratio
- Average jitter
- CPU and memory utilization

**Tools:** Tracealyzer, Perf, NI LabVIEW Real-Time Monitor.

**(iv) Interrupt and Concurrency Testing**

| **Aspect** | **Focus** |
|---|---|
| Interrupt Priority | Correct scheduling and queue management. |
| Concurrency | Deadlocks and race conditions. |
| Shared Data | Consistency under parallel access. |

**(v) Simulation and Hardware-in-the-Loop**

Simulators reproduce external stimuli while hardware-in-the-loop (HIL) connects real hardware devices to software for full-scale testing.
Tools: dSPACE, NI PXI, Simulink Real-Time.

**( vi) Case Study – Air Traffic Control Subsystem**

- Update interval: every 0.5 seconds.
- Interrupt handling verified for 200 simultaneous inputs.
- Fail-safe alarm triggered in 2.2 seconds under overload.

**(vii) Best Practices**

- Combine simulation with actual hardware.
- Test under stress and fault conditions.
- Log event timestamps for timing verification.
- Include power failure and sensor fault scenarios.

## 10.10 Summary

Software testing is a **critical activity** in software engineering that ensures the developed software performs as intended, meets user expectations, and is free from defects. It involves the **systematic execution of programs** to uncover errors and verify that all requirements have been correctly implemented.
The **fundamentals of software testing** revolve around key principles such as test planning, test case design, controlled execution, and results evaluation. Testing provides both **verification** ("Are we building the product right?") and **validation** ("Are we building the right product?").
Two major perspectives define testing approaches:

- The **internal view (white-box testing)**, which focuses on the logic and structure of the code.
- The **external view (black-box testing)**, which examines functionality without knowledge of the internal implementation.

In **white-box testing**, methods like **basis path testing** are used to derive test cases from control flow graphs.

Key techniques include:

- **Flow Graph Notation** – visualizing program logic.
- **Independent Program Paths** – identifying linearly independent paths for coverage.
- **Control Structure Testing** – including condition, data flow, and loop testing.
- In **black-box testing**, the focus shifts to functional behavior and system inputs/outputs.

Popular methods include:

- **Graph-Based Testing** – analyzing relationships between inputs, outputs, and system components.
- **Equivalence Partitioning** – dividing input data into valid and invalid partitions.
- **Boundary Value Analysis** – testing at input limits.
- **Orthogonal Array Testing** – reducing test combinations while maintaining high coverage.

Testing in **Object-Oriented (OO) Systems** introduces new challenges due to features like encapsulation and inheritance. Testers must validate **class hierarchies**, **polymorphic behavior**, and **object interactions**. Conventional test design methods are still applicable but require adaptation to the OO paradigm.
**Testing specialized environments** such as GUIs, client–server systems, and real-time applications demands domain-specific strategies:

- **GUI Testing** verifies layout, navigation, and user interactions.
- **Client–Server Testing** evaluates distributed communication, transaction integrity, and network performance.
- **Documentation Testing** ensures accuracy between software behavior and user manuals.
- **Real-Time Testing** validates timing, concurrency, and event response.
- Finally, the concept of **Testing Patterns** provides a structured, reusable framework for solving recurring testing problems.

Patterns such as **Data-Driven Testing**, **Page Object Model**, and **Risk-Based Testing** capture best practices and standardize testing processes across projects.

In essence, software testing integrates **methodology, automation, and continuous improvement**. By applying structured techniques and reusable patterns, organizations can ensure that their software systems are **robust, maintainable, and aligned with user needs**.

## 10.11 Technical Terms

1. Verification
2. Validation
3. White-Box Testing
4. Black-Box Testing
5. Basis Path Testing
6. Data Flow Testing
7. Loop Testing
8. Equivalence Partitioning
9. Boundary Value Analysis
10. Orthogonal Array Testing
11. Object-Oriented Testing
12. GUI Testing
13. Client–Server Testing
14. Real-Time System Testing
15. Testing Patterns

## 10.12 Self-Assessment Questions

### Essay Questions

1. Define software testing and explain its fundamental objectives.
2. Differentiate between internal and external views of software testing.
3. Explain **white-box testing** and describe the steps involved in **basis path testing**.
4. What are independent program paths? How are test cases derived from them?
5. Discuss **control structure testing** and explain condition, data flow, and loop testing.
6. Compare **white-box** and **black-box** testing techniques with suitable examples.
7. Describe **equivalence partitioning** and **boundary value analysis** with illustrations.
8. Explain the challenges and methods of **testing object-oriented software**.
9. How are **specialized environments** like GUIs and real-time systems tested?
10. What are **software testing patterns**? Discuss their classification and benefits.

### Short Questions

1. Verification vs. Validation
2. Flow Graph Notation
3. Independent Program Path
4. Data Flow Testing
5. Orthogonal Array Testing
6. OO Test-Case Design Implications
7. GUI Testing Challenges

8. Client–Server Testing Levels
9. Testing Documentation and Help Facilities
10. Page Object Model (POM) Pattern

## 10.13 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Myers, Glenford J., *The Art of Software Testing*, 3rd Ed., Wiley, 2011.
3. Beizer, Boris, *Software Testing Techniques*, 2nd Ed., Dreamtech Press, 2003.
4. Desikan, S. & Ramesh, G., *Software Testing: Principles and Practices*, Pearson Education, 2006.
5. Jorgensen, Paul C., *Software Testing: A Craftsman's Approach*, CRC Press, 2018.
6. Meszaros, Gerard, *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley, 2007.
7. Burnstein, Ilene, *Practical Software Testing*, Springer, 2003.
8. Kaner, Cem, Falk, Jack & Nguyen, Hung Q., *Testing Computer Software*, Wiley, 1999.
9. Sommerville, Ian, *Software Engineering*, Pearson Education, 10th Ed., 2015.
10. Binder, Robert V., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.

Dr. Vasantha Rudrarnalla

## LESSON- 11

# PROJECT MANAGEMENT

**AIMS AND OBJECTIVES**

To introduce the management principles, people issues, process considerations, and product-related factors that influence successful software project execution.

**After completing this lesson, you will be able to:**

1. Describe the management spectrum and its importance in software engineering.
2. Understand the roles and responsibilities of people in software projects.
3. Define software scope and perform problem decomposition.
4. Explain how to integrate the product with the process effectively.
5. Identify key project management principles and challenges.
6. Apply the W$^{6}$HH Principle to planning and control.
7. Appreciate how leadership, communication, and coordination ensure project **success.**

**STRUCTURE**

## 11.1 The Management Spectrum

Software project management involves balancing people, product, process, and project dimensions.Effective management ensures that these four aspects operate cohesively to deliver a high-quality product within schedule and budget.A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project.

A square diagram with four quadrants labeled:

- People – stakeholders, managers, teams.
- Product – software to be developed.
- Process – framework and methods used.
- Project – planning, tracking, and control.

This is known as the Management Spectrum, which defines how resources and responsibilities are distributed.

### 11.1.1 The People

People are the most critical factor in software project success. Even with advanced tools and methodologies, a project fails without motivated, skilled, and coordinated individuals.

**Key Roles:**

- Project Manager – leads planning, monitoring, and decision-making.
- Developers – design, code, and test software components.
- Testers – verify and validate functionality.
- Analysts – define requirements and system scope.
- Customers/Stakeholders – provide feedback and acceptance.

Effective project management requires understanding human behavior, fostering communication, and maintaining morale.
The people capability maturity model defines the following key practice areas for software people: staffing, communication and coordination, work environment, performance management, training, compensation, competency analysis and development, career development, workgroup development, team/culture development, and others. Organizations that achieve high levels of People-CMM maturity have a higher likelihood of implementing effective software project management practices.

### 11.1.2 The Product

The “product” defines what is to be built — its objectives, functionality, and performance. Managers must clearly understand the product scope, constraints, and quality expectations. Without a well-defined product vision, estimation and scheduling become inaccurate.

**Example:**
In an *Online Banking Project*, the product includes user authentication, transaction processing, report generation, and mobile integration. Each function affects scope and resource allocation.Once the product objectives and scope are understood, alternative solutions are considered. Although very little detail is discussed, the alternatives enable managers and practitioners to select a “best” approach, given the constraints imposed by delivery deadlines, budgetary restrictions, personnel availability, technical interfaces, and myriad other factors.

### 11.1.3 The Process

The process defines *how* the product will be built. It provides a structured framework for planning, executing, and controlling project tasks. A process model (e.g., Agile, Spiral, Waterfall) determines activity flow, deliverables, and quality checkpoints.

**Example:**
In Agile development, iterative sprints deliver working software every few weeks, allowing early feedback and adaptation.

### 11.1.4 The Project

The project integrates all elements — people, product, and process — within defined constraints of time, cost, and scope.

**Effective project management involves:**

- Planning – defining tasks, schedules, and resources.
- Monitoring – tracking progress and identifying risks.
- Controlling – applying corrective actions when deviations occur.

A triangle labeled with three corners: *Cost*, *Time*, *Quality*, forming the "Project Management Triangle."
To avoid project failure, a software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project.

## 11.2 People

### 11.2.1 The Stakeholders

Stakeholders include everyone affected by the project — customers, users, sponsors, developers, and testers.
Understanding stakeholder expectations and maintaining communication is essential.
The software process (and every software project) is populated by stakeholders who can be categorized into one of five constituencies:

**1. Senior managers** who define the business issues that often have a significant influence on the project. Given these statistics, it's reasonable to ask how the impact of computers continues to grow exponentially. Part of the answer, I think, is that a substantial number of these "failed" projects are ill conceived in the first place. Customers lose interest quickly (because what they've requested wasn't really as important as they first thought), and the projects are cancelled.

**2. Project (technical) managers** who must plan, motivate, organize, and control the practitioners who do software work.

**3. Practitioners** who deliver the technical skills that are necessary to engineer a product or application.

**4. Customers** who specify the requirements for the software to be engineered and other stakeholders who have a peripheral interest in the outcome.

**5. End users** who interact with the software once it is released for production use.

### 11.2.2 Team Leaders

A team leader bridges management and technical teams. They motivate, mentor, and maintain discipline.Successful leaders possess strong communication, empathy, and conflict resolution skills.
**MOI model of leadership:**
**Motivation.** The ability to encourage (by "push or pull") technical people to produce to their best ability.

**Organization.** The ability to mold existing processes (or invent new ones) that will enable the initial concept to be translated into a final product.
**Ideas or innovation.** The ability to encourage people to create and feel creative even when they must work within bounds established for a particular software product or application.

### 11.2.3 The Software Team

Software teams vary by project type:

- Democratic/Decentralized Team – decisions made collectively.
- Controlled Centralized Team – decisions flow from a leader.
- Mixed Structure – balances autonomy and guidance

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

- Difficulty of the problem to be solved
- "Size" of the resultant program(s) in lines of code or function points
- Time that the team will stay together (team lifetime)
- Degree to which the problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of sociability (communication) required for the project.

**To achieve a high-performance team:**

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to
- be maintained.

### 11.2.4 Agile Teams

Agile teams emphasize collaboration, flexibility, and customer involvement. They are cross-functional, including designers, testers, and analysts who work together in short iterations.

**Example:**
A Scrum team consists of a *Product Owner*, *Scrum Master*, and *Development Team*.
Many agile process models (e.g., Scrum) give the agile team significant autonomy to make the project management and technical decisions required to get the job done. Planning is kept to a minimum, and the team is allowed to select its own approach (e.g., process, methods, tools), constrained only by business requirements and organizational standards. As the project proceeds, the team self-organizes to focus individual competency in a way that is most beneficial to the project at a given point in time. To accomplish this, an agile team might conduct daily team meetings to coordinate and synchronize the work that must be accomplished for that day. Based on information obtained during these meetings, the team adapts its approach in a way that accomplishes an increment of work. As each day passes, continual self-organization and collaboration move the team toward a completed software increment.

### 11.2.5 Coordination and Communication

Coordination ensures tasks integrate smoothly; communication prevents misunderstandings.

**Techniques include:**

- Daily stand-ups
- Task boards
- Project management tools (e.g., Jira, Trello)
- Regular feedback sessions

A network showing communication links among team members, emphasizing clear pathways.
To deal with them effectively, you must establish effective methods for coordinating the people who do the work. To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established. Formal communication is accomplished through "writing, structured meetings, and other relatively non-interactive and impersonal communication channels" [Kra95]. Informal communication is more personal. Members of a software team share ideas on an ad hoc basis, ask for help as problems arise, and interact with one another on a daily basis.

## 11.3 The Product

### 11.3.1 Software Scope

Defining scope sets project boundaries.

**It specifies:**
**Context.** How does the software to be built fit into a larger system, product, or business context, and what constraints are imposed as a result of the context?

**Information objectives.** What customer-visible data objects are produced as output from the software? What data objects are required for input?

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

Software project scope must be unambiguous and understandable at the management and technical levels. A statement of software scope must be bounded. That is, quantitative data (e.g., number of simultaneous users, target environment, maximum allowable response time) are stated explicitly, constraints and/or limitations (e.g., product cost restricts memory size) are noted, and mitigating factors (e.g., desired algorithms are well understood and available in Java) are described.A well-defined scope avoids scope creep, a major cause of delays.

### 11.3.2 Problem Decomposition

Once the scope is clear, the system is decomposed into smaller, manageable parts (modules or components).
This process, known as modularization, allows parallel development and easier management.

**Example:**
In a Hospital Management System, the problem may be decomposed into:

- Patient Management
- Pharmacy Management
- Billing System
- Appointment Scheduling
- A tree diagram showing the decomposition of the main system into subsystems and modules.

As the statement of scope evolves, a first level of partitioning naturally occurs. The project team learns that the marketing department has talked with potential customers and found that the following functions should be part of automatic copy editing: (1) spell checking, (2) sentence grammar checking, (3) reference checking for large documents (e.g., Is a reference to a bibliography entry found in the list of entries in the bibliography?), (4) the implementation of a style sheet feature that imposed consistency across a document, and (5) section and chapter reference validation for large documents. Each of these features represents a subfunction to be implemented in software. Each can be further refined if the decomposition will make planning easier.

## 11.4 The Process

### 11.4.1 Melding the Product and the Process

The process must align with product characteristics.

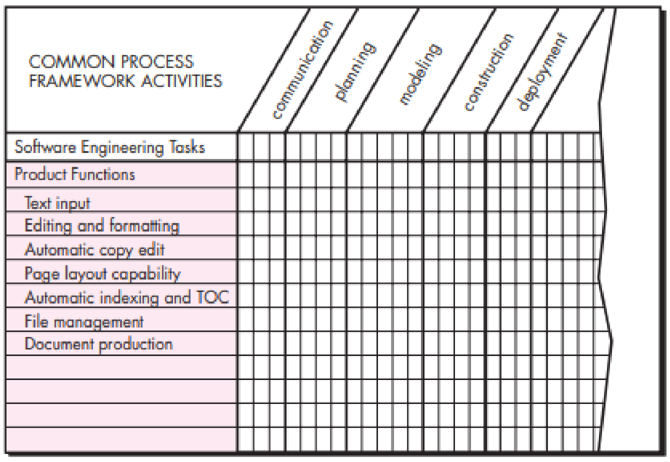


**Fig 11.1 Melding the problem and the process**

For example, a safety-critical system (like air traffic control) needs a rigorous, verified process, whereas a mobile app may use a rapid, iterative approach.

Project managers must select a process model (Waterfall, Incremental, Agile, Spiral) that best fits:

- Complexity of the product.
- Development team experience.
- Risk and change tolerance.

Assume that the organization has adopted the generic framework activities— communication, planning, modeling, construction, and deployment— discussed in Chapter 2. The team members who work on a product function will apply each of the framework activities to it. In essence, a matrix similar to the one shown in Figure 11.1 is created. Each major product function (the figure notes functions for the word-processing software discussed earlier) is listed in the left-hand column. Framework activities are listed in the top row. Software engineering work tasks (for each framework activity) would be entered in the following row.5 The job of the project manager (and other team members) is to estimate resource requirements for each matrix cell, start and end dates for the tasks associated with each cell, and work products to be produced as a consequence of each task.

### 11.4.2 Process Decomposition

The process can also be decomposed into phases:
1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment

Each phase produces deliverables reviewed for completeness and quality.
A waterfall-like flow diagram illustrating how each process phase feeds into the next.
But actual work tasks do vary. Process decomposition commences when the project manager asks, “How do we accomplish this framework activity?” For example, a
small, relatively simple project might require the following work tasks for the communication activity:
1. Develop list of clarification issues.
2. Meet with stakeholders to address clarification issues.
3. Jointly develop a statement of scope.
4. Review the statement of scope with all concerned.
5. Modify the statement of scope as required

Now, consider a more complex project, which has a broader scope and more significant business impact. Such a project might require the following work tasks for
the **communication:**
1. Review the customer request.
2. Plan and schedule a formal, facilitated meeting with all stakeholders.
3. Conduct research to specify the proposed solution and existing approaches.
4. Prepare a “working document” and an agenda for the formal meeting.
5. Conduct the meeting.
6. Jointly develop mini-specs that reflect data, functional, and behavioral
7. features of the software. Alternatively, develop use cases that describe the
8. software from the user’s point of view.
9. Review each mini-spec or use case for correctness, consistency, and lack of
10. ambiguity.
11. Assemble the mini-specs into a scoping document.
12. Review the scoping document or collection of use cases with all
13. concerned.
14. Modify the scoping document or use cases as required.

## 11.5 The Project

Project management involves balancing scope, schedule, and resources.

**Activities include:**
- Estimation of effort, time, and cost.
- Scheduling milestones and deliverables.
- Monitoring progress with performance metrics (e.g., Earned Value Analysis).

**Example:**
For a 12-week web application project, milestones may include:
- Week 1–2: Requirements finalized

- Week 3–5: Design and modeling
- Week 6–10: Coding and testing
- Week 11–12: Integration and deployment

A Gantt chart illustrating task durations and dependencies.

**John Reel [Ree99] defines 10 signs that indicate that an information systems project is in jeopardy:**

1. Software people don't understand their customer's needs.
2. The product scope is poorly defined.
3. Changes are managed poorly.
4. The chosen technology changes.
5. Business needs change [or are ill defined].
6. Deadlines are unrealistic.
7. Users are resistant.
8. Sponsorship is lost [or was never properly obtained].
9. The project team lacks people with appropriate skills.
10. Managers [and practitioners] avoid best practices and lessons learned.

**Reel [Ree99] suggests a five-part commonsense approach to software projects:**

1. **Start on the right foot.** This is accomplished by working hard (very hard) to understand the problem that is to be solved and then setting realistic objectives and expectations for everyone who will be involved in the project. It is reinforced by building the right team and giving the team
2. autonomy, authority, and technology needed to do the job.
3. **Maintain momentum.** Many projects get off to a good start and then slowly disintegrate. To maintain momentum, the project manager must provide incentives to keep turnover of personnel to an absolute minimum, the team should emphasize quality in every task it performs, and senior management should do everything possible to stay out of the team's way.
4. **Track progress.** For a software project, progress is tracked as work products (e.g., models, source code, sets of test cases) are produced and approved (using technical reviews) as part of a quality assurance activity. In addition, software process and project measures can be collected and used to assess progress against averages developed for the software development organization.
5. **Make smart decisions.** In essence, the decisions of the project manager and the software team should be to "keep it simple." Whenever possible, decide to use commercial off-the-shelf software or existing software components or patterns, decide to avoid custom interfaces when standard approaches are available, decide to identify and then avoid obvious risks, and decide to allocate more time than you think is needed to complex or risky tasks .
6. **Conduct a postmortem analysis.** Establish a consistent mechanism for extracting lessons learned for each project. Evaluate the planned and actual schedules, collect and analyze software project metrics, get feedback from team members and customers, and record findings in written form.

## 11.6 The W$^6$HH Principle

The W$^6$HH Principle, introduced by Barry Boehm, is one of the most widely adopted frameworks for software project planning and management.

It provides a set of seven guiding questions—six beginning with "W" and one with "H"—that ensure all critical aspects of a project are clearly defined and communicated before development begins.
This principle acts as a managerial checklist, helping software engineers and project managers to ensure completeness, traceability, and control in project planning.

**Purpose of the W$^6$HH Principle**
In software engineering, projects often fail not because of technical issues, but because of poor planning, unclear objectives, and miscommunication.
The W$^6$HH framework addresses these weaknesses by prompting project teams to think through every essential element: purpose, scope, responsibility, location, methodology, schedule, and cost.
Each question focuses on a vital dimension of planning, together forming a holistic project definition.

The Seven Questions of the W$^6$HH Principle

| **Question** | **Purpose** |
|---|---|
| Why is the system being developed? | Clarifies objectives and the business justification for undertaking the project. |
| What will be done? | Defines the scope, major deliverables, and success criteria. |
| When will it be completed? | Establishes project milestones, deadlines, and schedules. |
| Who is responsible? | Assigns roles, ownership, and accountability within the team. |
| Where are they located? | Identifies physical or virtual locations of teams, facilities, and resources. |
| How will it be accomplished? | Defines the development process, methodology, tools, and technologies to be used. |
| How much will it cost? | Determines budget allocation, resource cost, and financial feasibility. |

Each question, when thoroughly answered, provides clarity, alignment, and measurable control over project execution.

**Detailed Explanation of Each Element**

**1. Why is the System Being Developed?**
This question defines the purpose and motivation behind the project.
It aligns the software objectives with business strategy and user needs.
A clear "why" prevents scope drift and ensures the project delivers tangible value.
Example:
A healthcare organization wants to automate patient data management.
*Why?* To improve patient service, reduce administrative errors, and enable data-driven decision-making.
Key Management Output: Project Vision Document and Business Case.

**2. What Will Be Done?**
This addresses scope definition—the boundaries of the project and its expected outcomes.
It includes identifying deliverables such as software modules, documentation, training materials, and support systems.

**Example:**
For a *Hospital Management System*:
*What?* Develop modules for Patient Registration, Billing, Pharmacy, and Reporting.
Key Management Output: Software Requirements Specification (SRS) and Project Scope Statement.

**3. When Will It Be Completed?**
This determines the schedule and timeline, including key milestones, deadlines, and dependencies. Proper scheduling ensures time-bound progress and facilitates early detection of delays.

**Example:**
For an e-commerce project:
*When?* Minimum Viable Product (MVP) in 3 months; Full launch in 6 months.
Key Management Output: Project Schedule, Gantt Charts, and Milestone Chart.

**4. Who is Responsible?**
This defines roles and responsibilities for each team member and stakeholder. Assigning ownership avoids duplication of effort and clarifies accountability.

**Example:**
- *Project Manager* – oversees progress.
- *Business Analyst* – gathers requirements.
- *Developers* – code modules.
- *Testers* – ensure quality.
- *Customer Representative* – validates requirements.

Key Management Output: Responsibility Assignment Matrix (RAM) and RACI Chart.

**5. Where Are They Located?**
In modern software projects, teams are often distributed across different geographical locations.
This question ensures logistical planning—determining where resources, infrastructure, and support teams will operate.

**Example:**
- Development Team – Offshore (India)
- QA and Deployment – Onsite (U.S.)
- Client – Remote collaboration through Agile tools

Addressing "where" helps in planning communication channels, time zone coordination, and infrastructure support.
Key Management Output: Communication Plan and Infrastructure Layout.

**6. How Will It Be Accomplished?**
This question focuses on methodology and technical approach—how the team will deliver the system.
It includes choosing development models (Waterfall, Agile, Spiral), tools, technologies, testing frameworks, and version control mechanisms.

**Example:**
*How?* Using Agile Scrum methodology, two-week sprints, GitHub for source control, and Jenkins for CI/CD.
Key Management Output: Software Development Plan (SDP) and Process Guidelines.

**7. How Much Will It Cost?**
This addresses budget estimation and cost control.
Accurate cost estimation includes:
- Personnel costs
- Hardware/software resources
- Training and support
- Risk contingencies

**Example:**
For a banking application:

*How much?* ₹75 lakhs (including infrastructure, labor, and maintenance).
Key Management Output: Cost Baseline and Budget Tracking Sheets.

**Example: Applying $W^6HH$ to a Banking Project**

| Question | Banking Project Response |
|---|---|
| Why? | To enable secure online transactions and digital account management. |
| What? | Develop a responsive web-based and mobile-enabled banking platform. |
| When? | MVP in 6 months; Final release in 10 months. |
| Who? | Development and QA teams; managed by a project manager and supported by IT Ops. |
| Where? | Hybrid model – development offshore, deployment on client infrastructure. |
| How? | Agile methodology using sprints, code reviews, and automated testing. |
| How much? | ₹75 lakhs total cost with a 10% contingency reserve. |

**Outcome:**
A clear, structured plan that defines *purpose, responsibilities, timelines, and resources*—reducing ambiguity and improving coordination.

**Importance of the $W^6HH$ Principle**

2. Comprehensive Planning:
   Encourages managers to think through all aspects before development begins.
3. Improved Communication:
   Ensures all stakeholders share a common understanding of project goals.
4. Better Risk Management:
   Identifies uncertainties early through explicit questioning.
5. Enhanced Accountability:
   Assigns clear ownership for each task and deliverable.
6. Traceability and Transparency:
   Every decision and commitment can be linked to a specific question and answer.
7. Adaptability:
   The framework fits all methodologies—Agile, Spiral, or traditional.

## 11.7 Summary

Software project management integrates people, product, process, and project dimensions into a unified framework for achieving quality software under constraints. People are the core drivers of success, requiring effective leadership, motivation, and communication. The product defines what and *why*, while the process determines *how* the system is built. Decomposition simplifies complexity, enabling manageable work units and parallel progress. Project management provides planning, tracking, and corrective control. Finally, the $W^6HH$ principle ensures structured, question-based planning for better visibility and accountability. Together, these elements form the foundation of effective software engineering management.

## 11.8 Technical Terms

1. Management Spectrum
2. Stakeholder
3. Scope
4. Problem Decomposition
5. Agile Team
6. Coordination
7. Process Model
8. Project Estimation
9. Milestone
10. $W^6HH$ Principle

## 11.9 Self-Assessment Questions

### Essay Questions

1. Explain the Management Spectrum and its four key dimensions.
2. Discuss the role of people in software project success.
3. What is the importance of defining software scope?
4. Describe problem decomposition with suitable examples.
5. How are product and process related in project management?
6. Explain the $W^6HH$ principle and its relevance to project planning.
7. Discuss various team structures in software engineering.
8. What are the challenges of coordination and communication in teams?
9. Illustrate how process models influence project success.
10. Write short notes on Agile Teams and Leadership in software projects.

### Short Questions
1. The People Factor
2. Software Scope
3. Process Decomposition
4. Project Triangle
5. $W^6HH$ Questions

## 11.10 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Sommerville, Ian, *Software Engineering*, 10th Ed., Pearson Education, 2015.
3. Boehm, Barry W., *Software Engineering Economics*, Prentice Hall, 1981.
4. Brooks, Frederick P., *The Mythical Man-Month*, Addison-Wesley, 1995.
5. Schwalbe, Kathy, *Information Technology Project Management*, Cengage Learning, 2018.
6. Royce, Winston, *Managing the Development of Large Software Systems*, IEEE, 1970.
7. Kerzner, Harold, *Project Management: A Systems Approach to Planning, Scheduling, and Controlling*, Wiley, 2017.
8. McConnell, Steve, *Rapid Development: Taming Wild Software Schedules*, Microsoft Press, 1996.

- Testers must verify that each implementation correctly fulfills the intended behavior.
- Additional runtime tests are required since method binding occurs dynamically.

Dr. Vasantha Rudrarnalla

LESSON- 12

# METRICS FOR PROCESS AND PROJECTS

## AIMS AND OBJECTIVES

To introduce the concept, importance, and application of **software metrics** in process and project management, focusing on how measurement supports quality improvement, control, and performance evaluation in software engineering.

**After completing this lesson, you will be able to:**

1. Define **software metrics** and explain their role in software process improvement.
2. Differentiate between **process metrics**, **project metrics**, and **product metrics**.
3. Describe the steps and challenges involved in **software measurement**.
4. Explain **size-oriented**, **function-oriented**, and **object-oriented metrics**.
5. Understand the importance of **software quality metrics** and **defect removal efficiency**.
6. Apply metrics to small organizations and understand scaling issues.
7. Establish and maintain a **software metrics program** within a development organization.
8. Interpret metrics to support **decision-making**, **process control**, and **continuous improvement**.

## STRUCTURE

## 12.1 INTRODUCTION TO SOFTWARE METRICS

Software engineering depends on measurement and quantitative assessment to manage complexity and ensure quality.
A software metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
Pressman states:
"Measurement is fundamental to engineering; it provides the quantitative basis for process control and improvement."
Without measurement, software management relies only on intuition.
Metrics bring objectivity, consistency, and traceability to engineering decisions.

**Examples of Software Metrics:**

- *Defect Density* — Number of defects per thousand lines of code (KLOC)
- *Productivity* — Lines of code (LOC) or Function Points (FP) per person-month
- *Cycle Time* — Average time required to complete a task or iteration
- *Customer Satisfaction Index* — Rating of end-user satisfaction

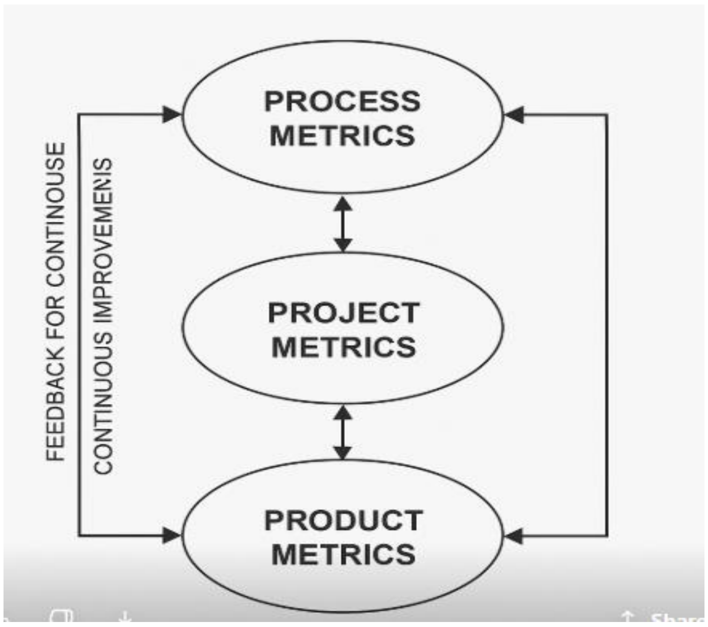


**Figure 12.1: Relationship between Process Metrics, Project Metrics, and Product Metrics (feedback loop for continuous improvement).**

This figure illustrates how process, project, and product metrics are interconnected in a continuous improvement cycle within software engineering. Each metric category provides feedback that influences the others, creating a closed-loop system for measurement, analysis, and enhancement.

1. **Process Metrics**
   - Measure the *efficiency and stability* of development activities (e.g., defect density, cycle time, rework percentage).
   - They provide feedback on how well the development process is functioning.
   - When analyzed, they highlight areas requiring process refinement — such as code review efficiency or testing effectiveness.

2. **Project Metrics**
   - Focus on *management and control aspects* of individual projects — such as cost, schedule variance, productivity, and effort.
   - They track progress and identify deviations from planned targets.
   - Data from project metrics feed back into process metrics, helping organizations determine whether the process supports predictable project performance.

3. **Product Metrics**
   - Evaluate *software quality attributes* — including size, reliability, maintainability, performance, and defect rates.
   - They provide a direct measure of product outcomes resulting from both process and project activities.
   - Defects or performance issues detected in the product trigger improvements in both project execution and process practices.

**The feedback loop ensures that information flows in all directions:**
- *Product metrics* inform *project managers* about software quality.
- *Project metrics* inform *process engineers* about process effectiveness.
- *Process improvements* enhance both future project execution and product outcomes.

This continuous loop supports data-driven decision-making and is central to Software Process Improvement (SPI) initiatives such as CMMI, Six Sigma, and ISO 9001.

## 12.2 MEASUREMENT IN SOFTWARE ENGINEERING

Software measurement helps achieve three goals:
1. Understanding – Analyze how the process behaves.
2. Control – Regulate process performance within predictable limits.
3. Improvement – Identify opportunities for enhancement.

Importance of Measurement
- Provides a basis for estimation and planning.
- Enables process comparison and benchmarking.
- Detects deviations from standards.
- Facilitates quantitative quality assurance.
- Supports CMMI and ISO 9001 maturity evaluation.

## 12.3 PROCESS METRICS

Process metrics evaluate the effectiveness and efficiency of software development activities. They help in understanding how processes affect quality, productivity, and cost.

### 12.3.1 Private and Public Metrics

| **Type** | **Used By** | **Purpose** |
|---|---|---|
| Private Metrics | Individual developers | For self-assessment and improvement (e.g., code reviews, defects found) |
| Public Metrics | Teams and managers | To evaluate process performance, resource use, and quality trends |

Private metrics help individuals grow, while public metrics aid organizational learning.

### 12.3.2 Process Metrics and Process Improvement

Process metrics are the foundation of Software Process Improvement (SPI) programs. They measure process health and maturity.

**Typical Process Metrics:**

| Metric | Purpose |
|---|---|
| Defect Density | Measures errors per KLOC or Function Point |
| Rework Percentage | Assesses efficiency of error correction |
| Process Yield | Percentage of outputs passing verification |
| Review Efficiency | Percentage of defects detected early |
| Cycle Time | Time to complete each process phase |

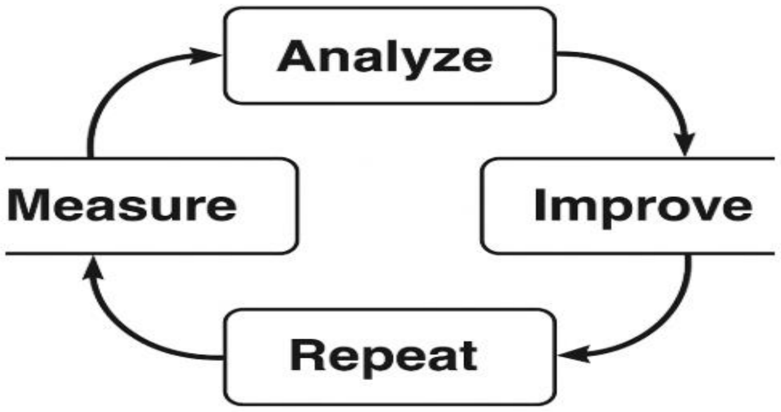


**Figure 12.2: Process Metrics Feedback Loop showing "Measure → Analyze → Improve → Repeat."**

This figure represents the cyclical nature of process measurement and improvement in software engineering. The feedback loop consists of four recurring stages — Measure, Analyze, Improve, and Repeat — forming the foundation of continuous process enhancement.

1. **Measure:**
   The first step involves collecting quantitative data on process activities. Typical measures include defect density, review efficiency, rework effort, or cycle time. Measurement must be consistent, objective, and aligned with organizational goals.
   - Example: Tracking the average number of defects per thousand lines of code (KLOC) across multiple projects.
2. **Analyze:**
   Collected data are interpreted and evaluated to identify trends, inefficiencies, and causes of variation. Statistical and graphical tools (e.g., control charts, Pareto analysis) are often used to pinpoint weaknesses in the process.
   - Example: If rework effort is increasing, analysis may reveal insufficient peer reviews or unstable requirements.
3. **Improve:**
   Based on analysis, corrective actions are planned and implemented to optimize process performance. Improvements may involve tool upgrades, staff training, or revising coding standards and testing practices.
   - Example: Introducing automated testing to reduce manual rework and improve defect detection.

4. **Repeat:**
After improvement, the process is re-measured to evaluate the effectiveness of implemented changes. This repetition creates a culture of ongoing learning and adaptation. Over time, the loop narrows the gap between current and desired performance.

The feedback loop embodies the principle of quantitative process management — a key element in Capability Maturity Model Integration (CMMI) Level 4 and 5 organizations. It ensures that decisions are based on empirical evidence rather than assumptions, leading to predictable quality outcomes.

## 12.4 PROJECT METRICS

Project metrics are used by managers to estimate, monitor, and control development activities.

### 12.4.1 Objectives of Project Metrics

- Track progress against schedule and cost.
- Identify risks and issues early.
- Evaluate resource utilization and productivity.
- Improve future project estimation accuracy.

### 12.4.2 Metrics for Estimation and Tracking

| **Metric** | **Meaning** |
|---|---|
| Size Metric (LOC, Function Points) | Basis for estimating effort and time |
| Effort Metric | Person-hours or person-months spent |
| Schedule Variance (SV) | Difference between planned and actual progress |
| Cost Variance (CV) | Deviation between planned and actual expenditure |
| Productivity Index | Output (LOC or FP) per effort unit |

**Earned Value Analysis (EVA):**

- CPI (Cost Performance Index) = EV / AC
- SPI (Schedule Performance Index) = EV / PV
- Values <1 indicate underperformance.

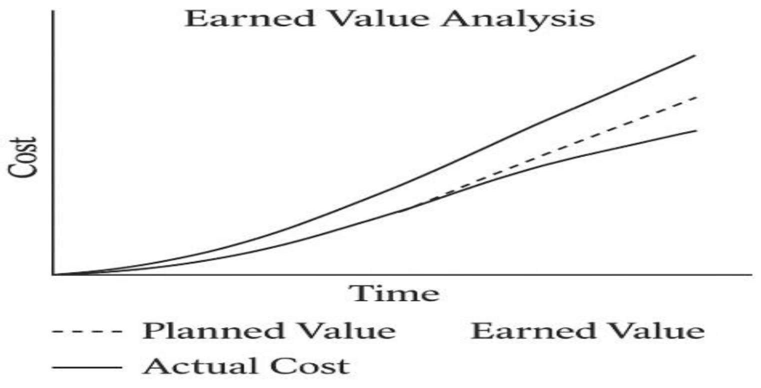


**Figure 12.3:Earned Value Analysis chart showing Planned Value, Earned Value, and Actual Cost lines.**

This figure illustrates the Earned Value Analysis (EVA) technique — a widely used project performance measurement tool in software engineering and project management. EVA integrates cost, schedule, and scope parameters to provide a quantitative assessment of project progress and efficiency.
The chart displays three primary curves plotted over time (x-axis) against cumulative cost or effort (y-axis):

1. **Planned Value (PV) – also called Budgeted Cost of Work Scheduled (BCWS)**
   - Represents the authorized budget for work planned to be completed by a specific date.
   - It is the baseline against which actual progress is compared.
   - *Example:* If ₹10 lakhs was planned for the first six months, PV = ₹10 lakhs at that point.
   - **Earned Value (EV) – also called Budgeted Cost of Work Performed (BCWP)**
   - Indicates the value of work actually completed, expressed in terms of the approved budget.
   - EV helps determine whether the project is ahead or behind schedule and under or over budget.
   - *Example:* If 80% of the planned work is done, EV = 0.8 × Total Budget.
   - **Actual Cost (AC) – also called Actual Cost of Work Performed (ACWP)**
   - Represents the actual expenditure incurred for the work completed so far.
   - It allows managers to evaluate cost performance relative to progress.

**12.4.3 Metrics in Agile Projects**
Agile teams use lightweight, real-time metrics instead of formal reports.

| **Metric** | **Description** |
|---|---|
| Velocity | Work completed per iteration (story points) |
| Lead Time | Time from feature request to delivery |
| Burndown Chart | Tracks remaining work across iterations |
| Defect Rate | Monitors product stability |
| Team Happiness | Gauges morale and collaboration |

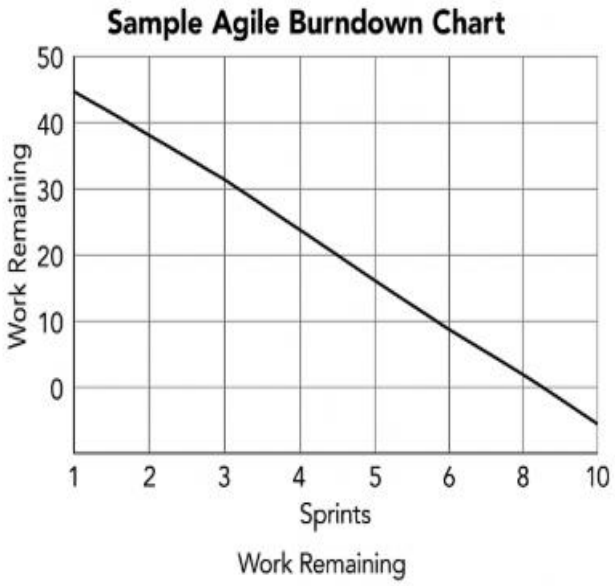


**Figure 12.4:Sample Agile Burndown Chart showing continuous progress across 10 sprints.**

## 12.5 SOFTWARE MEASUREMENT PRINCIPLES

**Pressman emphasizes six key principles:**

1. The objectives of measurement must be established before data collection.
2. Each metric should be derived from a defined model of software or process.
3. Metrics should be simple, objective, and computable.
4. Collect data consistently and interpret them cautiously.
5. Provide feedback to all stakeholders.
6. Use metrics to foster improvement, not punishment.

**Example:**

A developer's defect rate may be higher because they handle more complex modules — context matters in interpretation.

## 12.6 METRICS FOR SOFTWARE QUALITY

Quality metrics help determine how well software meets requirements and how reliable it is in operation.

### 12.6.1 Defect Metrics

| **Metric** | **Definition** |
|---|---|
| Defect Density | Defects per KLOC or FP |
| Defect Removal Efficiency (DRE) | DRE = (Defects removed before release / Total defects) × 100 |
| Mean Time to Repair (MTTR) | Average time to fix a failure |
| Customer-Reported Defects | Post-release defect count |

### 12.6.2 Reliability Metrics

| **Metric** | **Definition** |
|---|---|
| Mean Time Between Failures (MTBF) | Average operating time between system failures |
| Availability | Uptime ÷ (Uptime + Downtime) × 100 |
| Failure Rate | 1 / MTBF |
| Reliability Growth | Reduction in failure rate over time |

### 12.6.3 Complexity Metrics

| **Metric** | **Purpose** |
|---|---|
| Cyclomatic Complexity (McCabe) | Measures logical complexity of a program module |
| Halstead Metrics | Based on operators and operands count |
| Structural Complexity | Degree of inter-module dependency |
| Coupling and Cohesion | Assess modular design quality |

**Example:**
A module with a Cyclomatic Complexity >10 indicates high risk and requires focused testing.

## 12.7 METRICS FOR MAINTENANCE AND PROCESS IMPROVEMENT

**Maintenance consumes 60–80% of software cost; hence metrics are vital.**

| **Metric** | **Purpose** |
|---|---|
| Change Request Frequency | Measures stability of delivered product |
| Mean Time to Implement Change | Measures maintenance efficiency |
| Post-Release Defect Rate | Evaluates delivered quality |
| Maintainability Index | Quantifies ease of future changes |

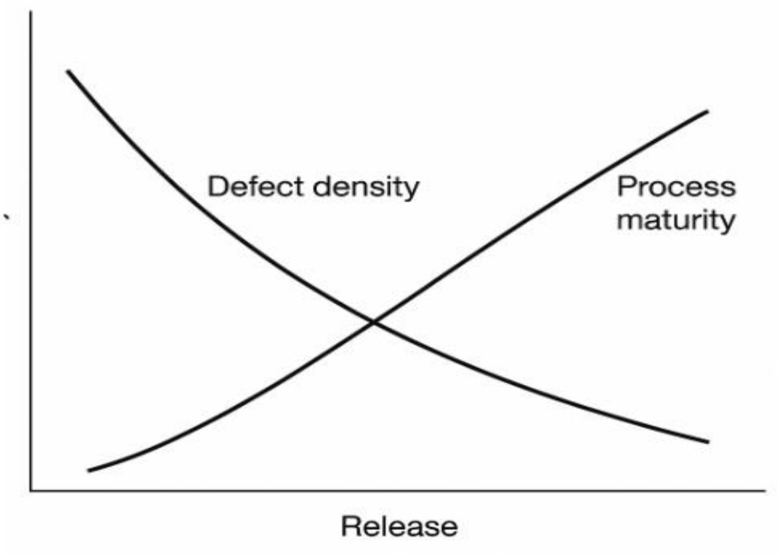


**Figure 12.5:Graph showing declining defect density and increasing process maturity over successive releases.**

This figure demonstrates the correlation between process maturity and product quality in software engineering. It shows how defect density decreases as process maturity improves across successive software releases.
The x-axis represents the sequence of software releases (Release 1, Release 2, Release 3, etc.), and the y-axes represent two complementary measures:

- Left Y-Axis: *Defect Density* (defects per KLOC or Function Points).
- Right Y-Axis: *Process Maturity Level* (qualitative or numerical scale, such as CMMI Levels 1–5).

Two curves are plotted:

1. **Defect Density Curve (Downward Slope)**
   - Indicates the number of defects per size unit in each release.
   - As process maturity improves, the number of defects found in each subsequent release declines significantly.
   - This downward trend represents better process control, early defect detection, and higher product reliability.
2. **Process Maturity Curve (Upward Slope)**
   - Shows the organization's process capability improvement over time (e.g., moving from ad hoc development to defined and optimized processes).
   - As practices such as code reviews, peer inspections, metrics-based management, and statistical process control are adopted, process maturity steadily increases.

**Example**

A software organization initially records 12 defects/KLOC in Release 1. By Release 5, after implementing formal reviews, automated testing, and root cause analysis, the defect density falls to 3 defects/KLOC. Simultaneously, their CMMI level rises from 2 (Repeatable) to 4 (Managed), showing a measurable improvement in process capability.

**Key Insights**

- Defect reduction is a measurable outcome of process improvement.
- Process maturity correlates strongly with predictability and quality.
- Continuous use of metrics feedback loops (Measure → Analyze → Improve → Repeat) sustains this trend.

As process maturity increases, software defects decrease — illustrating that quality is built into the process, not inspected into the product.

## 12.8 STATISTICAL SOFTWARE PROCESS IMPROVEMENT (SSPI)

- SSPI applies statistical process control (SPC) principles to software.
- It uses control charts, trend analysis, and process capability indices (Cp, Cpk) to monitor process stability.

**Benefits of SSPI:**

- Detects abnormal variations early.
- Identifies root causes of defects.
- Enables data-driven process tuning.

## 12.9 LIMITATIONS OF METRICS

- **Despite their importance, metrics have limitations:**
- Data collection can be costly.
- Poorly defined metrics can mislead.
- Overemphasis on numbers may ignore human factors.
- Context differences reduce comparability.
- Requires mature organizational culture for effective use.

## 12.10 SUMMARY

- Metrics provide a quantitative basis for understanding and improving software processes and projects.
- Process metrics evaluate efficiency; project metrics track progress and cost; product metrics assess quality.
- Measurement enables estimation, prediction, and control.
- Metrics must be used ethically and interpreted contextually to foster improvement.

## 12.11 TECHNICAL TERMS

Software Metric, Process Metric, Project Metric, Product Metric, Defect Density, DRE, MTBF, Complexity, Velocity, Burndown Chart, Earned Value, CPI, SPI, Function Point, Cyclomatic Complexity, Maintainability Index, SSPI.

## 12.12 SELF-ASSESSMENT QUESTIONS

***Essay Questions***

1. Define software metrics and explain their importance in software engineering.
2. Differentiate between process metrics and project metrics with examples.
3. Describe how metrics contribute to Software Process Improvement (SPI).
4. Explain the principles of software measurement.
5. Discuss various software quality metrics used in practice.

***Short Notes***

1. Earned Value Analysis (EVA)
2. Defect Removal Efficiency (DRE)
3. Cyclomatic Complexity
4. Process Maturity and SSPI
5. Agile Metrics

## 12.13 SUGGESTED READINGS

1. Roger S. Pressman, *Software Engineering – A Practitioner's Approach*, 6th Edition, TMH International.
2. Watts S. Humphrey, *Managing the Software Process*, Addison-Wesley.
3. Norman Fenton & James Bieman, *Software Metrics: A Rigorous and Practical Approach*, CRC Press.
4. Stephen H. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley.
5. ISO/IEC 9126: *Software Product Quality Model.*

Dr. Vasantha Rudrarnalla

LESSON- 13

# ESTIMATION

**AIMS AND OBJECTIVES**

To understand the principles, methods, and tools used in estimating the effort, cost, and resources required for software projects and to apply structured estimation techniques for accurate project planning.

**After completing this lesson, you will be able to :**

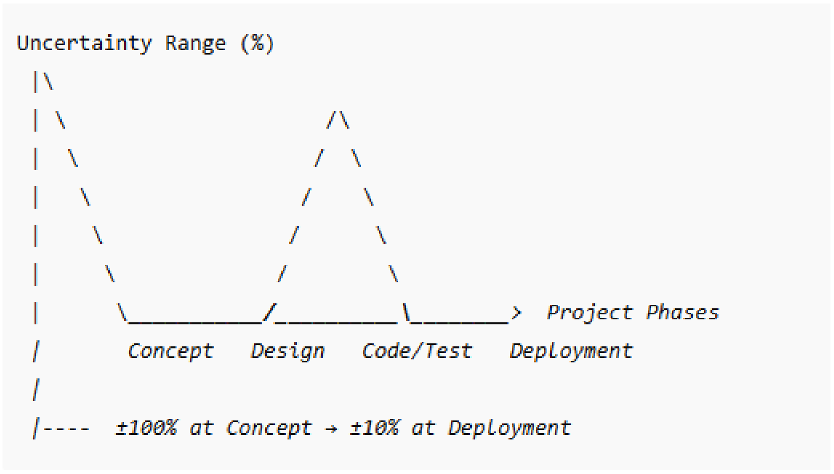
1. Explain the importance of estimation in successful software project planning.
2. Identify and describe the key components of software scope, feasibility, and resources.
3. Apply decomposition-based estimation techniques, including LOC, Function Points, and Use Cases.
4. Use empirical estimation models such as COCOMO II and the Software Equation.
5. Estimate resources for object-oriented, agile, and web-based projects.
6. Evaluate make-or-buy and outsourcing decisions through quantitative methods.
7. Integrate estimation results into the project planning process for better cost, schedule, and risk control.

**STRUCTURE**

## 13.1 Observations on Estimation

Estimation is prediction under uncertainty.Software effort cannot be measured directly before construction, so estimates rely on proxies (size, complexity, and experience).
Typical observations:

- Early estimates have ±100 % error; accuracy improves as project progresses.
- Under-estimation causes schedule slippage; over-estimation wastes budget.
- Good estimates require both quantitative models and expert judgment.
- Estimation should be iterative—revised after requirements, design, and coding stages.



```
Uncertainty Range (%)
 |\
 | \                  /\
 |  \                /  \
 |   \              /    \
 |    \            /      \
 |     \          /        \
 |      \__________/_________\_______>  Project Phases
 |       Concept   Design   Code/Test   Deployment
 |
 |----  ±100% at Concept → ±10% at Deployment
```

**Figure 13.1 – Estimation Uncertainty over the Project Life-Cycle**

A curve showing estimation uncertainty vs. project phase—wide range at concept stage narrowing toward closure. Shows how early estimates are less accurate and how uncertainty narrows as the project progresses. Accuracy improves as knowledge increases—emphasize iterative re-estimation after each major phase.
Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high. As a planner, you and the customer should recognize that variability in software requirements means instability in cost and schedule.
However, you should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible—although not always politically acceptable—to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

## 13.2 The Project Planning Process

Estimation is embedded within planning. Steps include:

1. Establish Scope – define system boundaries and objectives.
2. Estimate Size – LOC, Function Points, or Use Cases.
3. Derive Effort & Cost – using models or analogy.
4. Develop Schedule – allocate effort over phases.
5. Assess Risks – identify uncertainties affecting estimates.
6. Review & Refine – update as project data accumulate.

**Deliverables:**

- Work Breakdown Structure (WBS)
- Effort estimation sheet
- Schedule (Gantt/PERT)
- Resource plan

Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be adapted and updated as the project proceeds.

## 13.3 Software Scope and Feasibility

Scope defines *what* the software will do—inputs, outputs, data, and functions. Feasibility assesses whether scope is achievable within technical, economic, and schedule constraints.

**Feasibility factors:**

- Technical: Is required technology available?
- Economic: Cost $\leq$ benefits?
- Operational: Will users adopt it?
- Schedule: Can deadlines be met?

**Example:**
Developing an e-payment app—feasible only if secure transaction APIs and payment-gateway compliance exist.

## 13.4 Resources

### 13.4.1 Human Resources

Effort = person-months required.
Skill mix affects productivity—junior, senior, analyst, tester. Use staffing profile curves (Rayleigh distribution) to plan ramp-up and ramp-down.

### 13.4.2 Reusable Software Resources

Reuse lowers effort through libraries, frameworks, templates. Adjust estimate by reuse factor:

$$E_{new} = E_{base} \times (1 - r)$$

where $r$ = reuse fraction.

### 13.4.3 Environmental Resources

Hardware, tools, offices, network, testbeds—affect productivity. E.g., modern IDEs reduce coding effort by 10–20 %.

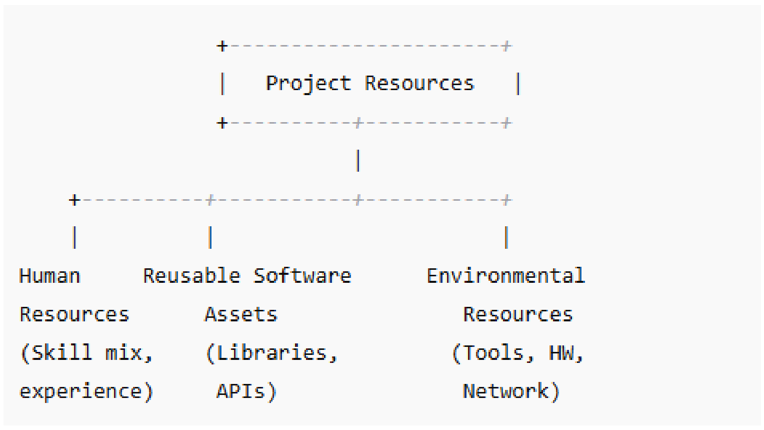

Project Resources
Human Resources (Skill mix, experience)
Reusable Software Assets (Libraries, APIs)
Environmental Resources (Tools, HW, Network)

**Figure 13.2 : Resource categories linked to productivity arrows showing positive/negative influence.**

Illustrates three major resource types and how each affects project performance. Each branch connects to a productivity arrow (↑ positive, ↓ negative). Explain that high skill or better tools increase effective output.

### 13.5 Software Project Estimation

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

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation

Estimation converts size → effort → cost → schedule.

Basic relationship:

$$\text{Effort (person-months)} = a \times (\text{Size})^b$$

where $a$ and $b$ are constants based on organization data.
Cost = Effort × Cost per person-month.
Schedule ≈ c × Effort$^d$ (d ≈ 0.38 – 0.4).

### 13.6 Decomposition Techniques

Decomposition divides a project into smaller parts to estimate more accurately.

#### 13.6.1 Software Sizing

Putnam and Myers [Put92] suggest four different approaches to the sizing problem:

- "Fuzzy logic" sizing. This approach uses approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
- Function point sizing. The planner develops estimates of the information domain characteristics .
- Standard component sizing. Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files,LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.
- Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Determine the "size" to be built:
- Lines of Code (LOC)
- Function Points (FP)
- Use Case Points (UCP)

#### 13.6.2 Problem-Based Estimation

1. Decompose into functions.
2. Estimate LOC for each function.
3. Apply productivity rate (e.g., 500 LOC/person-month).

#### 13.6.3 Example – LOC-Based Estimation

**Given Data**

| **Parameter** | **Value** | **Description** |
|---|---|---|
| Estimated System Size | 25 KLOC (25,000 LOC) | Size of the total code base predicted for the system |
| Productivity Rate | 500 LOC / person-month | Historical productivity from similar projects |
| Average Developer Cost | ₹1 lakh / person-month | Fully loaded cost (salary + overheads) |

**Step 1 – Effort Estimation**

$$\text{Effort (PM)} = \frac{\text{Estimated LOC}}{\text{Productivity (LOC / PM)}}$$

$$= \frac{25{,}000}{500} = 50\text{person-months}$$

**Hence, the project requires ≈ 50 person-months of total development effort.**

**Step 2 – Cost Estimation**

$$\text{Cost} = \text{Effort (PM)} \times \text{Cost per person-month}$$

$$= 50 \times ₹1{,}00{,}000 = ₹50{,}00{,}000(₹50 lakhs)$$

**Step 3 – Schedule Estimation**
**Typical empirical relation between effort and schedule:**

$$T_{dev} = 2.5 \times (\text{Effort})^{0.38}$$

$$T_{dev} = 2.5 \times (50)^{0.38} \approx 8.4\text{months}$$

**Thus, development time ≈ 8 to 9 months.**

**Step 4 – Staffing Level Approximation**
**Average staff = Effort / Schedule**

$$= 50PM/8.4months \approx 6engineers$$

**So a 6-person team over ~8 months can complete the system.**

**Step 5 – Managerial Interpretation**
- Feasibility: Within a year → acceptable for a medium-sized project.
- Sensitivity: If productivity improves to 600 LOC/PM, effort drops to 42 PM → ~₹42 lakhs.
- Risk: Early errors in size estimation (±10 %) change total cost by similar proportion.
- Schedule Check: COCOMO/Putnam models give similar range → estimate validated.

**13.6.4 Example – Function-Point-Based Estimation**
**Given Data**

| **Parameter** | **Value** | **Description** |
|---|---|---|
| Adjusted Function Points | 120 FP | Computed from system functions and complexity factors |
| Productivity | 5 FP / person-month | Based on organizational historical data |
| Cost per person-month | ₹1.2 lakh | Fully loaded development cost |

**Step 1 – Effort Estimation**

$$\text{Effort (PM)} = \frac{\text{Total FP}}{\text{Productivity (FP / PM)}} = \frac{120}{5} = 24PM$$

**Hence, Effort = 24 person-months.**

**Step 2 – Cost Estimation**

$$\text{Cost} = 24PM \times ₹1.2L = ₹28.8lakhs$$

**Thus, total development cost ≈ ₹28.8 lakhs.**

**Step 3 – Schedule Approximation**
**Empirical relation:**

$$T_{dev} = 2.5 \times (Effort)^{0.38}$$

$$T_{dev} = 2.5 \times (24)^{0.38} \approx 6.3\text{months}$$

**Hence, expected development duration ≈ 6 months.**

**Step 4 – Derived Staffing Profile**
**Average team size = 24 PM / 6.3 ≈ 4 developers.**

**Step 5 – Managerial Interpretation**

| **Aspect** | **LOC-Based Estimation** | **Function-Point-Based Estimation** | **Interpretation / Managerial Insight** |
|---|---|---|---|
| Effort | 50 Person-Months | 24 Person-Months | FP method yields a smaller effort estimate because it assumes higher productivity and reuse of existing components. |
| Cost | ₹ 50 Lakhs | ₹ 28.8 Lakhs | FP-based estimation indicates reduced cost, reflecting efficiency gains from reusable modules and mature development processes. |
| Schedule | 8.4 Months | 6.3 Months | FP-based estimation suggests shorter completion time; it is suitable for early project planning when functionality is well-defined. |

FP-based estimation is preferred in early SDLC, when code size is unknown but functional requirements are defined.

**Advantages of FP Estimation**
1. Early applicability (before design or coding).
2. Technology-neutral – works across languages.
3. Supports benchmarking and productivity analysis.
4. Links directly with user requirements.

**Limitations**
- Requires skilled function-point analyst.

- May ignore algorithmic complexity or non-functional aspects.
- Sensitive to subjective complexity ratings.

### 13.6.5 Process-Based Estimation

- Break work by SDLC phases—requirements, design, coding, testing.
- Assign percentage of total effort to each (e.g., 10-20-40-30).
- Useful for resource scheduling.

### 13.6.6 Example – Process-Based Estimation

**Given Data:**

| **Parameter** | **Value** | **Description** |
|---|---|---|
| Total estimated effort | 50 Person-Months | Derived from earlier LOC or FP estimation |
| Effort distribution model | 10% – 20% – 40% – 30% | Standard distribution across SDLC phases (Requirements, Design, Coding, Testing) |
| Management overhead | 10% of total development effort | For project management, reviews, and coordination activities |

**Step 1 – Allocate Effort by Development Phase**

**Divide total estimated effort (50 PM) according to the standard effort distribution:**

| **Development Phase** | **Percentage of Total Effort** | **Effort (Person-Months)** | **Typical Activities** |
|---|---|---|---|
| Requirements Analysis | 10 % | 5 PM | Gathering requirements, stakeholder meetings, use-case modeling |
| Design (System + Detailed) | 20 % | 10 PM | Architecture definition, module and interface design, database schema design |
| Coding / Implementation | 40 % | 20 PM | Programming, unit testing, code reviews, integration of modules |
| Testing (Integration + System) | 30 % | 15 PM | Integration testing, validation, performance and user acceptance testing |
| Subtotal (Development) | 100 % | 50 PM | — |

**Step 2 – Add Management Overhead**

Project management typically adds 10 % of development effort to handle tasks such as:

- Scheduling and progress tracking
- Risk management and quality reviews
- Meetings, reporting, documentation, and audits

$$\text{Management Effort} = 10\% \times 50 = 5 \text{ PM}$$

**Step 3 – Compute Total Effort**

Total Effort = Development Effort + Management Effort

$= 50PM + 5PM = 55PM$

Hence, the total project effort including management and coordination = 55 Person-Months.

**Step 4 – Interpretation and Resource Planning**

| **Aspect** | **Interpretation** |
|---|---|
| Project Duration | If the schedule from earlier estimation is ≈ 8 months, then average team size = 55 / 8 ≈ 7 people. |
| Phase Balance | The coding phase dominates (40 %), consistent with typical software projects. |
| Risk Sensitivity | If requirements are unstable, shift 5–10 % more effort to analysis and design to mitigate late changes. |
| Management Insight | Overhead of 10 % is realistic for mid-size projects; complex or distributed teams may need 15–20 %. |

**Step 5 – Visualization**

**A horizontal bar chart showing proportional effort across phases:**
Requirements [#####-----] 10%
Design [##########----------] 20%
Coding [####################----------------] 40%
Testing [###############-------------] 30%
Management [+] 10% (overhead added on total)

**13.6.7 Estimation with Use Cases**

UCP = (Actor Weight + Use Case Weight) × TCF × ECF.
Example factors: TCF = 1.1, ECF = 0.9.

**Step 1 – Identify and Classify Actors**
Actors **represent external entities interacting with the system (users, devices, other systems).**

| **Actor Type** | **Description** | **Weight** |
|---|---|---|
| **Simple** | **System interacts through a defined API or protocol** | **1** |
| **Average** | **System interacts via text-based interface** | **2** |
| **Complex** | **System interacts via GUI or complex protocol** | **3** |

**Example:**
For a hospital management system (HMS):

- 2 simple actors (insurance server, payment API) → 2 × 1 = 2
- 3 complex actors (doctor, patient, admin) → 3 × 3 = 9
  Total Actor Weight = 2 + 9 = 11

**Step 2 – Identify and Classify Use Cases**
Each use case is assigned a weight depending on the number of transactions (steps) in its main scenario.

| Use Case Type | No. of Transactions | Weight |
|---|---|---|
| **Simple** | **≤ 3** | **5** |
| **Average** | **4–7** | **10** |
| **Complex** | **≥ 8** | **15** |

**Example:**
For the same HMS:
- 3 simple use cases (View Patient Info, View Schedule, Logout) → 3 × 5 = 15
- 2 average use cases (Admit Patient, Generate Bill) → 2 × 10 = 20
- 1 complex use case (Process Payment) → 1 × 15 = 15
  Total Use Case Weight = 15 + 20 + 15 = 50

**Step 3 – Compute Unadjusted Use Case Points (UUCP)**

$$\text{UUCP} = \text{Actor Weight} + \text{Use Case Weight}$$

$$\text{UUCP} = 11 + 50 = 61$$

**Step 4 – Determine Technical Complexity Factor (TCF)**
The TCF accounts for 13 technical parameters (T1–T13), each rated from 0 (irrelevant) to 5 (essential).

| Factor | Description | Example Rating |
|---|---|---|
| T1 | Distributed System | 4 |
| T2 | Performance Objectives | 3 |
| T3 | End-User Efficiency | 4 |
| T4 | Complex Internal Processing | 3 |
| T5 | Reusability | 2 |
| T6 | Easy to Install | 1 |
| … | … | … |

Sum of all factor ratings (ΣFi) = 40 (out of max 65).
Formula:

$$\text{TCF} = 0.6 + (0.01 \times \Sigma Fi)$$

$$\text{TCF} = 0.6 + (0.01 \times 40) = 1.0$$

***Given example:* TCF = 1.1 (for slightly more technical complexity).**

**Step 5 – Determine Environmental Complexity Factor (ECF)**
The ECF accounts for 8 environmental parameters (E1–E8), such as team experience and tool support.

| Factor | Description | Weight | Rating (0–5) | Product |
|---|---|---|---|---|
| E1 | Familiarity with OO concepts | 1.5 | 3 | 4.5 |
| E2 | Application Experience | 0.5 | 4 | 2.0 |
| E3 | Team Motivation | 1.0 | 4 | 4.0 |
| E4 | Stable Requirements | 2.0 | 4 | 8.0 |
| … | … | … | … | … |

**Sum of weighted ratings (ΣEi × Wi) = 22.5**
**Formula:**

$$\text{ECF} = 1.4 + (-0.03 \times \Sigma Ei)$$

$$\text{ECF} = 1.4 - (0.03 \times 22.5) = 0.725 \approx 0.9$$

***Given example:*** **ECF = 0.9 (team is fairly competent).**

**Step 6 – Compute Final Use Case Points**

$$\text{UCP} = (\text{Actor Weight} + \text{Use Case Weight}) \times \text{TCF} \times \text{ECF}$$

$$\text{UCP} = 61 \times 1.1 \times 0.9 = 60.39 \approx 60\ \text{UCP}$$

**Step 7 – Compute Effort, Cost, and Schedule**
**Assume productivity = 20 person-hours per UCP**
**→ Effort = 60 × 20 = 1,200 person-hours**
**Convert to person-months:**

$$\text{Effort} = 1{,}200/160 = 7.5PM$$

**If the cost per PM = ₹1,20,000:**

$$\text{Cost} = 7.5 \times ₹1{,}20{,}000 = ₹9{,}00{,}000$$

**Schedule ≈ 2.5 × (Effort)^0.38**

$$= 2.5 \times (7.5)^{0.38} = 4.8\ \text{months}$$

**Advantages of UCP Estimation**
1. Early Estimation: Applicable right after use-case modeling.
2. Technology Independent: Works for any language or platform.
3. Supports OO Design: Aligns naturally with UML artifacts.
4. Calibratable: Improves accuracy as team collects past project data.

**Limitations**
- Requires consistent rating of actors and use cases.
- Sensitive to subjective assessment of complexity.
- Does not account directly for non-functional requirements like performance or scalability.

**13.6.8 Example – Use-Case-Based Estimation**
Actors = 5 × 2 = 10, Use Cases = 10 × 10 = 100; UCP = 99 ≈ 100.
Productivity = 20 person-hours / UCP → 2000 hours ≈ 12 PM.

**13.6.9 Reconciling Estimates**
Compare all methods.
If LOC → 50 PM, FP → 24 PM, UCP → 12 PM → take weighted average or adjust using risk factors.

```
            +-----------------+
            |  Final Estimate  |
            +-----------------+
                 ^     ^     ^
                 |     |     |
    LOC-based    |  FP-based  |  Use-Case-based
    (50 PM)         (30 PM)        (40 PM)
```

**Figure 13.3 : Triangle connecting LOC, FP, UCP estimates merging to final consensus value.**

## 13.7 Empirical Estimation Models

### 13.7.1 Structure of Estimation Models

Empirical models use data from past projects.
General form:

$$E = a \times (S)^b \times \prod EM_i$$

where $S$ = size, EM = effort multipliers.

### 13.7.2 The COCOMO II Model

Effort = A × (Size)^B × ∏ EM_i

- A ≈ 2.94, B = 0.91 + 0.01ΣSFj.
- EM = product of cost drivers (RELY, DATA, TIME, TEAM, TOOL).

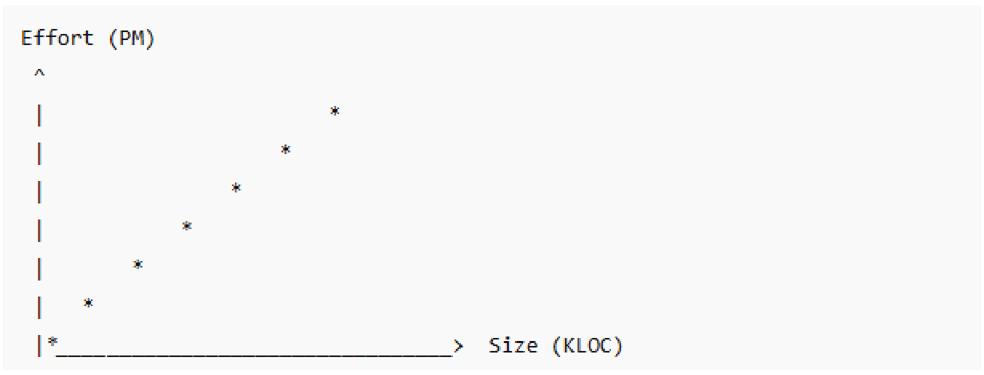
Example: Size = 30 KLOC, B = 1.05, A = 2.94 → Effort ≈ 2.94×(30)^1.05 ≈ 103 PM.
13.7.3 The Software Equation (Putnam Model)

$$E = (Ck)^{1/3} \times S^{4/3}$$

where S = size, C = environment constant, k = productivity.
Used for balancing schedule vs. effort.

```
Effort (PM)
 ^
 |            *
 |          *
 |        *
 |      *
 |   *
 |*_________________________> Development Time (months)
```

**Figure 13.4 : Curved graph showing trade-off between development time and effort—shorter time → exponentially higher effort.**



```
Effort (PM)
 ^
 |                       *
 |                   *
 |               *
 |           *
 |       *
 |   *
 |*_______________________________> Size (KLOC)
```

**Figure 13.5– COCOMO II Nominal Effort Curve**

Display non-linear relation between project size and effort.Doubling size increases effort by > 2× due to exponential term B > 1.

## 13.8 Estimation for Object-Oriented Projects

OO projects require counting classes, methods, and interactions.

**Metrics include:**

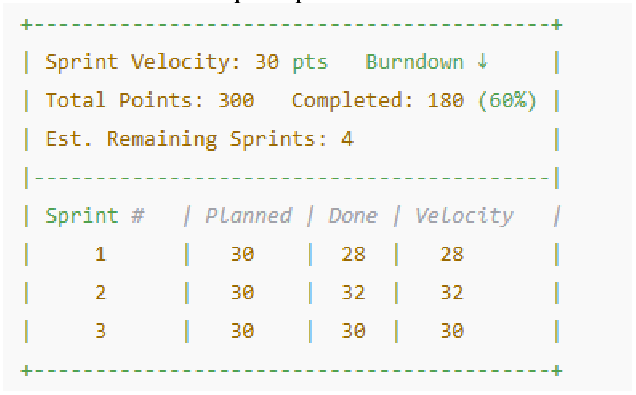
- Number of Classes (NOC)
- Number of Methods (NOM)
- Average Complexity per Method

Effort ≈ (NOC × NOM × Complexity Factor)/Productivity.
Example: 50 classes × 8 methods = 400 methods; Complexity 1.2; Productivity = 10 methods / PM → Effort ≈ 48 PM.

## 13.9 Specialized Estimation Techniques

### 13.9.1 Estimation for Agile Development

Agile uses story points and velocity.
Velocity = average story points completed per iteration.
If project = 300 points and velocity = 30 points/sprint → 10 sprints.
Convert to effort = team size × duration per sprint.



```
+-----------------------------------------+
| Sprint Velocity: 30 pts   Burndown ↓    |
| Total Points: 300   Completed: 180 (60%) |
| Est. Remaining Sprints: 4               |
|-----------------------------------------|
| Sprint #    | Planned | Done | Velocity  |
|     1       |   30    |  28  |   28      |
|     2       |   30    |  32  |   32      |
|     3       |   30    |  30  |   30      |
+-----------------------------------------+
```

**Figure 13.6– Agile Estimation Dashboard Example**

### 13.9.2 Estimation for WebApp Projects

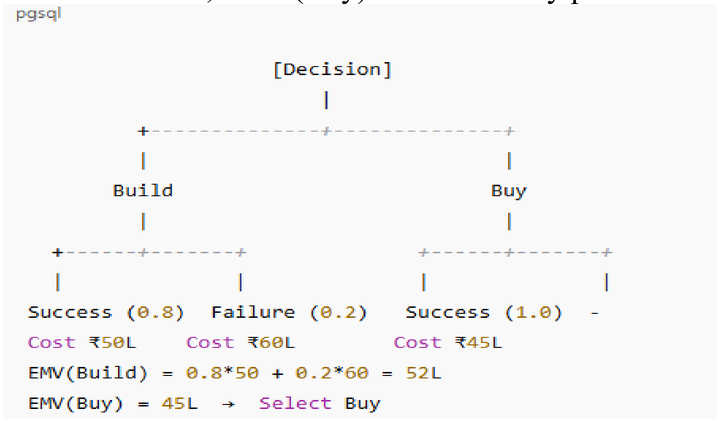
WebApps change rapidly; use page points = static + dynamic pages × complexity. Also consider server load and browser compatibility. Empirical data suggest ~15–25 hours/page for medium complex WebApp.

## 13.10 The Make/Buy Decision

Organizations must decide whether to develop in-house or purchase/outsource.

### 13.10.1 Creating a Decision Tree

List alternatives → assign probabilities and expected costs → choose lowest Expected Monetary Value (EMV).
Example:
Build = ₹50 L, risk of failure 0.2 (cost ₹10 L).
Buy = ₹40 L, customization ₹5 L.
EMV(Build) = 50 + 0.2×10 = 52 L; EMV(Buy) = 45 L → Buy preferred.



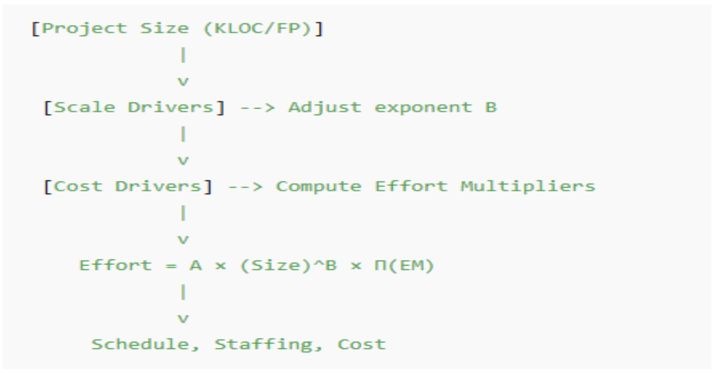
```pgsql
                    [Decision]
                        |
              +---------+---------+
              |                   |
            Build                Buy
              |                   |
        +-----+-----+       +-----+-----+
        |           |       |           |
Success (0.8)  Failure (0.2)  Success (1.0)  -
Cost ₹50L     Cost ₹60L     Cost ₹45L
EMV(Build) = 0.8*50 + 0.2*60 = 52L
EMV(Buy) = 45L  →  Select Buy
```

**Figure 13.7 – Decision Tree for Make/Buy Choice**

### 13.10.2 Outsourcing Considerations

- Vendor capability and reputation.
- Hidden costs of communication and quality control.
- Legal and IP agreements.
- Impact on schedule and security.

•



```
[Project Size (KLOC/FP)]
            |
            v
 [Scale Drivers] --> Adjust exponent B
            |
            v
 [Cost Drivers] --> Compute Effort Multipliers
            |
            v
    Effort = A × (Size)^B × Π(EM)
            |
            v
     Schedule, Staffing, Cost
```

**Figure 13.7 : Decision tree with branches for Build and Buy paths, showing probabilities and expected values.**

## 13.11 Summary

Software project estimation is a foundation for effective planning and control. Accurate estimates require clear scope, identified resources, and multiple techniques to triangulate effort. Decomposition methods (LOC, FP, Use Case) translate requirements into quantitative size. Empirical models (COCOMO II, Software Equation) use historical data for realistic predictions. Special adaptations exist for OO, Agile, and WebApp contexts. Finally, economic decisions like make or buy must consider total cost and risk. Systematic estimation turns uncertainty into manageable risk.

## 13.12 Technical Terms

1. Software Estimation
2. Function Point (FP)
3. Use Case Point (UCP)
4. COCOMO II Model
5. Software Equation
6. Productivity Rate
7. Decomposition Technique
8. Effort Multipliers
9. Make/Buy Decision
10. Story Point (Agile Metric)

## 13.13 Self-Assessment Questions

### Essay-Type

1. Explain the need for accurate software estimation and its challenges.
2. Describe the main steps in the project planning process.
3. Discuss three decomposition-based estimation techniques with examples.
4. Explain the COCOMO II model and its major parameters.
5. Differentiate between LOC-, FP-, and Use Case-based estimation.
6. What is process-based estimation and how is it applied?
7. How do object-oriented features influence estimation?
8. Outline the Agile approach to effort estimation using story points.
9. Describe the make/buy decision process with a numerical example.
10. List the key factors that affect estimation accuracy.

### Short -Type

1. Software Sizing
2. Feasibility Study
3. Effort and Cost Formulas
4. Use Case Points
5. COCOMO Constants
6. Software Equation
7. Agile Velocity
8. WebApp Page Point
9. Decision Tree
10. Outsourcing Risks

## 13.14 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Boehm, Barry W., *Software Engineering Economics*, Prentice Hall, 1981.
3. Sommerville, Ian, *Software Engineering*, 10th Ed., Pearson Education, 2015.
4. Putnam, Lawrence H., & Myers, Ware, *Measures for Excellence*, Yourdon Press, 1992.
5. Jalote, Pankaj, *An Integrated Approach to Software Engineering*, Narosa Publishing, 2005.
6. McConnell, Steve, *Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006.
7. Boehm, Barry W., et al., *COCOMO II Model Definition Manual*, USC-CSE, 2000.
8. Schwalbe, Kathy, *Information Technology Project Management*, Cengage Learning, 2018.
9. Fenton, Norman & Pfleeger, Shari, *Software Metrics: A Rigorous and Practical Approach*, CRC Press, 2014.
10. IEEE Std 1058, *Software Project Management Plans (SPMP)*, IEEE, 1998.

Dr. U. Surya Kameswari

LESSON- 14

# QUALITY MANAGEMENT

## AIMS AND OBJECTIVES

To understand the concept of software quality, its dimensions, measurement, and management practices essential for delivering reliable, maintainable, and user-satisfactory software products.

**After completing this lesson, you will be able to:**

1. Define software quality and explain its importance in the software engineering process.
2. Describe Garvin's quality dimensions, McCall's quality factors, and ISO 9126/25010 standards.
3. Identify the challenges involved in achieving and maintaining software quality.
4. Discuss the Software Quality Dilemma and the "Good Enough Software" concept.
5. Analyze the Cost of Quality (CoQ) and its implications for project management.
6. Recognize the roles of quality assurance, quality control, and process improvement.
7. Apply management and engineering techniques to achieve and sustain software quality

## STRUCTURE

## 14.1 What Is Quality?

"Quality" is the degree to which a product meets the needs and expectations of its users. For software, this translates to the conformance of software to functional and performance requirements, documented standards, and implicit customer expectations.
Two Views of Quality

1. User View (Fitness for Purpose):
   Software should perform tasks that satisfy the user's needs effectively.

2. Developer View (Conformance to Specification):
   Software should meet design, coding, and documentation standards.

Key Attributes of Software Quality

- Correctness
- Reliability
- Efficiency
- Integrity
- Usability
- Maintainability
- Testability
- Portability
- Reusability

## 14.2 Software Quality

Software quality is not absolute—it depends on context, users, and goals.
To make it measurable, various quality models have been proposed that define dimensions or factors.

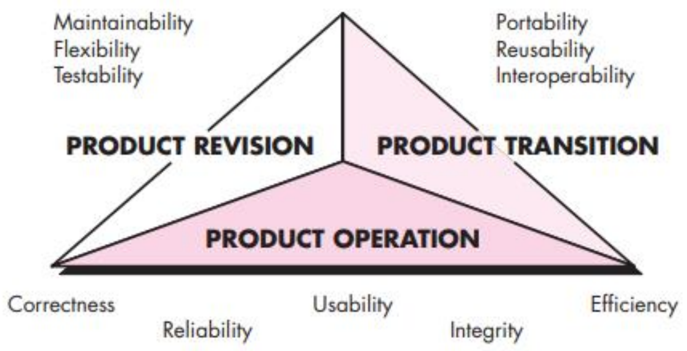


**FIGURE 14.1 McCall's software quality factors**

**Correctness.** The extent to which a program satisfies its specification and fulfills the customer's mission objectives.

**Reliability.** The extent to which a program can be expected to perform its intended function with required precision. [It should be noted that other, more complete definitions of reliability have been proposed.

**Efficiency.** The amount of computing resources and code required by a program to perform its function.

**Integrity.** Extent to which access to software or data by unauthorized persons can be controlled.

**Usability.** Effort required to learn, operate, prepare input for, and interpret output of a program.

**Maintainability.** Effort required to locate and fix an error in a program. [This is a very

**Flexibility.** Effort required to modify an operational program.

**Testability.** Effort required to test a program to ensure that it performs its intended function.

**Portability.** Effort required to transfer the program from one hardware and/or software system environment to another.

**Reusability.** Extent to which a program [or parts of a program] can be reused in other applications—related to the packaging and scope of the functions that the program performs.

**Interoperability.** Effort required to couple one system to another
It is difficult, and in some cases impossible, to develop direct measures2 of these quality factors. In fact, many of the metrics defined by McCall et al. can be measured only indirectly. However, assessing the quality of an application using these factors will provide you with a solid indication of software quality.

### 14.2.1 Garvin's Quality Dimensions

David Garvin (1984) proposed eight universal dimensions of quality applicable to all products, including software:

| **Dimension** | **Meaning in Software Context** |
|---|---|
| Performance | Execution speed, resource utilization |
| Features | Additional capabilities that enhance usefulness |
| Reliability | Frequency and severity of failures |
| Conformance | Adherence to standards and specifications |
| Durability | Long-term stability and error-free operation |
| Serviceability | Ease of maintenance and upgrade |
| Aesthetics | Interface appeal and user experience |
| Perceived Quality | Market reputation and user confidence |

**Example:**
In a banking system, *reliability* (error-free transactions) and *security* are dominant quality dimensions.

### 14.2.2 McCall's Quality Factors

McCall et al. (1977) introduced a hierarchical model linking product quality factors to measurable criteria.
Three Major Categories:

1. **Product Operation – Characteristics related to software execution.**
   - Correctness
   - Reliability
   - Efficiency
   - Integrity
   - Usability

2. **Product Revision – Ease of adapting software to changes.**
   - Maintainability
   - Flexibility
   - Testability

3. **Product Transition – Portability and interoperability.**
   - Portability
   - Reusability
   - Interoperability

McCall's model bridges engineering attributes (like code structure) with user expectations.

**1. Product Operation – Characteristics Related to Software Execution**

This category focuses on **the software's runtime behavior** and its ability to perform required functions efficiently, correctly, and reliably.

| Factor | Description | Example |
|---|---|---|
| **Correctness** | The extent to which the software performs its intended functions accurately and completely. | A billing system generating precise invoices without miscalculations. |
| **Reliability** | The probability of failure-free operation over a specified time and environment. | An ATM system functioning continuously for 99.9% uptime. |
| **Efficiency** | Optimal use of system resources such as CPU, memory, and I/O. | A data compression tool executing quickly with minimal resource load. |
| **Integrity** | Protection of software and data from unauthorized access or corruption. | Encryption and role-based access control in a hospital management system. |
| **Usability** | Ease with which users can learn, operate, and interact with the system. | A mobile banking app with intuitive design and help tips. |

**Interpretation:**
The **Product Operation factors** directly affect **user satisfaction and system performance.** Developers must balance these aspects—for instance, optimizing efficiency should not compromise usability.

**2. Product Revision – Ease of Adapting Software to Changes**
Software evolves due to changing requirements, technologies, or regulations.

**Product Revision** factors determine how easily a system can be **maintained, modified, or tested** after its initial release.

| Factor | Description | Example |
|---|---|---|
| **Maintainability** | The ease with which errors can be corrected and improvements made. | Modular code and meaningful comments simplify debugging. |
| **Flexibility** | The ability to adapt to new environments or changing requirements. | A report generation module that can easily switch between PDF and Excel output. |
| **Testability** | The degree to which software facilitates validation and verification. | Availability of unit testing hooks and diagnostic logging. |

**Interpretation:**
High-quality software should not only perform well but also be **easy to evolve**. These factors are critical for **reducing maintenance cost**—which can consume over 60% of total software life-cycle effort.

**3. Product Transition – Portability and Interoperability**
In an era of heterogeneous systems and cloud computing, software must **interact across platforms and integrate seamlessly** with other systems. **Product Transition** focuses on these adaptability aspects.

| Factor | Description | Example |
|---|---|---|
| **Portability** | The ease with which software can be transferred from one environment to another. | A Java-based system running on both Windows and Linux. |
| **Reusability** | The degree to which software modules can be reused in other applications. | A shared authentication service reused across multiple apps. |
| **Interoperability** | The ability of software to interact with other systems effectively. | An ERP system exchanging data with a CRM through REST APIs. |

**Interpretation:**
Product Transition factors emphasize **scalability and integration**, making software investments sustainable as technology evolves.

**Integration of Categories**
McCall's model emphasizes that these three categories are **interconnected**:

- **Operation factors** define immediate product quality as seen by users.
- **Revision factors** ensure long-term maintainability and sustainability.
- **Transition factors** enable the product to survive technological and market evolution.

Collectively, they ensure that software is not only functional but also **usable, modifiable, and future-proof.**

**Advantages of McCall's Model**

1. Provides a **structured framework** for defining and measuring quality.
2. Covers the **entire software life cycle**, from development to deployment.

3. Serves as a **foundation for later standards**, including **ISO 9126** and **CMMI**.
4. Bridges **technical metrics** and **customer satisfaction**.

**Limitations**

- Some factors overlap (e.g., maintainability and flexibility).
- Does not directly address modern aspects like **security**, **usability metrics**, or **UX design**.
- Lacks explicit quantification methods — qualitative assessments can vary among evaluators.

**Managerial Application**

Project managers can use McCall's factors to:

- Develop **quality checklists** for reviews and audits.
- Identify **trade-offs** (e.g., performance vs maintainability).
- Define **quality metrics**—e.g., reliability via MTBF, maintainability via defect-fix effort.
- Align engineering practices with **customer satisfaction goals**.
- **Example – Web Application:**
- A university portal aims for:
- **Product Operation:** Reliability of student result retrieval.
- **Product Revision:** Easy updates to curriculum modules.
- **Product Transition:** Portability to mobile and cloud platforms.

Applying McCall's model helps structure and prioritize these objectives.

**Summary of McCall's Quality Model**

| **Category** | **Focus** | **Typical Factors** | **Key Outcome** |
|---|---|---|---|
| Product Operation | User experience and runtime performance | Correctness, Reliability, Efficiency, Integrity, Usability | User satisfaction |
| Product Revision | Ease of modification | Maintainability, Flexibility, Testability | Long-term adaptability |
| Product Transition | Future adaptability and compatibility | Portability, Reusability, Interoperability | Longevity and scalability |

McCall's Quality Model remains a **cornerstone of software quality management**. Its enduring relevance lies in its simplicity and holistic view: balancing **operational excellence**, **maintainability**, and **adaptability**. While modern standards like ISO 25010 extend it, McCall's framework continues to guide both academic and industrial perspectives on **what makes software truly "high quality."**

### 14.2.3 ISO 9126 Quality Factors

The ISO/IEC 9126 standard (now evolved into ISO 25010) defines six major quality characteristics, each subdivided into measurable sub-characteristics.

**Purpose of the ISO 9126 Model**

- To establish a **common language** for discussing software quality.
- To define **key quality characteristics** and their measurable **sub-characteristics**.
- To help organizations **plan, evaluate, and improve** the quality of software throughout its life cycle.

ISO 9126 classifies software quality into **six main characteristics**, each broken down into several **sub-characteristics** that can be quantitatively or qualitatively evaluated.

| Main Characteristic | Sub-characteristics |
|---|---|
| Functionality | Suitability, Accuracy, Security, Compliance |
| Reliability | Maturity, Fault Tolerance, Recoverability |
| Usability | Understandability, Learnability, Operability |
| Efficiency | Time Behavior, Resource Utilization |
| Maintainability | Analyzability, Changeability, Stability, Testability |
| Portability | Adaptability, Installability, Co-existence, Replaceability |

**1. Functionality**

This characteristic addresses what the software does—its ability to provide functions that meet stated and implied needs when used under specified conditions.

| Sub-characteristic | Description | Example |
|---|---|---|
| Suitability | Appropriateness of functions for specified tasks. | An accounting system correctly supports all ledger operations. |
| Accuracy | Correctness of results or outputs produced. | A medical application calculates dosage precisely. |
| Security | Protection against unauthorized access or data corruption. | Multi-factor authentication and encryption in a banking app. |
| Compliance | Adherence to industry, legal, and safety standards. | GDPR-compliant data handling in European web applications. |

**Interpretation:**
Functionality defines whether the software meets its intended purpose and protects both the system and user data.

**2. Reliability**
Reliability refers to the capability of software to maintain performance levels under specific conditions for a specified period. It directly influences user confidence and system dependability.

| Sub-characteristic | Description | Example |
|---|---|---|
| Maturity | Frequency of software failure under normal use. | A POS system running continuously without crashing. |
| Fault Tolerance | Capability to maintain performance despite faults. | Redundant server clusters in an e-commerce website. |
| Recoverability | Ability to restore service after failure. | Automatic database restore after power outage. |

**Interpretation:**
High reliability reduces downtime, minimizes support costs, and improves customer trust—critical for mission-critical and embedded systems.

**3. Usability**
Usability concerns the ease of learning, operating, and understanding the software. It emphasizes the human–computer interaction (HCI) aspect of quality.

| Sub-characteristic | Description | Example |
|---|---|---|
| Understandability | Clarity of system concepts to users and maintainers. | Logical menu structures in an ERP system. |
| Learnability | Ease with which users can learn to operate the system. | A mobile app with intuitive onboarding and tutorials. |
| Operability | User comfort and control during operation. | Undo/redo functionality in document editors. |

Interpretation:
Usability has a direct impact on user satisfaction and productivity. Even a powerful system fails if users find it difficult to operate.

**4. Efficiency**
Efficiency represents the relationship between the software's performance and the resources it consumes (CPU, memory, bandwidth, etc.). It ensures that the software delivers required performance under resource constraints.

| Sub-characteristic | Description | Example |
|---|---|---|
| Time Behavior | Response times and processing speeds during execution. | Web server responding within 2 seconds for 95% of requests. |
| Resource Utilization | Efficient use of hardware and software resources. | A data analytics tool optimizing memory usage during processing. |

Efficiency affects scalability, energy use, and cost — especially vital for real-time systems, mobile devices, and cloud applications.

**5. Interpretation: Maintainability**
Maintainability refers to the ease with which a software product can be analyzed, modified, and tested to correct defects, improve performance, or adapt to a changed environment.

| Sub-characteristic | Description | Example |
|---|---|---|
| Analyzability | Ease of diagnosing faults or failures. | Log and trace features simplifying debugging. |
| Changeability | Ease of making modifications. | Modular code design enabling simple updates. |
| Stability | Risk of introducing new defects when making changes. | Version-controlled and regression-tested software releases. |
| Testability | Ease of validating modified software. | Automated test suites in continuous integration environments. |

**Interpretation:**
A maintainable system reduces long-term costs and supports continuous improvement—essential in Agile and DevOps settings.

**6. Portability**
Portability measures the ease with which software can be transferred from one environment to another, including hardware, operating systems, or networks.

| **Sub-characteristic** | **Description** | **Example** |
|---|---|---|
| Adaptability | Ability to be adapted to different environments. | A website functioning on various browsers and screen sizes. |
| Installability | Ease of installing and configuring the software. | One-click installation for desktop applications. |
| Co-existence | Ability to operate efficiently with other applications. | Shared resource usage between ERP and HR systems. |
| Replaceability | Ease of replacing one system with another equivalent. | Switching between email clients without data loss. |

**Interpretation:**
Portability ensures product longevity and customer flexibility, allowing migration to modern technologies without redesigning core systems.

**Advantages of ISO 9126 Model**

1. Provides standardized terminology for discussing software quality.
2. Supports quantitative evaluation of each characteristic.
3. Applicable throughout the software life cycle (design, testing, maintenance).
4. Serves as a benchmark for software certification and comparison.

**Limitations**
- Measurement of sub-characteristics may require subjective judgment.
- Inter-dependencies between factors can complicate assessment.
- Does not specify exact metric thresholds — organizations must define them.

**Practical Example: Online Banking System**

| **Quality Characteristic** | **Critical Requirement** |
|---|---|
| Functionality | Accurate and secure transaction processing |
| Reliability | Continuous 24×7 uptime with backup recovery |
| Usability | Intuitive user interface for all age groups |
| Efficiency | Instant fund transfer with minimal latency |
| Maintainability | Easy integration of new services (e.g., UPI, card linking) |
| Portability | Works on web, Android, and iOS platforms |

**14.2.4 Targeted Quality Factors**
Certain specialized applications demand **domain-specific quality goals**, such as:
- **Real-time systems:** Timing accuracy, predictability
- **Web applications:** Response time, scalability
- **Embedded systems:** Memory efficiency, reliability
- **Safety-critical systems:** Fault tolerance, certification compliance

Thus, “quality” must be **tailored** to the product’s operational domain.

**14.2.5 The Transition to a Quantitative View**
Initially, software quality was qualitative (“good” or “bad”).
Modern software engineering adopts **quantitative quality metrics**, e.g.:
- Defect density = defects / KLOC
- Mean Time Between Failures (MTBF)
- Customer Satisfaction Index (%)
- Defect Removal Efficiency (DRE)

These allow **objective tracking and continuous improvement**.

### 14.3 The Software Quality Dilemma

Every software team faces the trade-off between **quality**, **cost**, and **schedule**.

#### 14.3.1 "Good Enough" Software

Modern software markets often prioritize rapid delivery.
"Good enough" software meets the **minimum acceptable quality** to satisfy users and gain market entry — not necessarily defect-free.
**Pros:** Faster time-to-market.
**Cons:** Increased maintenance cost, brand damage, and user frustration.
**Managerial View:**
"Good enough" should never mean "unsafe" — it's a controlled compromise, not negligence.

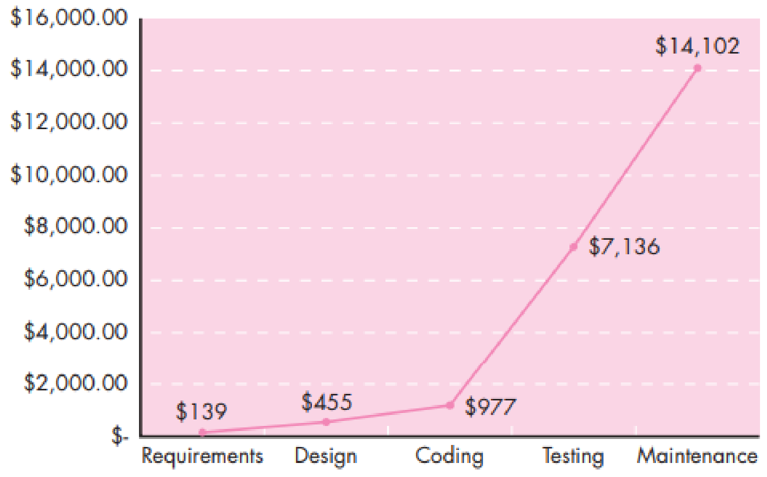


**FIGURE 14.2 Relative cost of correcting errors and defects Source: Adapted from**

According to industry average data, the cost of finding and correcting defects during the coding phase is $977 per defect. Thus, the total cost for correcting the 200 "critical" defects during this phase (200 $977) is approximately $195,400. Industry average data shows that the cost of finding and correcting defects during the system testing phase is $7,136 per defect. In this case, assuming that the system testing phase revealed approximately 50 critical defects (or only 25% of those found by Cigital in the coding phase), the cost of finding and fixing those defects (50 $7,136) would have been approximately $356,800. This would also have resulted in 150 critical errors going undetected and uncorrected. The cost of finding and fixing these remaining 150 defects in the maintenance phase (150 $14,102) would have been $2,115,300. Thus, the total cost of finding and fixing the 200 defects after the coding phase would have been $2,472,100 ($2,115,300 $356,800).

Even if your software organization has costs that are half of the industry average (most have no idea what their costs are!), the cost savings associated with early quality control and assurance activities (conducted during requirements analysis and design) are compelling.

#### 14.3.2 The Cost of Quality (CoQ)

The **total cost of quality** includes both achieving and failing to achieve quality.

| **Cost Category** | **Description** | **Examples** |
|---|---|---|
| **Prevention Costs** | Efforts to prevent defects | Training, standards, reviews |

| | | |
|---|---|---|
| **Appraisal Costs** | Efforts to detect defects | Testing, inspections |
| **Internal Failure Costs** | Defects found before delivery | Rework, debugging |
| **External Failure Costs** | Defects after delivery | Customer complaints, recalls |

**Key Principle:**

*Investing in prevention and appraisal early saves cost later.*

### 14.3.3 Risks
Poor quality introduces multiple risks:
- Financial losses due to system downtime.
- Reputational damage.
- Legal liability (especially in safety-critical domains).
- Security breaches due to low integrity.

### 14.3.4 Negligence and Liability
When software defects lead to damage or injury, developers may be legally liable.
**Example:** Defective medical device software causing incorrect dosage.
**Ethical Responsibility:** Quality is not optional; it's a moral and legal obligation.

### 14.3.5 Quality and Security
Security is now recognized as a **core quality attribute**.
Quality and security reinforce each other:
- Secure code prevents unauthorized access.
- Reliable error handling reduces vulnerabilities.
- Proper validation prevents injection attacks.

### 14.3.6 The Impact of Management Actions
Leadership determines quality outcomes through:
- Clear quality goals and policies.
- Allocation of adequate resources.
- Training and motivating teams.
- Monitoring metrics and audits.
- Quality culture starts at the top.

## 14.4 Achieving Software Quality
Software quality doesn't happen by chance — it's engineered through **methods, management, and measurement**.

### 14.4.1 Software Engineering Methods
- Use of systematic, repeatable engineering processes.
- Modular and structured design for maintainability.
- Code standards, design reviews, version control.
- Early defect detection using static analysis tools.

### 14.4.2 Project Management Techniques
- Clear quality planning in the **Software Project Plan (SPMP)**.
- Risk-based testing and milestone reviews.
- Metrics-driven decision-making (DRE, defect density).

- Periodic audits and feedback loops.

### 14.4.3 Quality Control

Quality Control (QC) focuses on **detecting defects** in the product.

**Techniques:**

- Peer reviews and walkthroughs.
- Unit, integration, and system testing.
- Regression testing after updates.
- Test automation tools.

QC is reactive — it ensures the product conforms to specifications.

### 14.4.4 Quality Assurance

Quality Assurance (QA) ensures the **processes used** to create software are followed and effective.

**QA Activities:**

- Process definition and documentation.
- Process audits and compliance checks.
- Continuous process improvement (CMMI, ISO 9001).
- Root cause analysis and corrective action plans.

QA is proactive — it builds quality into the process.

**QA vs QC**

| **Aspect** | **Quality Assurance (QA)** | **Quality Control (QC)** |
|---|---|---|
| Focus | Process-oriented | Product-oriented |
| Goal | Prevent defects | Detect defects |
| Timing | Throughout lifecycle | Post-development |
| Responsibility | Process and QA teams | Testing teams |
| Approach | Proactive | Reactive |

## 14.5 Summary

Software quality is a multidimensional concept involving **technical, managerial, and user-centered perspectives**. Models such as **Garvin's**, **McCall's**, and **ISO 9126** help structure quality goals. Organizations face a constant **quality–cost–schedule** dilemma, but investing in quality early reduces total cost. Through disciplined engineering, management practices, and assurance activities, sustainable quality can be achieved. Ultimately, software quality is a **team culture**, not just a metric.

## 14.6 Technical Terms

1. Software Quality
2. Garvin's Quality Dimensions
3. McCall's Quality Factors
4. ISO 9126
5. Targeted Quality Factors
6. Cost of Quality (CoQ)
7. Quality Assurance (QA)
8. Quality Control (QC)
9. Reliability
10. Maintainability

## 14.7 Self-Assessment Questions

### Essay Questions

1. Define software quality. Explain Garvin's and McCall's quality models.
2. Compare ISO 9126 and McCall's quality frameworks.
3. Explain the "Software Quality Dilemma" and the "Good Enough" concept.
4. What are the components of the Cost of Quality (CoQ)?
5. Discuss management's role in achieving software quality.
6. Differentiate between QA and QC with examples.
7. Explain how quality metrics help in process improvement.
8. Describe how security relates to software quality.
9. What are targeted quality factors? Provide examples for web and real-time systems.
10. How can a software organization balance time, cost, and quality?

### Short Notes

1. Garvin's Eight Dimensions
2. Product Operation vs Product Transition
3. Quality Metrics
4. ISO 25010
5. Good Enough Software
6. Prevention vs Appraisal Cost
7. Negligence in Software Failures
8. QA/QC Comparison
9. Role of Management in Quality
10. Continuous Improvement

## 14.8 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Sommerville, Ian, *Software Engineering*, 10th Ed., Pearson Education, 2015.
3. ISO/IEC 25010:2011, *Systems and Software Engineering – Systems and Software Quality Models.*
4. McCall, J. A., *Factors in Software Quality*, U.S. Rome Air Development Center Report, 1977.
5. Garvin, David A., *Managing Quality: The Strategic and Competitive Edge*, Free Press, 1988.
6. Fenton, Norman & Pfleeger, Shari, *Software Metrics: A Rigorous and Practical Approach*, CRC Press, 2014.
7. Humphrey, Watts S., *Managing the Software Process*, Addison-Wesley, 1989.
8. Kan, Stephen H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2003.
9. Crosby, Philip B., *Quality is Free*, McGraw-Hill, 1979.
10. IEEE Std 730, *Software Quality Assurance Plans*, IEEE, 2018.

Dr. U. Surya Kameswari

LESSON- 15

# FORMAL METHODS

**AIMS AND OBJECTIVES**

To understand the principles, purpose, and practical application of formal methods in software engineering—mathematically based approaches used to specify, design, and verify software systems.

**After completing this lesson, you will be able to:**

1. Define formal methods and explain their role in software specification and verification.
2. Describe the basic concepts and advantages of using formal methods.
3. Understand the syntax and role of the Object Constraint Language (OCL) in modeling systems.
4. Explain the Z specification language and its use in defining system behavior mathematically.
5. Discuss The Ten Commandments for Formal Methods — practical guidelines for using them effectively.
6. Recognize the limitations and challenges of adopting formal methods in real-world projects.

**STRUCTURE**

## 15.1 Introduction to Formal Methods

**Formal Methods** are **mathematically rigorous techniques** used in software engineering to specify, develop, and verify software systems.
They involve the use of **formal specification languages**, based on logic and set theory, to ensure that software behaves exactly as intended.

"If a system is critical enough to demand correctness, it demands formality."
— Roger Pressman

**Why Formal Methods?**

Traditional software specifications (written in natural language) are often **ambiguous, inconsistent, or incomplete**.
Formal methods aim to **remove ambiguity** and provide **proofs of correctness** through mathematical reasoning.

**Applications**

- Safety-critical systems (aviation, medical, nuclear control).
- Financial systems (banking, trading, auditing).
- Communication protocols and embedded systems.

**1. Safety-Critical Systems**

Safety-critical systems are those where software failures can have **catastrophic consequences**, such as injury, death, or large-scale operational failure.
Formal methods help engineers **prove that critical safety requirements are always satisfied** under all operating conditions.

**Common Application Domains**

- **Aviation:**
  Flight control systems, autopilot modules, and air traffic management rely heavily on formally verified software to ensure stable flight, navigation, and collision avoidance.
  *Example:* The **Airbus A380 flight control software** uses the **B-Method** and **Z specifications** to formally verify control laws.
- **Medical Devices:**
  Software in infusion pumps, pacemakers, and diagnostic systems must comply with strict safety standards (IEC 62304).
  *Example:* Formal verification is applied to ensure that a pacemaker delivers correct pulse timing under all heart rate conditions.
- **Nuclear and Industrial Control Systems:**
  Reactor monitoring, emergency shutdown systems, and automated control software require **proof of correctness** and **predictable timing**.
  Formal models help confirm that no unsafe states are reachable during operation.

**Benefits**

- Elimination of design ambiguities before implementation.
- Verification of safety properties (e.g., "system never enters unsafe state").
- Support for certification standards such as **DO-178C** (aviation) or **IEC 61508** (industrial control).

**2. Financial and Banking Systems**

In financial systems, errors can cause **huge monetary losses**, data inconsistency, or security breaches.

Formal methods provide a **mathematical guarantee of integrity, accuracy, and compliance** in financial software.

**Typical Applications**

- **Banking Software:**
  Formal models ensure that transactions such as deposits, withdrawals, and fund transfers are consistent and atomic (no partial operations).
  *Example:* Formal specification of transaction logic prevents overdraft and ensures correct balance reconciliation.
- **Stock Trading Platforms:**
  In high-frequency trading, timing precision and concurrency are critical. Formal verification ensures that trade-matching algorithms behave deterministically under heavy load.
- **Auditing and Compliance Tools:**
  Regulatory frameworks (like Basel III, SEPA, or GDPR) require traceable and correct software logic.
  Formal verification validates rule-based compliance automation.

**Benefits**

- Assurance of transaction accuracy and data consistency.
- Prevention of financial fraud due to software logic errors.
- Compliance with legal and regulatory standards.

**Example Case**

In **London Stock Exchange's trading system (TradElect)**, formal specification helped detect timing and concurrency errors during simulation—preventing multimillion-pound losses.

**3. Communication Protocols and Embedded Systems**

Communication systems rely on protocols that define **structured message exchange**, **synchronization**, and **error handling**.

Formal methods are particularly suited to **protocol verification**, where timing, concurrency, and state transitions must be rigorously analyzed.

**Communication Protocols**

- Used to specify and verify **data transmission rules**, ensuring no deadlocks or data loss occur.
  *Example:* Formal verification of the **TCP/IP protocol stack** and **Bluetooth communication protocol** helped detect synchronization bugs during development.
- Formal models such as **finite state machines (FSMs)** and **temporal logic** describe how messages are exchanged and how systems respond to failures or delays.

**Embedded Systems**

- Embedded systems operate in **resource-constrained and real-time environments**, where timing correctness is critical.
  *Example:* Automotive systems (e.g., **braking control, airbag deployment**) use formal timing models to guarantee response within milliseconds.

**Benefits**

- Verification of timing, synchronization, and fault recovery.
- Assurance that concurrent processes never reach inconsistent states.
- Increased reliability of IoT, sensor networks, and autonomous systems.

**4. Additional Emerging Domains**

| Domain | Use of Formal Methods | Example |
|---|---|---|
| **Cybersecurity** | Proving correctness of cryptographic algorithms and protocols. | Formal proof of SSL/TLS handshake integrity. |
| **Blockchain and Smart Contracts** | Ensuring correctness of immutable transaction logic. | Ethereum smart contracts verified using Coq or Isabelle. |
| **Autonomous Vehicles** | Validating control algorithms for path planning and obstacle avoidance. | Safety verification of self-driving car decision systems. |
| **AI Systems** | Ensuring transparency and predictability in ML models. | Formal verification of AI-based decision boundaries in healthcare diagnostics. |

**Summary of Applications**

| Domain | Focus of Formal Methods | Key Benefits |
|---|---|---|
| **Safety-Critical Systems** | Safety, fault tolerance, timing correctness | Prevents catastrophic failure |
| **Financial Systems** | Accuracy, consistency, compliance | Prevents financial loss and fraud |
| **Communication Systems** | Synchronization, concurrency, state transitions | Ensures reliable data exchange |
| **Embedded Systems** | Real-time response, hardware–software integration | Guarantees predictable behavior |
| **Cybersecurity / AI / Blockchain** | Logic correctness, model transparency | Builds trust and compliance |

Formal methods have become a **strategic necessity** in domains where **failure is unacceptable**.
While their adoption is often limited by complexity and cost, the increasing need for **reliable, safe, and provable software** is driving wider use in aerospace, healthcare, finance, and autonomous systems.
As software continues to control more aspects of human life, formal methods provide the **foundation of trust and assurance** in system behavior.

## 15.2 Basic Concepts of Formal Methods

Formal methods involve three key activities:

| Activity | Description | Example |
|---|---|---|
| **Specification** | Defining system behavior mathematically using logic and set theory. | Describing a banking system's account balance rules formally. |
| **Development** | Refining the specification into executable code while preserving correctness. | Transforming Z specification into Ada or Java code. |
| **Verification** | Proving that implementation meets the formal specification. | Using theorem provers or model checkers to verify consistency. |

**Core Concepts**

1. **Formal Specification Language:**
   A mathematically based language used to express software requirements precisely. Examples: **Z**, **VDM**, **B-Method**.
2. **Model-Based vs Property-Based Approaches:**
   - **Model-Based:** Describes system state and operations (e.g., Z, VDM).
   - **Property-Based:** Focuses on system properties expressed in logic (e.g., temporal logic).
3. **Verification Techniques:**
   - **Proof of correctness** using mathematical logic.
   - **Model checking** (automatic verification against specifications).
   - **Consistency checking** (ensuring no contradictions).
4. **Formal Semantics:**
   Defines **meaning** of system constructs using mathematical rules, ensuring the model's behavior matches reality.

**Benefits**

- Removes ambiguity from specifications.
- Enables early detection of design errors.
- Provides proof of correctness.
- Improves documentation quality.
- Reduces maintenance effort in long-lifecycle systems.

**Challenges**

- Requires highly trained engineers.
- Time-consuming and costly for large systems.
- Often seen as difficult to integrate with agile methods.

## 15.3 Object Constraint Language (OCL)

**Overview**

The **Object Constraint Language (OCL)** is a formal language used in **UML (Unified Modeling Language)** to specify **invariants**, **preconditions**, and **postconditions** on UML models.

It ensures that models are **precise and unambiguous**, extending UML diagrams with mathematical rigor.

**Key Features**

- Based on **first-order logic**.
- **Declarative**: describes what must hold true, not how to implement it.
- **Side-effect free**: does not change system state.
- Works alongside **UML class diagrams, use-case diagrams, and state machines**.

**1. Based on First-Order Logic**

OCL is grounded in first-order predicate logic, allowing the expression of constraints and rules that can be formally evaluated and verified.
This provides a mathematically precise semantics for UML models.
**Example:**
To specify that an account balance can never go negative:
context Account inv: balance >= 0
This invariant uses logical quantifiers and relational operators (>=, =, and, or) from first-order logic to ensure formal validity.
Interpretation:
Unlike informal notes or comments, OCL constraints can be parsed, verified, and automatically checked using UML tools.

**2. Declarative: Describes What Must Hold True**

OCL is declarative, meaning it states conditions and truths about a system, not the steps to enforce them.
It focuses on the intent, not the procedure.
**Example:**
To ensure that a withdrawal operation only happens when funds are sufficient:
context Account::withdraw(amount: Real)
pre: amount > 0 and amount <= balance
This constraint declares what must be true before the operation executes but does not specify how to code the withdrawal logic.
Significance:
Declarative rules make OCL independent of programming language and implementation details, ensuring that system design remains consistent across platforms.

**3. Side-Effect Free: Does Not Change System State**

OCL expressions are pure — they do not alter system data or behavior.
They only evaluate conditions and return logical results (true/false) or computed values.
This ensures safe, predictable analysis of models.
**Example:**
context Order
inv: totalPrice = items->collect(subtotal)->sum()

Here, the constraint calculates the total price from order items, but it does not modify any data values in the system.
Thus, the model remains stable during validation.
**Importance:**
Side-effect-free evaluation makes OCL ideal for automated validation, model checking, and test generation, as it avoids unintended data changes.

**4. Works Alongside UML Models**
OCL is designed to extend and complement UML — it does not replace graphical modeling but enhances it by adding semantic precision.
It integrates seamlessly with:

- Class Diagrams: Define class-level invariants (e.g., attribute constraints).
- Use-Case Diagrams: Specify preconditions and postconditions for operations.
- State Machines: Define guard conditions and valid state transitions.

**Example:**
context Order::addItem(item: Product)
pre: not items->includes(item)
post: items->size() = items@pre->size() + 1

- *Precondition:* The item to add must not already exist.
- *Postcondition:* The number of items increases by one after the operation.

A UML *Order* class diagram connected to an *Item* class, with OCL rules attached:

- Invariant: totalAmount = items->collect(price)->sum()
- Precondition: quantity > 0
- Postcondition: balance = balance@pre - totalAmount

**Interpretation:**
This integration bridges the gap between visual modeling and formal verification, ensuring both clarity and correctness.

**Core OCL Expressions**

1. **Invariant:**
   A condition that must always hold true for a class.
2. context Account
3. inv: balance >= 0

→ An account's balance must never be negative.

4. **Precondition:**
   Must hold true before an operation executes.
5. context Account::withdraw(amount: Real)
6. pre: amount > 0 and amount <= balance

→ Withdrawal allowed only for positive amounts within balance.

7. **Postcondition:**
   Must hold true after operation completes.
8. context Account::withdraw(amount: Real)
9. post: balance = balance@pre - amount

→ Ensures account balance decreases correctly.

10. **Derived Attribute:**
11. context Customer
12. def: totalBalance : Real = accounts->collect(balance)->sum()

**Advantages of OCL**

- Adds **precision** to UML models.
- Detects **inconsistencies** early during modeling.
- Facilitates **automated validation** of UML designs.
- Non-programmatic — readable by both analysts and developers.

### 15.4 The Z Specification Language

**Overview**

The **Z language** (pronounced "Zed") is a formal specification language based on **set theory** and **first-order predicate logic**.
It was developed at **Oxford University** in the 1980s and is used to describe complex systems in a precise mathematical way.

| Concept | Description | Example |
|---|---|---|
| **Schema** | Structural unit representing part of the system — defines variables and their constraints. | Account schema with balance ≥ 0 |
| **State Schema** | Describes the data model and invariant properties of the system. | BankAccount defines balance must remain non-negative. |
| **Operation Schema** | Defines changes in system state caused by operations. | Deposit updates balance after deposit amount. |

**1. Schema**

A **Schema** is the fundamental building block of a Z specification.
It represents a **logical module** or **unit of specification**, combining both **declarations** (data) and **predicates** (constraints or rules).
Each schema is depicted as a **box divided into two parts**:

- The **upper part** lists variables and their types (the declarations).
- The **lower part** contains predicates (the constraints or properties that must always hold true).

**Structure Example:**

```
Account
------------
balance: ℤ
------------
balance ≥ 0
```

**Explanation:**

- The schema is named **Account**.

- It declares a variable balance of type **integer** (ℤ).
- The predicate section specifies that the balance must always be **greater than or equal to zero**.

**Interpretation:**
The schema acts as a **template** defining both the data structure and the invariant conditions for that part of the system.
**Analogy:**
Think of a Z schema as a **mathematical version of a class** in object-oriented programming — it defines attributes (data) and rules (invariants).

**2. State Schema**
A **State Schema** describes the **current condition or data state** of the system.
It defines **variables** that hold system data and the **invariants** that must always be true, representing the stable properties of the system.
**Example:**
BankAccount
------------
balance: ℤ
------------
balance ≥ 0
Here:

- balance represents the **state variable**.
- The invariant balance ≥ 0 ensures that the system never enters an invalid state (e.g., negative balance).

**Purpose:**

- Defines the valid states that the system can be in.
- Ensures that **all operations** preserve these invariant conditions.

**Interpretation:**
The state schema defines the **foundation** on which all operations act.
If an operation violates an invariant (e.g., resulting in a negative balance), the Z model identifies it as an **invalid operation**.
**Figure 15.3 (Text Description):**
A rectangular schema box divided into two sections — the upper section lists balance: ℤ, and the lower section contains the invariant balance ≥ 0.
This visually separates data declarations from logical constraints.

**3. Operation Schema**
An **Operation Schema** defines how the **system's state changes** in response to an operation.
It describes:

- **Inputs** (data received by the system),
- **Outputs** (data returned), and
- **State transformations** (changes to internal variables).

Z distinguishes between:

- Δ (delta): indicates the state **changes**.
- Ξ (xi): indicates the state **remains unchanged**.
- A variable with a **prime (′)** denotes the **post-operation value** of that variable.

**Example: Deposit Operation**

Deposit
ΔBankAccount
amount?: ℤ
------------
amount? > 0
balance' = balance + amount?

**Explanation:**

- ΔBankAccount means the operation modifies the account state.
- amount? denotes an **input variable**.
- The predicate section defines constraints:
  - The deposit amount must be **positive**.
  - The new balance balance' equals the old balance plus the deposit amount.

**Withdrawal Operation Example:**

Withdraw
ΔBankAccount
amount?: ℤ
------------
0 < amount? ≤ balance
balance' = balance - amount?

**Explanation:**

- Ensures that a withdrawal only occurs if sufficient funds exist.
- The invariant balance' ≥ 0 remains valid after the operation.

**Summary of Z Concepts**

| Concept | Description | Example |
|---|---|---|
| **Schema** | Structural unit representing part of the system — defines variables and their constraints. | Account schema with balance ≥ 0 |
| **State Schema** | Describes the data model and invariant properties of the system. | BankAccount defines balance must remain non-negative. |
| **Operation Schema** | Defines changes in system state caused by operations. | Deposit updates balance after deposit amount. |

---

**Key Features of Z Schemas**

1. **Clarity:** Each schema is self-contained, improving readability.
2. **Consistency:** Invariants ensure data integrity across operations.
3. **Refinement:** Complex systems can be built incrementally by composing schemas.
4. **Verifiability:** Logical proofs can confirm that all operations preserve invariants.

**Advantages of Schema-Based Modeling**

- Encourages **modularity** and **structured specification**.
- Ensures **error-free system design** by catching logical contradictions early.
- Enables **formal verification** using theorem provers (e.g., Z/EVES).
- Bridges the gap between **mathematical specification** and **system implementation**.

## 15.5 The Ten Commandments for Formal Methods

Formulated by **J. R. Abrial and colleagues**, the *Ten Commandments* guide engineers in **applying formal methods effectively**.

| **Commandment** | **Explanation** |
|---|---|
| 1. **Choose the right notation.** | Select a formal language suited to the problem (Z, B, VDM, OCL). |
| 2. **Formalize incrementally.** | Don't specify everything at once—build and refine. |
| 3. **Avoid excessive formalism.** | Use only as much mathematics as necessary for clarity. |
| 4. **Involve all stakeholders.** | Ensure specifications are understandable by non-mathematicians. |
| 5. **Validate specifications early.** | Review and simulate formal models before coding. |
| 6. **Integrate with existing methods.** | Combine formal models with UML, Agile, or testing practices. |
| 7. **Keep specifications consistent.** | Check for contradictions and ambiguities regularly. |
| 8. **Use tools to automate proofs.** | Apply theorem provers and model checkers to verify properties. |
| 9. **Maintain traceability.** | Link formal requirements to design and test cases. |
| 10. **Train the team.** | Build formal method competence through education and mentoring. |

**Commandment 1 – Choose the Right Notation**

**Principle:**

Select the formal language or notation most suitable for your system type, complexity, and team expertise.

**Explanation:**

Different formal methods languages serve different purposes:

- **Z** and **VDM** are model-based and ideal for state-oriented systems.
- **B-Method** supports stepwise refinement and tool-based proofs.
- **OCL (Object Constraint Language)** integrates with UML for object-oriented systems.

**Example:**
For safety-critical avionics, use **B-Method** (supports refinement proofs).
For a UML-based enterprise system, apply **OCL** constraints.

**Commandment 2 – Formalize Incrementally**
**Principle:**
Develop formal specifications **step-by-step** instead of attempting to specify the entire system at once.
**Explanation:**
Start with a high-level abstract model describing the overall system behavior. Then refine smaller components in detail.
This approach supports **progressive validation**, reduces risk of inconsistency, and fits iterative development.
**Example:**
In a railway signaling system, first specify train movement rules globally, then formalize sensor, switch, and signal modules incrementally.

**Commandment 3 – Avoid Excessive Formalism**
**Principle:**
Use formality **only where it adds value** — don't make the model unnecessarily complex.
**Explanation:**
Mathematical precision is important, but excessive proofs and notations can obscure understanding.
Focus on **critical parts** — safety, synchronization, data integrity — while keeping the rest semi-formal.
**Example:**
In an online booking system, formalize **payment processing logic**, but model **UI behavior** informally using UML.

**Commandment 4 – Involve All Stakeholders**
**Principle:**
Ensure that **non-mathematical stakeholders** (domain experts, users, testers) can understand and validate the specifications.
**Explanation:**
Formal models must remain **readable and communicative**. Combine formal notations with diagrams, comments, and examples.
Stakeholder review ensures that the formal specification actually matches **real user intent**.
**Example:**
In a medical software project, use graphical statecharts and natural-language explanations alongside formal Z schemas to communicate with doctors and regulators.
**Commandment 5 – Validate Specifications Early**

**Principle:**
Validate and verify the formal specification **as early as possible**, before implementation begins.
**Explanation:**
Early validation ensures that specifications are **consistent, complete, and feasible**.
Simulation tools, model checkers, and test-case generation can be used to find design flaws before coding starts.
**Example:**
Use the **SPIN model checker** to simulate communication protocols before hardware integration.

**Commandment 6 – Integrate with Existing Methods**
**Principle:**
Formal methods should **complement**, not replace, existing software processes (like Agile, UML, or testing).
**Explanation:**
Formal models can coexist with use cases, UML diagrams, and test-driven development.
Integration makes formal methods more **practical and accessible** within mainstream software engineering.
**Example:**
An Agile team uses **OCL constraints** to formally specify business rules while continuing sprint-based coding and testing.

**Commandment 7 – Keep Specifications Consistent**
**Principle:**
Maintain **consistency** within and between formal specifications, models, and system documentation.
**Explanation:**
As the system evolves, specifications must be regularly reviewed and synchronized with implementation changes.
Automated consistency checkers or theorem provers can detect contradictions.
**Example:**
In a Z specification, ensure that the defined invariant balance $\geq 0$ always holds after any deposit or withdrawal operation.

**Commandment 8 – Use Tools to Automate Proofs**
**Principle:**
Leverage **formal verification tools** to check correctness automatically wherever possible.
**Explanation:**
Manual proofing is time-consuming and error-prone. Modern tools (e.g., **Coq**, **Isabelle**, **SPIN**, **Z/EVES**, **Atelier B**) can generate or verify proofs mechanically.
Automation accelerates validation and increases confidence in correctness.

**Example:**
Using **Atelier B**, engineers at Airbus verified thousands of proof obligations automatically in the A380 flight control software.

**Commandment 9 – Maintain Traceability**
**Principle:**
Establish **traceability** between formal specifications, design models, code, and test cases.
**Explanation:**
Every formal requirement should link directly to implementation elements and validation tests.
Traceability ensures accountability, simplifies maintenance, and supports regulatory compliance.
**Example:**
In a railway interlocking system, formal requirement IDs are linked to corresponding test scenarios in the verification report.

**Commandment 10 – Train the Team**
**Principle:**
Invest in **education and mentoring** to build formal methods competence across the team.
**Explanation:**
The success of formal methods depends on skilled practitioners who understand both the mathematical foundations and the practical aspects of software engineering.
Training ensures correct application and long-term sustainability.
**Example:**
Organizations like NASA and Airbus conduct specialized internal training in Z, B, and model checking for their engineers.

**Essence of the Ten Commandments**
Together, these principles emphasize that formal methods should be:
- **Selective** — applied to critical components.
- **Incremental** — developed step by step.
- **Understandable** — usable by all stakeholders.
- **Integrated** — compatible with other engineering practices.
- **Sustainable** — supported by tools and training.

The Ten Commandments transform formal methods from a purely academic exercise into a **practical engineering discipline**.
They remind practitioners that **formality must enhance understanding, not hinder it**, and that **mathematical precision should always serve the goal of building safer, more reliable software.**
When applied wisely—selectively, collaboratively, and with the aid of tools—formal methods can dramatically **improve software quality, reduce defects, and increase confidence in critical systems**.

## 15.6 Summary

Formal methods introduce **mathematical rigor** into software engineering, ensuring that specifications are **consistent, complete, and correct**.
They rely on languages like **OCL** for constraint-based modeling and **Z** for model-based specification.
When applied judiciously—guided by the Ten Commandments—formal methods enhance **reliability, safety, and verifiability**, especially in mission-critical systems.

## 15.7 Technical Terms

1. Formal Methods
2. Specification
3. Verification
4. OCL (Object Constraint Language)
5. Invariant
6. Precondition / Postcondition
7. Z Language
8. Schema
9. Predicate Logic
10. Theorem Prover

## 15.8 Self-Assessment Questions

**Essay Questions**

1. Define formal methods and explain their importance in software engineering.
2. Explain the role of OCL in enhancing UML models with examples.
3. Describe the structure and syntax of the Z specification language.
4. Discuss the Ten Commandments for Formal Methods and their significance.
5. Compare model-based and property-based approaches in formal specification.
6. Explain how formal methods contribute to software verification and validation.

**Short Notes**

1. Formal Specification
2. Proof of Correctness
3. State Schema and Operation Schema in Z
4. OCL Invariants
5. Model Checking
6. Ten Commandments (summary points)

## 15.9 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice Hall, 1992.

3. Meyer, Bertrand, *Object-Oriented Software Construction*, Prentice Hall, 2000.
4. Wing, J.M., *A Specifier's Introduction to Formal Methods*, IEEE Computer, 1990.
5. ISO/IEC 19507:2012, *Object Constraint Language (OCL)*, OMG Standard.
6. Hinchey, M. & Bowen, J.P., *Applications of Formal Methods*, Prentice Hall, 1995.
7. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.
8. Jackson, Daniel, *Software Abstractions: Logic, Language, and Analysis*, MIT Press, 2012.

Dr. U. Surya Kameswari

LESSON- 16

# CLEANROOM SOFTWARE ENGINEERING

**AIMS AND OBJECTIVES**

To understand the Cleanroom approach to software engineering — a disciplined, formal, and statistical process for developing high-quality, error-free software with mathematical correctness and controlled reliability assessment.

**After completing this lesson, you will be able to:**

1. Define the Cleanroom Software Engineering (CSE) philosophy and its goals.
2. Describe the Cleanroom process model and its major components.
3. Explain how functional specification and formal verification are applied in Cleanroom development.
4. Understand the principles of Cleanroom design using hierarchical structure and correctness proofs.
5. Discuss Cleanroom testing and its focus on statistical reliability rather than debugging.
6. Appreciate how the Cleanroom approach supports zero-defect software and continuous quality improvement

**STRUCTURE**

## 16.1 Introduction to Cleanroom Software Engineering

**Cleanroom Software Engineering (CSE)** is a formalized, process-driven approach to software development that emphasizes **error prevention** rather than error correction.
It was developed at **IBM Federal Systems Division (mid-1980s)** by **Harlan D. Mills** and colleagues.
The name *"Cleanroom"* comes from **hardware manufacturing**, where microchips are built in dust-free environments — symbolizing **defect-free software creation** through discipline and formality.

**Philosophy**
The core philosophy of Cleanroom engineering is:
*"Don't test for correctness — develop software correctly the first time."*
It replaces the traditional "code → test → fix" cycle with a **"specify → design → verify → certify"** paradigm.
Errors are avoided through:

- **Formal specification**
- **Incremental development**
- **Statistical usage testing**
- **Team-based correctness verification**

**The Traditional Cycle vs. Cleanroom Paradigm**

| **Traditional Development** | **Cleanroom Engineering** |
|---|---|
| Errors discovered late during testing. | Errors prevented through design correctness proofs. |
| Debugging dominates the schedule. | Verification and statistical testing dominate. |
| Reliance on trial and error. | Reliance on mathematical and process discipline. |
| Quality measured by defect count. | Quality measured by reliability certification. |
| Testing seeks to find bugs. | Testing quantifies reliability, not defects. |

**Key Principles Underlying the Philosophy**

1. **Formal Specification:**
   System behavior is defined mathematically — eliminating ambiguity and ensuring completeness before any coding begins.
   Every functional requirement has a corresponding formal model or invariant.
   *Example:* In a banking system, balance invariants (balance $\geq 0$) are specified formally and verified at each refinement step.
2. **Incremental Development:**
   The system is developed and verified in **small, certifiable increments**.
   Each increment is statistically tested and certified before integration.
   This ensures **controlled reliability growth** and easy traceability.
3. **Statistical Usage Testing:**
   Instead of exhaustive or arbitrary test cases, testing is based on **real-world usage probabilities**.
   The aim is not to locate defects but to **measure operational reliability** using scientific, repeatable methods.
4. **Team-Based Correctness Verification:**
   Teams review and verify every specification, design, and code artifact using **formal reasoning techniques**.
   Collective verification replaces individual debugging, ensuring **shared accountability for correctness.**

**Goals of Cleanroom Approach**

- Achieve **ultra-high reliability** and **certified quality**.
- Reduce time and cost of debugging.
- Promote **discipline and mathematical precision**.
- Support **incremental delivery** with measurable reliability growth.

**1. Achieve Ultra-High Reliability and Certified Quality**
Cleanroom aims for near-zero defect density (often below 0.1 defects per KLOC) and statistically certified reliability levels of 99.9% or higher.
By combining formal verification and statistical testing, it produces software suitable for safety-critical and mission-critical systems, such as avionics, medical devices, and banking networks.
*Example:*
IBM's Cleanroom projects achieved defect densities 10 times lower than comparable conventionally developed systems.

**2. Reduce Time and Cost of Debugging**
Traditional software projects can spend 40–60% of effort on debugging and rework.
Cleanroom eliminates debugging as a post-testing activity — defects are removed at the specification and design stages.
Since software is verified before testing, failures found during statistical testing indicate process-level issues, not code errors.
Outcome:

- Lower maintenance costs.
- Fewer post-release failures.
- Shorter development cycles for future increments.

**3. Promote Discipline and Mathematical Precision**
Cleanroom emphasizes formal methods, structured design, and correctness proofs as engineering disciplines.
Mathematical rigor replaces intuition, ensuring every design decision is logically justified.
Key Techniques:

- *Formal specification* (Z, VDM, FSM models).
- *Correctness verification* using logical assertions.
- *Box Structure Methodology* for traceable refinement.

This formal discipline ensures that system behavior is provably correct with respect to its specification.

**4. Support Incremental Delivery with Measurable Reliability Growth**
Cleanroom development is incremental — the system is delivered in verified subunits, each statistically tested and certified.
Reliability grows steadily as each increment contributes verified functionality.
Reliability Growth Example:

| **Increment** | **Scope** | **Reliability Achieved** |
|---|---|---|
| **1** | **Core Login & Authentication** | **99.90%** |
| **2** | **Balance Inquiry & Transfer** | **99.95%** |
| **3** | **Bill Payment & Alerts** | **99.97%** |
| **4** | **Full Deployment** | **99.99%** |

This approach enables continuous quality improvement with each release, supporting business needs for early delivery and high assurance.

**Summary of Cleanroom Goals**

| **Goal** | **Description** | **Outcome** |
|---|---|---|
| **Ultra-High Reliability** | **Achieve statistically certified reliability (≥ 99.9%).** | **Defect-free, mission-critical systems.** |
| **Eliminate Debugging** | **Prevent errors through verification instead of testing.** | **Reduced cost and faster delivery.** |
| **Mathematical Discipline** | **Apply formal methods to ensure provable correctness.** | **Predictable, traceable quality.** |
| **Incremental Certification** | **Deliver verified increments with measurable reliability growth.** | **Continuous improvement and early deployment.** |

## 16.2 The Cleanroom Approach

**Overview**

The Cleanroom process model focuses on **defect prevention** and **process maturity** rather than debugging.
It integrates **formal methods**, **incremental development**, and **statistical testing** to ensure correctness and reliability.

**Key Characteristics of the Approach**

1. **Incremental Development:**
   Software is developed in small, certified increments, each verified and statistically tested before integration.
2. **Formal Specification:**
   Requirements and system behavior are specified using **mathematical models** (e.g., FSMs, formal logic).
3. **Formal Design Verification:**
   Every design step is **mathematically verified** to meet the specification.
4. **Statistical Usage Testing:**
   Testing is driven by **usage profiles**, representing real-world operation probabilities — measuring reliability quantitatively.
5. **No Debugging Culture:**
   Failures found during testing are analyzed, but **code is never modified** in response to test results — instead, the cause is traced back to process improvement.

**Process Framework**

| **Phase** | **Activity** | **Outcome** |
|---|---|---|
| **Specification** | Define functions and expected behaviors formally. | Correct and complete requirements. |
| **Design** | Transform formal specs into structured designs. | Verified design model. |
| **Implementation** | Develop code directly from verified designs. | Correct, readable code. |
| **Certification** | Conduct statistical testing to measure reliability. | Certified reliability rating. |

## 16.3 Functional Specification

**Purpose**

The **functional specification** defines **what the software must do** using **formal notations** that remove ambiguity.
It serves as a **mathematical contract** between customer and developer.

**Features**

1. Written in **state-based** or **functional notation** (e.g., Z, FSM, or VDM).
2. Defines system inputs, outputs, and transformation rules.
3. Provides basis for **verification** and **test generation**.
4. Serves as reference for all subsequent design and certification steps.

**Specification Techniques**

- **Black-box specification:** Describes the system in terms of external stimuli and responses.
- **State-transition specification:** Models system as a **finite-state machine** (FSM).
- **Formal assertions:** Represent correctness conditions using logical expressions.

**Example: Banking System Specification**
Inputs: deposit(amount), withdraw(amount)
Outputs: new balance
Invariant: balance ≥ 0
Operation: deposit increases balance, withdraw decreases if sufficient funds.
**Interpretation:**
The Cleanroom functional specification uses **mathematical precision** to define behaviors unambiguously.

## 16.4 Cleanroom Design

**Goal**

The goal of Cleanroom design is to transform the formal specification into a **verified design structure** that is both correct and maintainable.
It uses **structured programming constructs** and **hierarchical refinement** to derive the implementation.

**Core Design Principles**

1. **Box Structure Methodology (BSM):**
   Cleanroom design uses *three hierarchical boxes* to separate concerns:
   - **Black Box:** Describes external behavior.
   - **State Box:** Describes data and state transitions.
   - **Clear Box:** Defines procedural design and algorithms.

```
          +---------------------+
|     Black Box      | → External Specification
          +---------------------+
                   ↓
          +---------------------+
|     State Box      | → Data & Transition Logic
          +---------------------+
                   ↓
          +---------------------+
|     Clear Box      | → Implementation Algorithm
          +---------------------+
```

**Figure 16.1 Box Structure Methodology**

2. **Stepwise Refinement:**
   Each box is expanded and refined in small steps until code-level detail is achieved.

3. **Correctness Verification:**
   Each refinement is accompanied by a **formal proof** that it satisfies its predecessor (e.g., State Box satisfies Black Box).
4. **Incremental Integration:**
   Cleanroom teams integrate only **verified modules** — preventing error propagation.

**Design Example (ATM System)**

- **Black Box:** User inserts card → System requests PIN → Returns balance.
- **State Box:** Tracks state variables like cardInserted, authenticated.
- **Clear Box:** Implements actual logic for card validation, PIN verification, and display.

**Outcome:**
Each level ensures **correct transformation** and **formal traceability**.

**Design Example: ATM System (Expanded)**
To illustrate how Cleanroom design principles are applied, let us consider the development of an **Automated Teller Machine (ATM)** software system.
The ATM allows users to:

- Insert their bank card,
- Enter a Personal Identification Number (PIN),
- Perform balance inquiries, and
- Withdraw cash.

The Cleanroom design process models this system using the **Box Structure Methodology (BSM)** — a hierarchical, correctness-preserving design technique.

**Overview of the Box Structure Methodology**
The **Box Structure Methodology**, introduced by Harlan D. Mills, divides system design into three hierarchical abstraction layers:

1. **Black Box** – Defines *what* the system does (external behavior).
2. **State Box** – Defines *how* the system behaves internally (state transitions).
3. **Clear Box** – Defines *how* the system is implemented procedurally (algorithmic details).

Each layer refines the previous one, ensuring **correctness preservation** through formal verification.

**1. Black Box Design**
The **Black Box** represents the **external view** of the system — how the user perceives it.
It describes the **input–output behavior** without revealing internal logic or data structures.
**Black Box Description (ATM System)**

| **Aspect** | **Description** |
|---|---|
| **Purpose** | Define user-visible functions of the ATM. |
| **Inputs** | Card insertion, PIN entry, menu selection (e.g., “Balance Inquiry”, “Withdraw”). |
| **Outputs** | Displayed balance, confirmation message, dispensed cash. |
| **Behavioral Specification** | For every input, specify corresponding output or system response. |

**Functional Behavior**
User inserts card → System requests PIN
User enters valid PIN → System displays main menu
User selects “Balance Inquiry” → System shows current balance
User selects “Withdraw” → System requests amount and validates availability

User removes card → System terminates session
**Interpretation:**
This model treats the ATM as a **black box** — only external inputs (user actions) and outputs (system responses) are visible.
Internal details such as data validation, state transitions, or encryption mechanisms are hidden at this level.
**Benefits:**

- Defines precise user-level expectations.
- Supports **functional verification** independent of implementation.
- Acts as a foundation for **formal specification** in Cleanroom development.

**2. State Box Design**
The **State Box** refines the Black Box by introducing **state variables** and **transition logic** that describe how the system behaves internally between inputs and outputs.
This level bridges abstract functionality (Black Box) and implementation logic (Clear Box).
**State Box Description (ATM System)**

| **Aspect** | **Description** |
|---|---|
| **Purpose** | Define how the ATM maintains and transitions between operational states. |
| **State Variables** | cardInserted, authenticated, selectedOperation, balance. |
| **Inputs** | Same as Black Box (user actions). |
| **State Transitions** | Describes how each input changes system state. |

**State Transition Model**

| **Current State** | **Input** | **Next State** | **Output** |
|---|---|---|---|
| Idle | Insert Card | CardInserted | Display “Enter PIN” |
| CardInserted | Enter Valid PIN | Authenticated | Display Main Menu |
| Authenticated | Select “Balance Inquiry” | ShowBalance | Display Account Balance |
| Authenticated | Select “Withdraw” | WithdrawalMode | Request Amount |
| WithdrawalMode | Enter Amount | Authenticated | Dispense Cash |
| Authenticated | Remove Card | Idle | Display “Thank You” |

**State Variable Rules**

- cardInserted = true only after physical card detection.
- authenticated = true only when PIN is validated.
- balance is updated only after successful withdrawal operation.

**Interpretation:**
The **State Box** explicitly models **control logic** and **data flow** inside the system.
It guarantees that only valid state transitions occur (e.g., withdrawal cannot happen before authentication).
**Formal Invariant Example:**
authenticated ⇒ cardInserted
→ The system cannot be authenticated unless a card is inserted.

**3. Clear Box Design**
The **Clear Box** level provides the **implementation view** — describing *how* the system achieves the specified behavior through algorithms, control structures, and data manipulation.
Here, the state transitions from the State Box are refined into **step-by-step procedural logic** or pseudocode.
**Clear Box Design (ATM System)**

**Procedure: ATM_Main()**

```
1. Wait for card insertion
2. Read card data → set cardInserted = true
3. Prompt for PIN entry
4. If (verifyPIN(PIN) == true)
       authenticated = true
       Display main menu
       While (sessionActive)
          If (option == "Balance Inquiry")
             Display current balance
          Else if (option == "Withdraw")
             Request amount
             If (amount ≤ balance)
                balance = balance - amount
                Dispense cash
                Display "Transaction Successful"
             Else
                Display "Insufficient Funds"
          EndIf
       EndWhile
   Else
       Display "Invalid PIN"
   EndIf
5. Eject card and reset all states
```

**Notes:**

- The procedural logic directly maps from **State Box transitions**.
- Each logical path maintains state invariants (e.g., balance $\geq 0$).
- Code generation is straightforward since the design is already verified.

**Verification Between Boxes**

Each box level is **verified** to ensure correctness against its predecessor:

| Verification Step | Goal |
|---|---|
| Black Box → State Box | Verify that internal states cover all functional behaviors. |
| State Box → Clear Box | Verify that procedural logic correctly implements state transitions. |
| Clear Box → Code | Code generation preserves correctness proofs. |

**Formal Verification Example:**

For every operation in the Clear Box:

PRE: $\text{authenticated} = \text{true} \wedge \text{amount} > 0$

POST: $\text{balance}' = \text{balance} - \text{amount} \wedge \text{balance}' \geq 0$

This ensures **withdrawal correctness** is mathematically proven.

**Benefits of Cleanroom ATM Design**

1. **Error Prevention:**
   Each refinement is verified formally, preventing logical inconsistencies.
2. **Traceability:**
   Every Clear Box construct maps directly to a Black Box function and State Box transition.
3. **Reliability:**
   Correctness proofs ensure invariant preservation (no overdraft or unauthorized access).
4. **Maintainability:**
   Clear modular separation simplifies updates (e.g., adding "Deposit" function).

**Summary of Box Structure for ATM**

| Box Type | Focus | Content | Example in ATM System |
|---|---|---|---|
| **Black Box** | External behavior | Input-output relationship | Card insertion → Balance inquiry → Output balance |
| **State Box** | Logical behavior | State variables and transitions | Tracks cardInserted, authenticated, etc. |
| **Clear Box** | Implementation logic | Algorithms and control flow | Procedural steps for validation and transaction |

The **ATM System example** demonstrates how Cleanroom design separates concerns across three abstraction levels, ensuring that each step is **formally correct and verifiable**.By progressing systematically from **Black Box (what)** to **Clear Box (how)**, the development process maintains **mathematical consistency**, **traceability**, and **high reliability** — the core objectives of Cleanroom Software Engineering.

## 16.5 Cleanroom Testing

**Objective**

Unlike traditional testing aimed at *finding and fixing defects*, Cleanroom testing focuses on **measuring reliability** and **certifying correctness** statistically.

**Key Concepts**

1. **Statistical Usage Testing:**
   - Tests are designed according to **usage profiles** — probability distributions of real-world operations.
   - Each test run reflects expected user behavior frequencies.
   - Helps determine *mean time to failure (MTTF)* and *software reliability* quantitatively.
2. **Reliability Certification:**
   - The software's reliability (R) is estimated as a measurable value (e.g., 0.9999 success probability per operation).
   - Used to demonstrate **quantitative quality** before release.
3. **No Debugging During Testing:**
   - o Any failure during testing is recorded and analyzed.
   - o Source code is not modified; instead, the process is reviewed and improved in future increments.

**Testing Process Flow**

| Step | Description | Outcome |
|---|---|---|
| 1. Develop Usage Profile | Define typical user operations with probabilities. | Usage model ready. |
| 2. Generate Test Cases | Randomly select test cases based on profile. | Representative test set. |
| 3. Execute Tests | Run tests on certified increments. | Failure data collected. |
| 4. Compute Reliability | Use statistical models (e.g., exponential distribution). | Reliability metrics. |
| 5. Certify Product | Compare results with target reliability. | Certified release. |

**Step 1: Develop Usage Profile**
A **usage profile** defines how various operations are expected to be executed during normal use.
It is usually expressed as a **probability distribution** of actions or functions.
**Example – ATM System Usage Profile**

| **Operation** | **Probability of Use (p)** |
|---|---|
| Insert Card | 0.05 |
| Enter PIN | 0.10 |
| Balance Inquiry | 0.25 |
| Withdraw Cash | 0.40 |
| Deposit Cash | 0.10 |
| Cancel / Exit | 0.10 |

This data ensures that testing reflects **real user activity frequencies** rather than random or exhaustive test case generation.

**Step 2: Generate Test Cases**
Test cases are generated automatically or manually according to the **usage profile**. The goal is to **sample real-world usage scenarios**, not to test every possible path.
For instance, if 40% of users perform withdrawals, then 40% of the test cases should focus on withdrawal operations.
**Example:**
For 10,000 total test cases:

- 4,000 → Withdraw
- 2,500 → Balance Inquiry
- 1,000 → Deposit
- Remaining → Other actions

This ensures a statistically representative test suite.

**Step 3: Execute Tests**
Tests are executed under **controlled, monitored conditions**.
Each test case simulates a user transaction or operation and records:

- Whether it succeeded or failed.
- The time between failures (for reliability analysis).
- Environmental variables (e.g., load, input sequence).

**Key Principle:**
Code is **never modified** during Cleanroom testing — testing measures reliability, not correctness.
Failures are documented for statistical evaluation, not debug correction.

**Step 4: Compute Reliability**
After test execution, the data is used to compute quantitative reliability metrics.
Two important measures are:

1. **Reliability (R):**

$$R = 1 - \frac{\text{Number of Failures}}{\text{Total Test Cases}}$$

It represents the probability that a transaction will execute successfully.

2. **Mean Time To Failure (MTTF):**

$$MTTF = \frac{\text{Total Execution Time}}{\text{Number of Failures}}$$

It represents the expected operational time between failures.

3. **Failure Rate (λ):**

$$\lambda = \frac{1}{MTTF}$$

It represents how frequently failures occur during operation.

**Interpretation:**

Higher reliability (R close to 1) and longer MTTF indicate a more dependable system.

**Step 5: Certify Product**

When the computed reliability meets or exceeds predefined thresholds, the software is **certified for release**.

Otherwise, the failure data is analyzed to identify the **root cause** — often a process or design defect — and improvements are made for the next iteration.

**Example Thresholds:**

| **Application Type** | **Target Reliability** |
|---|---|
| Banking / Financial | 99.99% |
| Avionics | 99.999% |
| Consumer Web App | 99.90% |
| Educational Software | 99.50% |

Certification ensures that software meets **quantitative reliability standards**, not just functional correctness.

**Example: Reliability Estimation (Expanded)**

Let us calculate the reliability of a Cleanroom-developed system based on test results.

**Test Data**

- Total transactions tested = **10,000**
- Failures observed = **2**

**Reliability Calculation**

$$R = 1 - \frac{2}{10{,}000} = 1 - 0.0002 = 0.9998$$

So, the system's **operational reliability** is **99.98%**.

**MTTF Calculation**

Assume total testing time = **500 hours**.

$$MTTF = \frac{500}{2} = 250 \text{ hours}$$

This means, on average, the system can operate for **250 hours before a failure occurs**.

**Interpretation**

- The system achieves **0.9998 reliability**, which translates to **2 failures per 10,000 operations**.
- If the reliability goal was ≥ **99.95%**, this product qualifies for **certified release**.

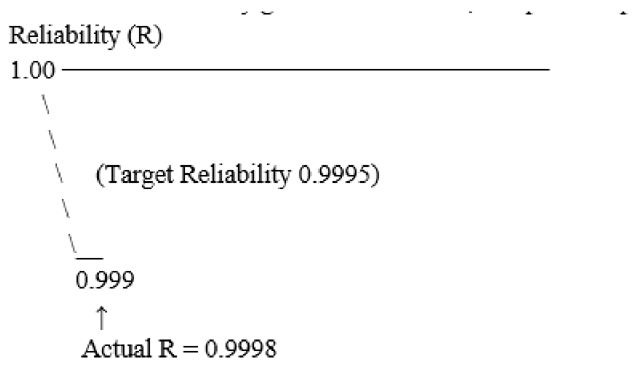


**Fig 16.2 Reliability Curve**

This curve illustrates that the observed reliability exceeds the certification threshold.

**Advantages of Cleanroom Testing Approach**

| **Aspect** | **Traditional Testing** | **Cleanroom Testing** |
|---|---|---|
| **Objective** | Find and fix defects | Measure and certify reliability |
| **Basis** | Functional test cases | Statistical usage profile |
| **Test Focus** | Code coverage | Operational reliability |
| **Response to Failure** | Debug and recompile | Record and analyze; no code change |
| **Outcome** | Bug fixes | Certified reliability metric |

**Result:** Cleanroom testing ensures **quantifiable, repeatable reliability certification** rather than subjective "it works" validation.

**Key Insights**

- Reliability certification quantifies **user-level trust** in software.
- Statistical testing models **real-world behavior**, ensuring meaningful results.
- Because debugging is not part of testing, the process drives **defect prevention** through disciplined design.
- Reliability measurement supports **continuous improvement** and **predictable software performance**.

The Cleanroom testing process treats testing as a **scientific reliability experiment**, not a debugging task.By combining **usage-based test case generation** with **statistical analysis**, it provides objective, numerical proof that the software meets its reliability targets.This approach transforms software validation from a **subjective activity** into a **quantitative certification process**, setting a higher standard for software quality assurance.

**Benefits of Cleanroom Testing**

- Quantifiable reliability metrics.
- Reduced cost due to **zero debugging**.
- Prevents defect injection early.
- Builds **confidence in correctness**.

## 16.6 Summary

The **Cleanroom Software Engineering** process emphasizes **formal specification**, **design verification**, and **statistical certification** instead of traditional debugging. By combining mathematics, structured design, and disciplined process control, it produces software of **exceptionally high quality** and **predictable reliability**. The focus shifts from *error detection* to *error prevention*, aligning engineering rigor with industrial practicality.

## 16.7 Technical Terms

1. Cleanroom Process
2. Formal Specification
3. Box Structure Methodology
4. Incremental Development
5. Correctness Verification
6. Statistical Usage Testing
7. Reliability Certification
8. No Debugging Principle
9. Mean Time to Failure (MTTF)
10. Reliability Growth Model

## 16.8 Self-Assessment Questions

### Essay Questions

1. Explain the philosophy and objectives of the Cleanroom approach.
2. Discuss the stages of the Cleanroom process model.
3. Describe the Box Structure Methodology (Black Box, State Box, Clear Box).
4. Explain how statistical usage testing differs from traditional testing.
5. Illustrate Cleanroom functional specification with an example.
6. How does Cleanroom ensure correctness without debugging?

### Short Notes

1. Incremental Development
2. Formal Specification
3. Correctness Verification
4. Usage Profile
5. Reliability Certification

## 16.9 Suggested Readings

1. Pressman, Roger S. & Maxim, Bruce R., *Software Engineering: A Practitioner's Approach*, 7th Ed., McGraw-Hill, 2014.
2. Mills, Harlan D., Dyer, Michael, & Linger, Richard C., *Cleanroom Software Engineering*, IEEE Software, 1987.
3. Linger, Richard C., *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1993.
4. Whittaker, James A., *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2003.
5. Musa, John D., *Software Reliability Engineering*, McGraw-Hill, 1999.
6. IEEE Std 982.2–1988, *Guide for Software Quality and Reliability Measurement*.

Dr. U. Surya Kameswari