Master of Computer Applications (MCA)

Software Engineering

(OMCACO202T24)

Self-Learning Material (SEM II)

Jaipur National University
Centre for Distance and Online Education

Established by Government of Rajasthan
Approved by UGC under Sec 2(f) of UGC ACT 1956
&
NAAC A+ Accredited

TABLE OF CONTENTS

EXPERT COMMITTEE

Prof. Sunil Gupta
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Deepak Shekhawat
(Department of Computer and Systems Sciences, JNU Jaipur)

Dr. Shalini Rajawat
(Department of Computer and Systems Sciences, JNU Jaipur)

COURSE COORDINATOR

Mr. Shish Kumar Dubey, Asst. Prof.
(Department of Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	Assisting & Proofreading	Unit Editor
Mr. Pawan Jakhar (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 1-4)	Mr. Satender Singh (Department of Computer and Systems Sciences, JNU Jaipur)	Mr. Hitendra Agarwal (Department of Computer and Systems Sciences, JNU Jaipur)
Ms. Rachana Yadav (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 5- 8)		
Ms. Heena Shrimali (Department of Computer and Systems Sciences, JNU Jaipur) (Unit 9-12)		

Secretarial Assistance

Mr. Mukesh Sharma

COURSE INTRODUCTION

"Clarity is the cornerstone of effective software engineering."

- Austin Freeman

Software Engineering is a crucial discipline in computer science focused on the systematic design, development, testing, and maintenance of software systems. This course provides a comprehensive overview of the principles and practices that guide the creation of robust, efficient, and reliable software. As technology continues to evolve and software becomes increasingly integral to all aspects of life, understanding software engineering becomes essential for anyone involved in the development and management of software systems. This course aims to equip students with the foundational knowledge and practical skills needed to navigate the complexities of software engineering effectively.

This course has 3 credits and is divided into 12 Units. The course begins with an introduction to the software development lifecycle (SDLC), covering various models such as Waterfall, Agile, and DevOps. Students will explore how these methodologies impact project management, development processes, and team dynamics. The Waterfall model, with its linear and sequential approach, contrasts with Agile methodologies, which emphasize iterative development, flexibility, and collaboration. DevOps, combining development and operations practices, highlights the importance of continuous integration and delivery in modern software projects. By understanding these models, students will gain insights into how to manage and execute software projects in different contexts.

Requirements analysis focuses on understanding and documenting what the software needs to achieve, involving stakeholders to ensure that all functional and non-functional requirements are met. Design principles cover architectural patterns, such as MVC (Model-View-Controller) and micro services that influence how software components are structured and interact. Implementation involves writing code and integrating various software modules, while testing ensures that the software meets quality standards and functions correctly. Maintenance addresses the ongoing support and enhancement of software systems post-deployment.

The course also emphasizes best practices in software engineering, including version control, documentation, and code quality. Students will learn about tools and techniques for managing code changes and collaborating with team members.

Course Outcomes:

At the completion of the course, a student will be able to:

1. Learn different software engineering approaches to resolve different software crises like failure in operation, non-meeting of requirements, delayed delivery, over budget.
2. Compare different software process models to find the appropriate one.
3. Apply 4 GL techniques to develop software systems.
4. Develop and manage software projects from project initiation to project closure.
5. Develop quality software systems with the latest tools and techniques.

Acknowledgements:

The content we have utilized is solely educational in nature. The copyright proprietors of the materials reproduced in this book have been tracked down as much as possible. The editors apologize for any violation that may have happened, and they will be happy to rectify any such material in later versions of this book.

Unit 1
Introduction to Software Engineering

Learning Objectives:

- To understand the importance of software engineering.
- To understand the emergence of software engineering practices.
- To find the impact of the best software engineering methodology.
- To explore the various software engineering concepts.
- To go through the various changes that occur red in software engineering concepts and methodology.
- To understand system engineering.
- To understand the role of system analysts in the development process.

Structure:

Introduction: Software Engineering

In the earlier stages of computer programming, when computer processing speeds were relatively slow, programming tasks were typically carried out by individuals or small teams. Programmers relied heavily on assembly language to accomplish specific tasks. However, as programming languages advanced, the approach to writing applications underwent significant changes.

By the early 1960s and 1970s, the concept of software engineering began to gain prominence. It became apparent that developing enterprise-scale applications without proper analysis of requirements would not be cost-effective or time-efficient. This recognition marked a significant shift in the way software development was approached.

Following the era of low-level programming languages, high-level programming languages emerged in the computing industry. Practitioners began using flowcharts to represent algorithms, facilitating a more structured approach to programming. The term "software engineering" gained further recognition in 1968 at a conference, solidifying its importance in the computing industry.

Around 1980, Object-Oriented Programming (OOP) gained prominence, revolutionizing the way software was designed and developed. OOP introduced concepts such as encapsulation, inheritance, and polymorphism, providing developers with powerful tools for building complex software systems.

In the 1990s, Agile software development methodology emerged as a response to the need for faster software development cycles and the ability to adapt to changing requirements. Agile methodologies, such as Scrum and Extreme Programming (XP), prioritize flexibility and collaboration, allowing teams to deliver working software incrementally and respond quickly to evolving customer needs.

Today, software engineering has become a stable and disciplined practice in the software industry. It has evolved over time and matured, incorporating best practices, methodologies, and tools that play a vital role in the software development process. From requirements analysis to

design, implementation, testing, and maintenance, software engineering principles guide developers in creating robust, scalable, and high-quality software solutions.

1.1 History of Software Engineering

In the early years of computer programming, when computing power was limited, software development was typically carried out by small teams or even individuals. Programmers primarily used assembly language to write programs, performing tasks at a low level. However, as programming languages evolved, so did the approach to software development. By the early 1960s and 1970s, the term "software engineering" began to gain prominence, highlighting the importance of structured and disciplined approaches to software development. This shift emphasized the need for thorough analysis of requirements before embarking on the development process, recognizing that creating large-scale applications without proper planning would be inefficient in terms of both time and cost.

Following the era of low-level programming languages, high-level programming languages emerged in the computing industry. Practitioners began utilizing flowcharts to visually represent algorithms, aiding in the design and development process. The significance of software engineering was further underscored in 1968 at a conference, solidifying its importance in the computing field.

Around 1980, Object-Oriented Programming (OOP) gained traction, fundamentally altering the way software was designed and developed. OOP introduced concepts such as encapsulation, inheritance, and polymorphism, offering developers powerful tools for creating more modular and maintainable software systems.

In the 1990s, Agile software development methodologies emerged as a response to the need for faster adaptation to changing requirements and quicker delivery of software. Agile methodologies, including Scrum and Extreme Programming (XP), prioritize flexibility and collaboration, enabling teams to respond promptly to evolving customer needs.

Today, Software Engineering has become a well-established and disciplined practice in the software industry. It has evolved over time, incorporating best practices, methodologies, and tools that are essential for the development process. From requirements analysis to design, implementation, testing, and maintenance, Software Engineering principles guide developers in creating robust, scalable, and high-quality software solutions.

1.2 Importance of Software Engineering

Let's delve into a scenario analogous to building a house: Suppose someone embarks on constructing their home without conducting thorough analysis of costs, labor, and materials. As the construction progresses, they continually invest money in adding new parts to the house. However, due to insufficient financial resources or other constraints, they eventually halt the project, resulting in its failure.

Several factors contribute to the failure of the project:

a. **Lack of Proper Requirement Analysis**: Without a comprehensive understanding of the homeowner's needs and preferences, the construction lacks direction and may not meet their expectations.

b. **Inadequate Building Design**: Absence of a well-thought-out architectural plan leads to haphazard construction, potentially resulting in structural flaws and inefficiencies.

c. **Absence of Cost Analysis**: Without assessing the financial implications of the project beforehand, there is no clarity on budget constraints, leading to overspending and financial strain.

d. **Failure to Analyze Required Changes**: Inability to anticipate and accommodate changes during the construction process can lead to delays, rework, and additional costs.

e. **Lack of Time Analysis**: Without a timeline for completing various stages of construction, there is a risk of indefinite delays and project stagnation.

Similarly, in software development, proper planning and analysis are crucial for success. While software is intangible, it requires meticulous planning and design to avoid complexity and ensure project success. Software engineering, as a disciplined approach to software development, offers several benefits:

a. **Cost-Effectiveness**: Proper software engineering practices reduce development, testing, and maintenance costs by streamlining processes and optimizing resources.

b. **Time Efficiency**: Software engineering methodologies enable accurate estimation of project timelines, facilitating timely delivery.

c. **Quality Maintenance**: Systematic approaches to software design ensure high-quality outputs, aided by tools and methodologies for quality assurance.

d. **Complexity Reduction**: Effective planning and tools break down complex software problems into manageable components, simplifying the development process.

e. **Improved Communication**: Clear design and requirement documentation enhance communication among team members and stakeholders, reducing misunderstandings and errors.

f. **Increased Productivity**: Clear design and well-defined requirements improve productivity by providing clarity and direction to development efforts.

g. **Easy Maintenance**: Systematically designed software projects have lower maintenance costs and are easier to manage and update.

h. **Scalability**: Software engineered with scalability in mind can accommodate future changes and updates without major disruptions to the system.

By embracing software engineering principles, developers can navigate complex software projects more effectively, leading to successful outcomes and satisfied stakeholders.

1.3 Changes in Software Development Practices

Software development is a complex process encompassing various stages such as analysis, design, development, testing, and maintenance. Over time, the practices followed in software development have evolved, driven by changes in programming languages, technology, resource availability, and work culture. These changes have led to the emergence of trending practices aimed at ensuring the quality, security, and timely delivery of software products.

Some of the trending practices in the software development industry include:

a. Transition from Traditional Waterfall to Agile Methodology: The waterfall model, known for its rigid structure, often struggled to accommodate changes. Agile methodology, on the other hand, emphasizes flexibility and responsiveness to change, involving stakeholders in the development process to provide feedback and ensure continuous improvement.

b. DevOps: DevOps methodology integrates development and IT operations teams to deliver fast and high-quality software. Automation tools and technologies streamline development processes, reducing development time and enhancing collaboration between teams.

c. Cloud Computing: Cloud computing has become a prevalent practice in software development, offering scalability and resource sharing benefits. Cloud services like Google Cloud Platform, Amazon Web Services, and Microsoft Azure have transformed the software development process by providing flexible infrastructure and platform management.

d. Automated Testing: Testing is now predominantly done using automated tools such as Selenium, Cucumber, and Robot Framework. Automation in testing reduces the time and effort required for testing, ensuring high accuracy and speed in the testing process.

e. Microservices Architecture: Microservices break down applications into small, independently deployable modules, facilitating scalability and fault isolation. This architecture enables developers to develop and deploy modules independently, resulting in faster development cycles and easier error detection.

f. Container Orchestration Tools: Container orchestration tools like Docker simplify the development and maintenance process by automating tasks such as scaling and load balancing. These tools provide a lightweight environment for running applications efficiently.

g. Big Data: With the increasing volume of data, data security has become a significant concern for organizations. Practices like authentication and authorization play a crucial role in safeguarding sensitive data from unauthorized access and misuse.

h. Machine Learning: Machine learning is increasingly used in software development to develop prediction models and enhance business intelligence. Algorithms and APIs are utilized to analyze large amounts of data, deriving valuable insights to drive business decisions.

i. AI: Artificial intelligence tools and technology have made software development more efficient and accessible. Tools that enable non-technical or semi-technical individuals to write software applications are becoming popular, opening up opportunities in the IT sector.

j. Progressive Web Applications (PWA): PWA frameworks provide a fast and efficient environment for web application development. These applications can work offline and on mobile devices, offering high accuracy and fast loading times.

These trending practices, along with others, are reshaping the software development landscape, driving innovation, and improving the efficiency and effectiveness of development processes.

1.4 System Engineering

System engineering encompasses the entire system, including hardware, software, and the system development process. It covers various aspects such as system requirement analysis, system design, system components development, system configuration, system testing, system deployment, and system maintenance and retirement planning.

The primary responsibility of system engineering is to ensure the proper functioning of the developed components in the production environment. It is particularly essential for developing complex systems, as system engineers design the system from inception to its operational state.

System engineers collaborate with multiple teams, including stakeholders, software engineers, testers, developers, and production environment teams, to understand requirements and ensure the quality of the system. Their role is crucial in delivering high-quality software with accuracy in the production environment.

1.4.1 Difference between Software Engineering and System Engineering

System engineering and software engineering are closely related disciplines but have distinct areas of focus and competence. Here are the main differences between the two:

Perspective & Scope:

System engineering takes a broader approach, concentrating on the design, development, and management of complex systems that may comprise hardware, software, firmware, and other components. It considers the interactions and integration of these components within the system's environment. Software engineering, on the other hand, specifically targets the software component within a larger system. It deals with tasks such as coding, testing, and software design, focusing on the development and maintenance of software applications.

1.5 Role of System Analyst

The role of a system analyst is pivotal and holds significant importance in ensuring the overall success of a project. System analysts engage with stakeholders and business clients to gather comprehensive information about the project to be developed. They undertake various tasks in their role, including:

Requirement Gathering: System analysts employ various techniques such as creating questionnaires, conducting workshops, and interviewing stakeholders to collect data about the problem at hand. The process of gathering requirements can be challenging, especially if there are no existing models to refer to, leading to a thorough and sometimes lengthy process.

Requirement Analysis: Once all the necessary information is gathered, system analysts analyze the requirements to gain insights into the problem and devise suitable solutions. The proposed solution must align with the needs and expectations of all stakeholders involved.

Documentation: System analysts are responsible for documenting the proposed solution in official documents such as the Software Requirement Specification (SRS). The SRS document outlines the functional and non-functional requirements, as well as the implementation procedure. It serves as a crucial reference for the development team to understand the project's requirements accurately.

Prepare System Design: Another key responsibility of system analysts is to design the system model. This involves creating data-flow diagrams, use cases, integration points, and other architectural aspects. System analysts ensure that the proposed design meets the requirements of the stakeholders and provides clear guidelines for software development.

Communication: System analysts act as a bridge between stakeholders and the development team, communicating all aspects of the project. They interact with developers, testers, and deployment teams to ensure that the work is aligned with the requirements outlined in the SRS document. Effective communication and documentation are essential to avoid misunderstandings and misinterpretations during the development process.

Recommendations: System analysts propose business solutions to address existing problems and recommend technology-based solutions to enhance business operations. Their recommendations are based on a thorough understanding of the project requirements and stakeholder expectations.

1.6 Summary

First, we established the boundaries of software engineering. We developed two additional, equivalent definitions:

1. The methodical compilation of years of programming expertise along with the discoveries made by researchers towards producing high-quality software economically. It represents the method of engineering used to create software.
2. Software engineering techniques are essential for the development of large software products where a group of engineers work in a team to develop the product. However, most of the principles of software engineering are useful even while developing small programs.

To construct complete systems, computer systems engineering must integrate the development of both software and hardware components. Software engineering falls under the umbrella of computer systems engineering.

1.7 Keywords

- **Software Engineering:** Software engineering refers to the disciplined approach of analyzing system requirements, designing the model, developing, configuring, testing, deploying, and maintaining software applications.
- **System Engineering:** System engineering is a disciplined methodology associated with the entire system, encompassing hardware, software, and the development process. It ensures the delivery of a working system with proper integration of all developed components.

- **System Analyst:** The role of a System Analyst is pivotal in the development process. They interact with stakeholders to gather system requirements. Without proper communication and requirement analysis, the system may fail to meet the desired outcome.
- **Stakeholders:** Stakeholders are individuals or groups directly impacted by the product or the development process. They can include end-users as well as team members involved in various aspects of the product's development.

1.8 Self-Assessment Questions

- Why is software engineering important?
- What are the different kinds of software development practices followed in the IT industry?
- What are the key challenges of software engineering?
- Why is system engineering different from software engineering?
- What is the role of a System Analyst in the IT industry?

1.9 Case Study

Title: Online Retail Platform

Introduction: This case study delves into the challenges encountered by the online retail platform "Shop Now" as it experienced rapid growth. As computer science students, our task is to identify a suitable Software Development Life Cycle (SDLC) process to address these issues while prioritizing a better user experience.

Case Study Background: Initially starting as a small website selling niche products, the online retail platform "Shop Now" expanded rapidly, leading to various challenges such as performance issues, security vulnerabilities, and managing increasing software complexity. Recognizing the

need for a dedicated software engineering approach, the company sought solutions to overcome these challenges.

Your Task: Our objective is to recommend the best software engineering methodology to address the business problems outlined in the case study. This solution should focus on improving user experience while resolving the identified issues.

Questions to be Considered:

1. **System Reliability and Performance:**
    - Software engineering ensures reliability and stability through robust development processes.
    - Measures like performance optimization, load balancing, and caching enhance platform responsiveness and handle high traffic volumes.
2. **User Interface and User Experience (UI/UX):**
    - Software engineering contributes to designing intuitive interfaces and improving accessibility and visual aesthetics.
    - Gathering user feedback and continuous improvement are integral to refining UI/UX aspects.
3. **Security and Data Protection:**
    - Software engineering addresses security vulnerabilities and protects sensitive data through secure coding practices and encryption techniques.
    - Compliance with data protection regulations such as GDPR or CCPA is ensured through best practices and frameworks.
4. **Scalability and Flexibility:**
    - Software engineering enables scalability by implementing architectural designs like microservices and supporting future expansions and integrations.
    - Adapting to changing market trends and customer demands is facilitated by flexible software architecture.
5. **Backend Operations and Integration:**

- Smooth integration of backend systems is ensured through suitable technologies and frameworks.
- Optimizing backend operations streamlines processes and reduces manual effort through automation.

6. **Maintenance and Upgrades:**
   - Software engineering assists in ongoing maintenance, bug fixing, and performance monitoring.
   - Strategies for handling software upgrades and ensuring compatibility with evolving technologies are essential.
7. **Analytics and Insights:**
   - Capturing and analyzing relevant data for business decision-making is facilitated by software engineering techniques.
   - Tools and frameworks are utilized for generating actionable insights and integrating data-driven optimizations.

1.10 References

2 1.9.1 I. Sommerville: Software Engineering, 9th Edition, Pearson Education, 2017.

3 1.9.2 Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.

4 1.9.3 Roger S. Pressman: A Practitioner's Approach, 7th Edition, McGraw Hill Education, 2009.

Unit - 2

Software Development Life Cycle Model

Learning Objectives:

- Explore the different SDLC Models.
- To understand the entry and exit plan of the development model
- Understand the selection criteria of the model.
- To understand the different ways to develop the software.
- To understand the emergence of Agile methodology.

Structure:

Introduction: Software Development Life Cycle Model

Software engineering methodologies are well-structured and sequential sets of software development tasks. SDLC, also known as a software development process model, encompasses several lifecycle models in the software industry according to the system's requirements. These models can be modified according to specific needs, with a primary focus on commonly used SDLC methodologies. Before delving into the discussion, it's essential to highlight some fundamental points of SDLC.

2.1 Need for the Software Development Life Cycle Models

Software Development Life Cycle (SDLC) models are plans that offer structure and direction for the development of software applications. In the software development process, they fulfill several important needs:

a. Approach: SDLC models provide an organized, methodical approach to software development, specifying the order in which various tasks, activities, and deliverables must be completed. This ensures that development teams adhere to a standardized procedure.

b. Analysis of User Needs: SDLC models place a strong emphasis on acquiring and analyzing user needs before beginning development. They assist in ensuring that the finished software product satisfies the requirements and expectations of its target consumers by undertaking a thorough examination.

c. Project Planning and Management: By providing a roadmap for the development process, SDLC models help with project planning and management. Project managers can effectively allocate resources, predict costs, and measure progress, deadlines, and dependencies.

d. Risk Management: SDLC models improve risk management by identifying potential risks early in the development cycle. This allows teams to apply mitigation techniques, deal with problems before they arise, and lessen the impact of risks on the finished product by completing risk assessments at various stages.

e. Quality Control: SDLC models encourage the use of appropriate quality control procedures throughout the entire creation process. They implement strategies such as testing, code reviews, and quality checkpoints at various stages to ensure that the program is thoroughly tested and adheres to necessary quality standards.

f. Communication and Teamwork: SDLC models promote communication and teamwork among stakeholders including business analysts, testers, designers, developers, and other

team members. They provide team members with the same vocabulary and framework, making it easier for them to work together and interact effectively.

g. Documentation: SDLC models emphasize the importance of documentation throughout every stage of development. Explicit documentation of requirements, design choices, code implementation, and testing processes simplifies future maintenance and development of the product.

h. Scalability and Maintainability: SDLC models take scalability and maintainability factors into account throughout the design phase. By including these criteria, they ensure that the program can handle increasing workloads, be responsive to changing requirements, and be easily maintained and developed over time.

2.2 Entry and Existing Criteria of the SDLC Method

A crucial element of the software development life cycle (SDLC) model is the establishment of entry and exit criteria, which outline specific requirements that must be met to qualify a project to progress to the next phase. Let's discuss the entry and exit standards for each stage of a standard SDLC model, such as the Waterfall model:

a. **Requirement Gathering and Analysis:**

Entry requirements include the initiation of the project, understanding of the problem statement, and identification of key stakeholders. Exit criteria encompass approved and documented customer demands, both functional and non-functional requirements, and an approved requirements document.

b. **Designing:**

Entry prerequisites consist of completed requirements documents, functional specifications, and an approved architecture design methodology. Exit criteria include documents related to system design, such as architecture diagrams, database schemas, interface requirements, and design reviews.

c. **Coding or Development:**

Entry requirements entail approved system design documentation, coding standards, and guidelines. Exit criteria consist of code reviews, unit test reports, and developed and tested code modules.

d. **Testing:**

Entry requirements include test plans, test cases, and completion of the coding phase. Exit criteria encompass executed test cases, test results, defect reports, and an approved test summary report.

e**. Configuration and Deployment:**

Entry requirements include completion of the testing phase, approval of the test summary report, and a stable and fully functional software build. Exit criteria include software that has been installed in the intended environment, installation and user manuals, and an approved deployment checklist.

f. Maintenance and Support: Entry requirements comprise software that has been successfully deployed, user approval, and sign-off. Exit criteria include a plan for ongoing support and maintenance, as well as resolution of critical bugs and issues.

It's important to note that these requirements may vary depending on the specific SDLC model and project specifications. Agile approaches, for instance, may have variable entry and exit criteria due to their iterative and incremental nature.

Entrance and exit criteria serve as checkpoints to ensure the proper completion and validation of each phase of the SDLC before proceeding. Throughout the entire development lifecycle, they aid in managing project progress, setting clear goals, and maintaining quality.

.1 Different SDLC Models

There are various software development lifecycle models available, each tailored to specific user requirements and system design approaches. Here, we'll discuss some commonly used ones:

a. Classical Waterfall Model: In the Waterfall model, the core process activities of specification, development, validation, and evolution are segmented into distinct process phases. These phases typically include requirements definition, software design, implementation, testing, and so forth, as illustrated in the figure. b. The foundational model for the software development process was derived from more generic system engineering methods and was first published by Royce in 1970. The concept is depicted in the following figure. Named the 'waterfall model' or life cycle of software, this model describes how the phases flow into one another. The waterfall approach represents a process driven by plans, where all process activities must be scheduled and planned before work commences.

Each process should ultimately produce one or more documents that have been approved or "signed off." It is advisable to wait until the preceding phase is complete before beginning the next.

However, in reality, these steps interconnect and exchange data. Issues with requirements may arise during design, design issues may surface during coding, and so forth. The software development process involves feedback from one phase to the next and is not strictly linear.

The documents generated throughout each phase may need to be adjusted to reflect any changes. Iterations can be costly and require extensive revision due to the expenses associated with creating and approving documents. Consequently, it is common to freeze early phases of development, such as the specification, and proceed to later phases after a limited number of iterations.

Problems may be deferred for later resolution, ignored, or addressed with workarounds in programming. However, this early requirement freezing could potentially hinder the system from performing the desired user actions and lead to poorly structured systems as implementation techniques are used to compensate for design flaws.

The software is utilized throughout the final life cycle phase, operation, and maintenance. The original software requirements may contain errors and omissions that are identified over time. Errors in design and programming may arise, and new functionality may be deemed necessary. Therefore, to ensure continued usefulness, the system must undergo changes. Revisiting earlier process phases may be necessary to implement these modifications, a practice known as software maintenance (Figure 2.1).

a. Iterative Waterfall Model:

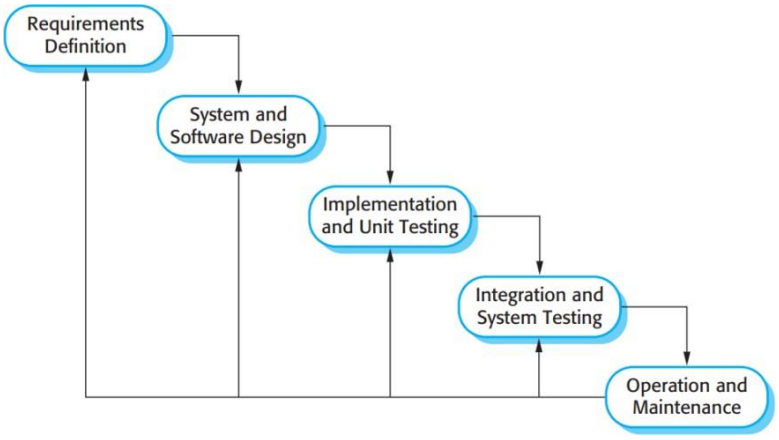

Figure 2.1: Iterative Waterfall Model

In the previous section, we highlighted the challenges of applying the conventional waterfall approach to real-world software development projects due to its perceived unrealistic nature. In response to this, the iterative waterfall model emerges as a modification of the traditional waterfall model, tailored to suit actual software development projects.

The primary distinction between the iterative waterfall model and the traditional waterfall model lies in the incorporation of feedback paths from each stage to its preceding phase. Figure 2 illustrates the feedback pathways introduced in the iterative waterfall model. These feedback pathways allow for the correction of programming faults as they are discovered in later phases, facilitating timely adjustments and improvements. For example, if a design error is identified during the testing phase, the feedback path enables the revision of the design and the incorporation of changes into all subsequent documents.

b. Prototype Model

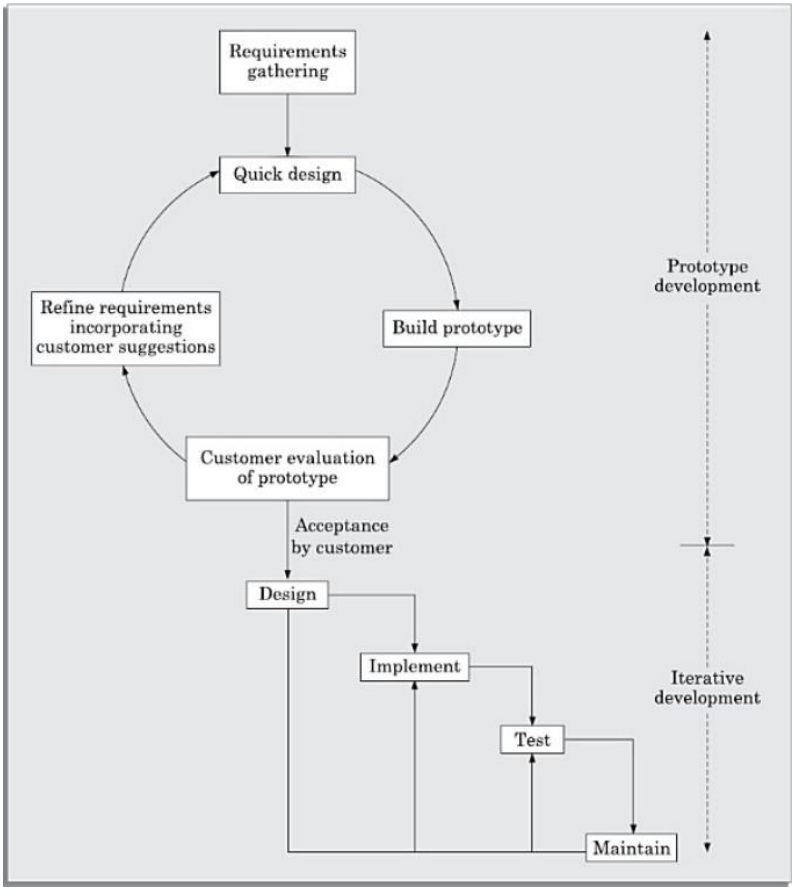

Figure 2.2: Prototype model of the software development process

Another popular lifecycle model is the prototyping model, which can be seen as an extension of the waterfall model. This model advocates for the creation of a functioning system prototype before the actual software development begins. A prototype is a system that is roughly implemented as a demonstration. It may perform inefficiently, with low reliability, or may have functional limitations compared to the final software. Quick methods can be employed to develop a prototype rapidly, such as implementing crude or ineffective functions, like using table look-ups instead of actual computations to achieve desired results. The term "rapid prototyping" is often used when referring to software tools (Figure 2.2).

Constructing a prototype and presenting it for user review helps specify many customer requirements accurately, while technical issues can be addressed by experimenting with the prototype. This approach minimizes the costs associated with additional client requests for modifications and redesigns.

Advantages of the Prototype Model:

The necessity to iterate and refine software solutions due to the difficulty of getting it right the first time is a key argument for developing prototypes. According to Brooks [1975], preparing to discard future software is essential to creating high-quality software. Therefore, the prototype paradigm can be employed when highly optimized and effective software development is required.

Advantages of the Prototyping Approach: This model is most suitable for projects prone to technical and requirements risks, as a built prototype helps in mitigating these risks.

Disadvantages of the Prototype Model:

For projects that are straightforward development tasks without significant risks, the prototyping model may increase development costs. Even for projects at risk, the prototyping model is most effective when risks can be identified upfront before development begins. The prototyping approach is ineffective for risks identified later in the development cycle since it is only built at the beginning of the project. For projects where risks can only be identified once development is underway, the prototyping paradigm would not be suitable.

c. Evolutionary Model:

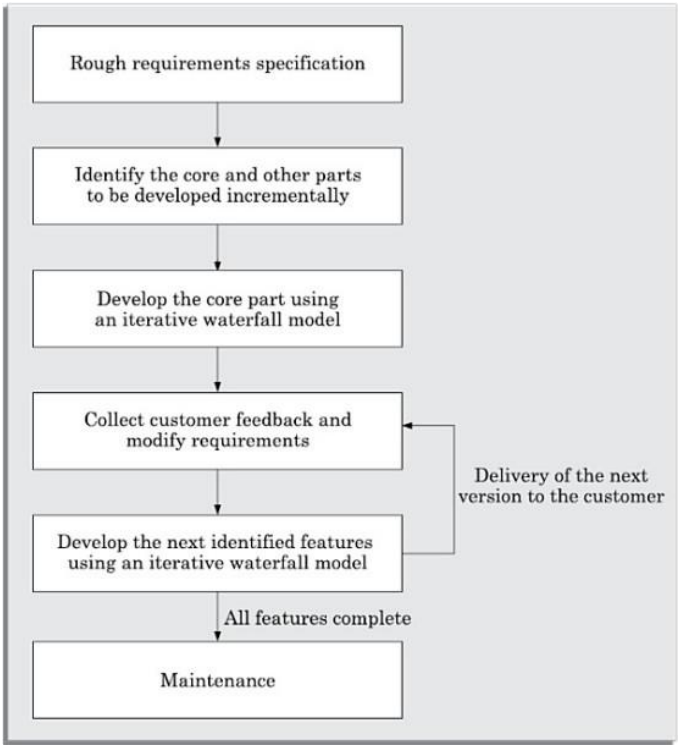

Figure 2.3: Evolutionary Model

Numerous aspects of the evolutionary model resemble those of the incremental model. The program is developed over multiple increments, with each increment implementing an idea or feature deployed at the client site. The software is then enhanced over time with more features until it reaches its final state. The name of the evolutionary lifecycle model reflects its main premise. Initially, complete requirements are generated, and the Software Requirements Specification (SRS) document is created under the incremental development paradigm. In contrast, under the evolutionary model, requirements, plans, estimates, and solutions evolve throughout the iterations rather than being fully specified and fixed in advance before the start of development cycles. This evolution aligns with the pattern of unforeseen feature discovery and alterations that occur throughout the development of new products (Figure 2.3).

The evolutionary software development model is used to develop object-oriented applications where software can be divided into small individual units.

Advantages:

a. Effective elicitation of real customer requirements: In this model, users can test out software that has not yet been fully constructed, allowing for precise elicitation of user requirements through feedback gathered from the distribution of several software versions. As a result, there are considerably fewer change requests once the software is delivered in its entirety.

b. Managing change requests is simple: Since there are no long-term plans in place, addressing change requests is simpler under this paradigm. Consequently, reworks needed as a result of change requests are typically far less frequent than with sequential models.

Disadvantages:

a. It can be challenging to divide a feature into incremental pieces: Breaking up necessary features into numerous pieces for incremental delivery can be challenging for many development projects, especially smaller ones. Additionally, even for larger challenges, it is frequently difficult for a specialist to plan incremental delivery because the features are often so intertwined and dependent on one another.

b. Ad hoc design: Because only the current increment is designed at a given time, the design may become hasty without paying close attention to maintainability and efficiency. The iterative waterfall methodology can produce a superior solution for moderately sized problems and those for which customer needs are explicit.

The V-model, unlike the waterfall approach, varies as steps for verification and validation are performed throughout the development lifecycle. This reduces the likelihood of errors in the work outputs, making it suitable for initiatives involving the creation of highly reliable, safety-critical software. The development phases are represented by the left half of the model, while the right half represents the validation phases.

Advantages:

a. Many testing tasks (such as test case design, test planning, etc.) in the V-model are completed concurrently with development tasks. Consequently, a significant portion of testing operations, such as test case creation and test planning, are finished before the testing phase even begins.

b. Compared to the iterative methodology, this model typically results in a shorter testing period and speedier overall product development.

c. The quality of test cases is typically higher because they are created before schedule pressure increases. Unlike the waterfall paradigm, where testers are only involved during the testing phase, the test team is reasonably busy throughout the development cycle. This results in a more effective use of human resources.

d. The test team is included in the project from the start according to the V-model. As a result, they have a solid understanding of the development artifacts, enabling them to test the program efficiently. In contrast, since no testing activities are conducted before the beginning of the implementation and testing phase, the test team frequently joins the waterfall model late in the development cycle.

Disadvantages:

Because this is the successor of the water fall model. So most of the weaknesses of the waterfall model are also present in the V model.

f. Boehm's Spiral Model:

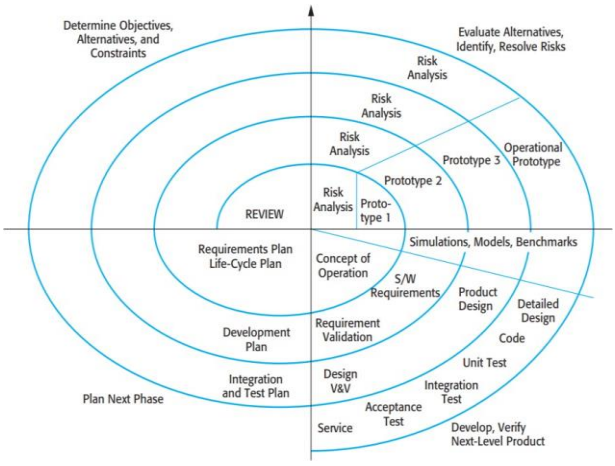

Figure 2.4: Spiral life cycle Model

Change avoidance and change tolerance are integrated in the spiral model.

The spiral's loops are divided into four sectors (Figure 2.4):

- Setting Specific Objectives: The project's specific objectives are determined, and a comprehensive management plan is formulated after identifying constraints on both the process and the product. Project risks are identified, and alternative strategies may be devised depending on these risks.
- Risk Evaluation and Mitigation: A thorough analysis is conducted for each identified project risk. Measures are implemented to mitigate these risks. For example, if there is a risk of incorrect requirements, a prototype system may be created.
- Creation and Approval: A system development model is selected following risk assessment. For instance, if user interface risks are predominant, throwaway prototyping may be the optimal development technique. If safety issues are the primary concern, development based on formal transformations may be preferred, and so on. The waterfall approach may be the optimal development strategy if sub-system integration is the primary identified risk.
- Planning: After reviewing the project, a decision is made whether to proceed with the subsequent spiral loop. Plans for the project's next phase are created if it is chosen to move further.

2.4 Selection Criteria of Software Development Life Cycle Models:

To effectively choose a Software Development Life Cycle (SDLC) technique, several factors and project-specific standards must be considered. Here are some selection criteria to assist in decision-making:

1. Project Specifications: Understand the precise requirements of the project, including its scope, level of difficulty, and expected results. Some projects may require a more linear and predictable approach, while others may benefit from a flexible and iterative approach.
2. Project Scope and Duration: Consider the size and duration of the project. Larger and more complex projects may require structured and rigorous processes, while smaller projects with well-defined requirements may benefit from simpler and faster SDLC techniques.
3. Team Members and Expertise: Assess the qualifications and experience of the development team. Choose an SDLC methodology that aligns with the skills and experience of the team, whether it's traditional waterfall methodologies or agile practices.
4. Adaptive to Change: Evaluate the project's capacity for adapting to changes in requirements, scope, or technology. Agile approaches, with their flexibility and iterative development cycles, may be suitable if changes are anticipated or frequent.
5. Risk Management: Consider the project's tolerance for risk and the importance of completing the product by a given date. SDLC techniques with robust risk evaluation and handling procedures may be necessary if hazards need to be recognized and managed early in the development process.
6. Stakeholder Involvement: Determine the level of end-user or consumer involvement required throughout the development process. Agile methodologies emphasize frequent customer cooperation and input compared to more traditional techniques.
7. Regulation and Compliance Requirements: Take into account any regulations or compliance standards that must be followed. SDLC techniques like the V-Model, which emphasize thorough documentation and formal verification, may be appropriate for projects requiring adherence to stringent regulatory standards.
8. Cost and Timing Limitations: Evaluate the project's budget and timing constraints. Some SDLC methodologies offer simpler and more affordable development processes, while others may require additional resources, time, and money.

9. Organizational Culture: Consider the organization's culture and its readiness to implement certain SDLC techniques. Assess the assets and support available for the implementation and maintenance of the chosen method.

These criteria should be considered while selecting an SDLC method, and the chosen model can be adjusted according to the specific project requirements.

2.5 Agile Methodology

Agile methodologies are based on an iterative process for specifying, creating, and delivering software. They are particularly effective when developing applications in environments where system requirements frequently evolve throughout the development process. The primary aim is to deliver functional software quickly to clients, enabling them to propose new or modified requirements for inclusion in subsequent versions of the system.

By minimizing work that adds questionable long-term value and eliminating documentation that is unlikely to be utilized, agile methodologies aim to streamline the development process and reduce complexity.

Some well-known agile methodologies include:

a. Extreme Programming (Beck, 1999; Beck, 2000)

b. Adaptive Software Development (Highsmith, 2000)

c. DSDM (Stapleton, 1997; Stapleton, 2003)

d. Scrum (Cohn, 2009; Schwaber, 2004; Schwaber and Beedle, 2001)

e. Crystal (Cockburn, 2001; Cockburn, 2004)

f. Feature-Driven Development (Palmer & Felsing, 2002)

These methodologies have demonstrated effectiveness and have been combined with traditional system modeling-based development techniques to give rise to concepts such as agile modeling (Ambler & Jeffries, 2002) and agile iterations of the Rational Unified Process (Larman, 2002).

2.6 Summary

This chapter delves into various Software Development Life Cycle methodologies prevalent in the software development industry.

Adhering to a proper process model has become widely accepted among software development firms for any software development project, as it is recognized as essential for project success.

We have covered the fundamental concepts underlying significant process models. The precise process model used by proficient software development businesses is meticulously documented, typically including:

- Identification of phases in the development cycle.
- A list of tasks to be completed in each phase, along with their respective timelines.
- Criteria for entering and exiting each phase.
- Procedures employed to carry out various activities.

Our focus has been on the selection criteria of SDLC based on project features.

We have also examined the entry and exit points of traditional software development life cycle models.

2.7 Keywords

SDLC: The Software Development Life Cycle is a systematic approach to developing a software system from inception to the final product.

Agile Method: Agile methodology is a disciplined approach to developing business software that accommodates rapid changes in requirements. It encompasses a set of SDLC models that consider changing requirements during the software development process.

Waterfall Model: The Waterfall Model is a traditional software development model comprising five phases. Each phase begins only after the previous one is completed. However, its sequential process can be a drawback as error correction becomes expensive after each phase is finished.

Stakeholders: Stakeholders refer to individuals or groups directly affected by the product or the development process. They can include end-users as well as team members involved in various aspects of product development.

Prototype: A prototype is a mock implementation of the real project, showcasing user requirement implementations through GUI, forms, and reports. While prototyping adds additional costs to the software development process, it helps reduce the overall cost by facilitating early feedback and requirement validation.

2.8 Self-Assessment Questions

- What are the main stages of software development using the waterfall model? Which stage of creating a standard piece of software requires the most work?
- Determine the standards by which a lifecycle model that works for a particular project can be selected. Explain your response using appropriate examples.
- Discuss how the time spent on the various phases is distributed while developing commercial software for an industrial application using the iterative waterfall methodology.
- Explain the V Model in a few lines. Explain why the VSDLC model is typically regarded as being suited for designing safety-critical software.
- Describe the agile software development model in brief. Give two examples of projects—one for which the agile approach would be acceptable and one for which it would not—and explain why.

2.9 Case study

Title: Selection of SDLC Method for an E-Commerce Website Development Project

Introduction: A company is planning to develop an e-commerce website to sell its products online. The website will include product listings, shopping cart functionality, payment gateways, order tracking, and customer support features. The project timeline is six months, and the development team consists of five members, including developers, testers, and designers.

Questions to be considered:

a. What are the project's specific requirements, scope, and expected deliverables?

b. What is the size and duration of the project?

c. What are the capabilities and expertise of the development team?

d. Is there a need for flexibility and adaptability due to potential changes in requirements?

e. What is the project's risk tolerance and criticality of timely delivery?

f. How much customer or end-user involvement is required throughout the development process?

g. Are there any regulatory or compliance requirements that need to be met?

h. What is the organizational culture regarding SDLC adoption and readiness?

i. What are the budget and time constraints for the project?

j. What lessons can be learned from past projects within the organization?

Recommendations:

a. Based on the project requirements, a dynamic and iterative SDLC method like Agile or Scrum would be suitable. These methods allow for incremental development, frequent feedback from stakeholders, and adaptability to changing requirements.

b. Considering the project's relatively small size and six-month timeline, an Agile approach with shorter sprints could ensure timely delivery and quick response to evolving needs.

c. The development team's capabilities and expertise should be assessed. If the team has experience and knowledge in Agile practices, selecting an Agile SDLC method would leverage their strengths and promote collaboration within the team.

d. Due to the nature of e-commerce projects and the potential for evolving requirements in response to market demands, an Agile SDLC method would provide the necessary flexibility to accommodate changes throughout the development process.

e. Risk tolerance should be evaluated. Agile methods allow for early identification and mitigation of risks through iterative cycles, making them suitable for managing risks effectively.

f. Given the customer-centric nature of e-commerce websites, customer involvement and feedback are crucial. Agile methodologies emphasize frequent customer collaboration and can ensure that the end product aligns with customer expectations.

g. Regulatory and compliance requirements, such as data protection or payment processing standards, should be considered. Depending on the level of compliance needed, a hybrid approach combining Agile development with more formal verification methods may be appropriate.

h. The organizational culture should be assessed for its acceptance and readiness to adopt an Agile approach. If the organization has previously embraced Agile practices successfully, it will be easier to implement and sustain an Agile SDLC method.

i. Budget and time constraints should be taken into account. Agile methods can provide better cost control and help deliver incremental value within the given timeline.

j. Lessons learned from past projects should inform the decision-making process. If previous projects have demonstrated successful outcomes with Agile methods, it would be

beneficial to leverage that experience and select an Agile SDLC approach for this e-commerce website development project.

By taking into account these queries and suggestions, the corporation may increase the likelihood of successful project execution and delivery by choosing an SDLC technique that is compatible with the project's needs, team capabilities, and organizational environment.

2.9 References

I. Summerville: Software Engineering, 9th Edition, Pearson Education, 2017.

Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.

Roger S. Pressman: A Practitioner’s Approach, 7th Edition, McGraw Hill Education, 2009.

UNIT 3

Software Requirement Specifications

Learning Objectives:

- To understand the Requirement gathering and analysis phase of SDLC.
- To get insights into Functional system requirements.
- To understand the SRS document structure.
- To understand formal system Specifications.
- To understand the Role of Decision Tables and Decision trees in the development cycles.
- To understand the 4Gl process and algebraic and axiomatic specification system.

Structure:

Introduction

It is widely acknowledged that having a high-quality requirements document is essential for the successful execution of any software development project. A comprehensive requirements specification not only aids in developing a clear understanding of all the features that need to be incorporated into the software but also provides a framework for various tasks that must be completed during later stages of the lifecycle.

The role of documentation is particularly crucial when software is being developed under contract for another organization, such as in an outsourced project. Top of Form Bottom of Form.

3.1 Requirement Gathering and Analysis

3.1.1 Requirement Gathering

Requirement gathering, also known as requirements elicitation, involves collecting requirements from stakeholders, which is the primary objective of this task. While gathering requirements may seem straightforward, it is actually quite challenging to compile all the necessary data from numerous stakeholders and documents, especially without a functioning model of the software.

Requirement gathering activities often commence even before reaching the client site, by examining existing records to gather as much information as possible about the system to be implemented. Analysts typically conduct interviews with end users and customer representatives on-site at the client's location, along with various requirements-gathering tasks such as task analysis, scenario analysis, and form analysis.

Many consumers lack significant computer experience, resulting in very general descriptions of their needs. Skilled analysts engage in discussions with clients to refine these descriptions and make them more complete and universal.

3.1.2 Requirement Analysis

After the requirements gathering phase, analysts analyze the collected requirements to gain a clear understanding of customer requirements and address any flaws. Since stakeholders often have only partial views of the program, it is common for the collected data to contain contradictions, ambiguities, and insufficient information. Hence, any shortcomings in the requirements must be identified and resolved through further discussions with the client.

The primary objective of the requirement analysis task is to clarify the application to be developed and remove any ambiguities, incompleteness, or conflicts from the gathered client requirements. Before beginning the analysis, the analyst should address fundamental inquiries regarding the project's problem, its importance, inputs and outputs, possible steps to address the issue, potential complexities, and data exchange formats.

During requirement analysis, analysts must identify and resolve three basic categories of requirements challenges: anomalies, consistency issues, and incompleteness.

- **Anomalies:** Different interpretations of a single requirement.
- **Consistency Issues:** Conflicts between requirements.
- **Incompleteness:** Client's inability to visualize all necessary features.

By addressing these challenges, analysts ensure that the requirements are clear, consistent, and complete, laying a solid foundation for the subsequent stages of software development. Bottom of Form.

3.2 Functional Requirement

Functional requirements refer to the services provided by the developed system, detailing how the system will behave in specific situations and respond to particular inputs. They articulate what the system should accomplish. These requirements are influenced by factors such as the type of software being developed, the target audience, and the organization's writing style.

Functional requirements are typically expressed in an abstract manner that system users can understand when they are presented as user requirements. However, more detailed functional system requirements offer a comprehensive description of the system's operations, inputs, outputs, exceptions, and so forth.

Functional system requirements can vary from broad needs outlining what the system should achieve to more detailed requirements that consider regional work practices or existing systems within an organization.

3.3 Structure of SRS Document

The Software Requirements Specification (SRS) document holds significant importance in the software development lifecycle. It is considered one of the most crucial documents, albeit challenging to create, due to its diverse user base and comprehensive nature.

The SRS document serves several key users, including:

a. **Users, Clients, and Marketing Staff:** These stakeholders rely on the SRS document to understand the functionality and features of the software being developed. It provides them with insights into what the software will be capable of and helps manage their expectations.

b. **Software Engineers:** For software engineers, the SRS document serves as a blueprint for development. It outlines the requirements and specifications that need to be implemented, guiding the development process from start to finish.

c. **Testing Team:** The testing team uses the SRS document as a reference to create test cases and plans. It helps ensure that the software meets the specified requirements and functions correctly under various scenarios.

d. **Document Writers:** Document writers refer to the SRS document to create user manuals, technical documentation, and other supporting materials. It provides them with detailed information about the software's features and functionalities.

e. **Project Manager:** The project manager relies on the SRS document to track progress, manage resources, and ensure that the development team stays aligned with the project objectives. It serves as a baseline for project planning and execution.

f. **Maintenance Specialists:** After the software is deployed, maintenance specialists use the SRS document to understand the software's architecture, functionality, and requirements. It helps them identify and address any issues or updates needed during the maintenance phase.

Overall, the SRS document plays a crucial role in facilitating communication, collaboration, and understanding among various stakeholders involved in the software development process. It serves as a comprehensive reference guide that ensures everyone involved has a clear understanding of the software's requirements and objectives.

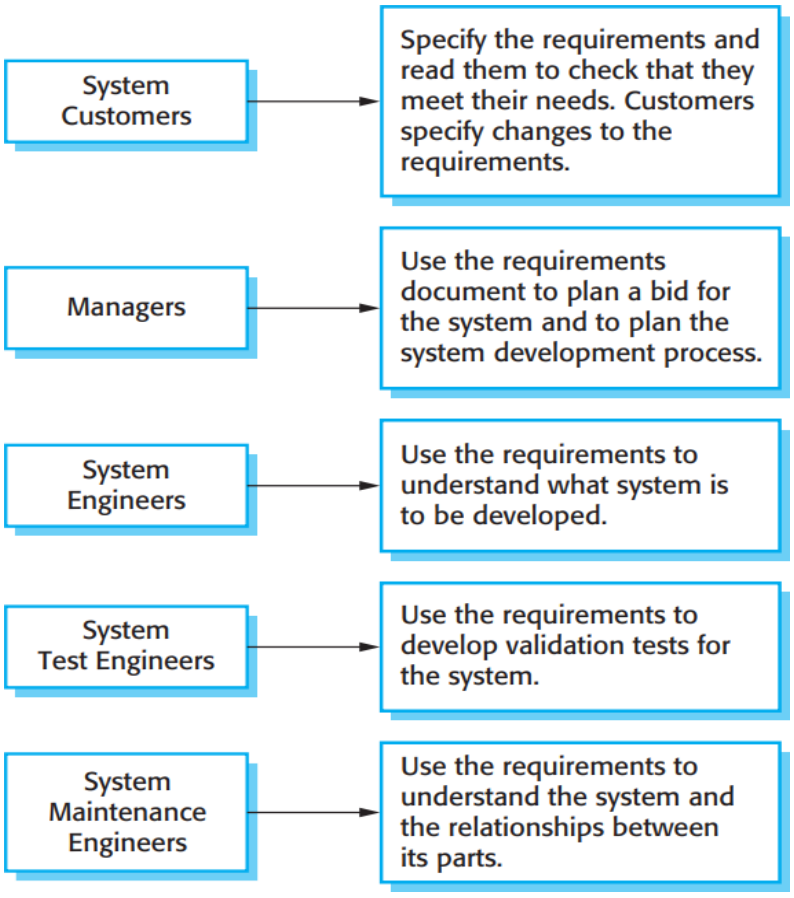

Figure 3.1: User of SRS document

The Software Requirements Specification (SRS) document holds significant importance not only during the software development process but also in various other aspects, including legal matters and project management. Here's how the SRS document is utilized and the characteristics of a well-written SRS document (Figure 3.1):

Usage of SRS document:

a. **Size Analysis:** Project managers often analyze the SRS document to determine the size and complexity of the software project, which helps in resource allocation and project planning.

b. **Early Consideration of Needs:** Before design and development commence, creating the SRS document forces stakeholders to carefully consider and document each requirement. This reduces the need for later modifications, retesting, and rework.

c. **Error Identification:** Early in the development cycle, errors, misconceptions, and inconsistencies can be identified by carefully reviewing the SRS document, allowing for timely corrections.

d. **Contractual Agreement:** The SRS document serves as a contract between the clients and the developers, outlining the agreed-upon requirements and specifications for the software project.

e. **Standard for Compliance:** It establishes a standard against which the compliance of the software developed can be evaluated, ensuring that it meets the specified requirements.

f. **Test Strategy Development:** Test engineers utilize the SRS document to develop the test strategy, defining the scope and approach for testing the software.

g. **Future Enhancements:** Future enhancements and modifications to the software are typically made using the SRS document as a reference, ensuring consistency and alignment with the original requirements.

2. Characteristics of a well-written SRS document:

a. **Verifiability:** Every requirement listed in the SRS document should be verifiable, meaning that it should be possible to determine whether the software meets the specified requirement.

b. **Handling of Undesirable Occurrences:** The SRS document should cover the system's responses to various undesirable occurrences and extraordinary circumstances that may occur during its operation.

c. **Ease of Modification:** A well-written SRS document should be simple to modify. It should be organized in a clear and structured manner, making it easy to comprehend and edit as needed.

d. **Traceability:** Each specified requirement should be traceable to the design components that implement it, and vice versa, ensuring that each requirement is addressed appropriately in the design and implementation phases.

e. **Focus on External Behavior:** The document should describe the system's externally visible behavior and functionality, avoiding discussions about internal implementation details.

f. **Conciseness and Clarity:** The SRS document should be concise, yet clear, consistent, and comprehensive, providing all necessary information without unnecessary verbosity or ambiguity.

Design of the SRS document

In the design of the Software Requirements Specification (SRS) document, it is crucial to categorize and organize requirements properly into sections following the guidelines provided by IEEE830. The document should outline the following significant customer requirement categories:

1. Functional Requirements
2. Non-functional Requirements, which encompass:
    a. Implementation and design limitations
    b. Other non-functional criteria
    c. Need for external interfaces
3. Implementation objectives

3.4 Decision Trees and Decision Tables

Decision trees and decision tables serve as fundamental tools for analyzing and representing complex processing logic. Once the decision-making logic is encapsulated in the form of trees or tables, test cases for validating this logic can be generated automatically. However, it's important to acknowledge that decision trees and decision tables have broader applications beyond just defining intricate processing logic in an SRS document.

For instance, decision trees and decision tables find applications in information theory and switching theory, showcasing their versatility and usefulness beyond software development contexts.

Decision Trees

A decision tree illustrates the logic behind decision-making and the subsequent actions to be taken. On the other hand, decision tables outline the parameters to be tested and the choices to be made based on the outcomes of decision-making logic, as well as the sequence in which decision-making occurs. In a decision tree, the edges represent conditions, while the leaf nodes indicate actions to be taken based on the results of testing those conditions (Figure 3.2).

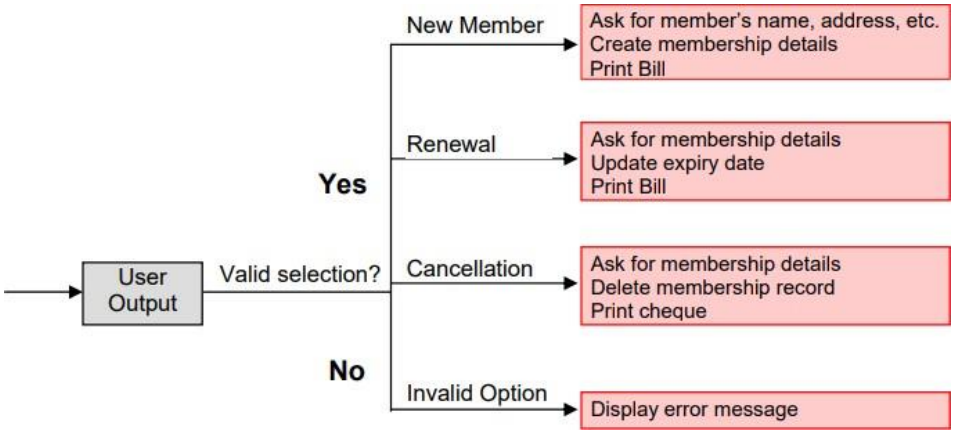

Figure 3.2: Decision Tree for the Library management system

Decision Table

A decision table presents the decision logic and the corresponding actions in a tabular or matrix format. The conditions or factors to be evaluated are listed in the upper rows of the table, while the actions to be taken after the evaluation succeeds are listed in the lower rows. Each column in the table represents a rule, indicating that a particular action is taken if a specific set of conditions is true. Below is an example of a decision-making table for the Library Management System (Figure 3.3):

Conditions				
Valid selection	No	Yes	Yes	Yes
New member	-	Yes	No	No
Renewal	-	No	Yes	No
Cancellation	-	No	No	Yes
Actions				
Display error message	x	-	-	-
Ask member's details	-	x	-	-
Build customer record	-	x	-	-
Generate bill	-	x	x	-
Ask member's name & membership number	-	-	x	x
Update expiry date	-	-	x	-
Print cheque	-	-	-	x
Delete record	-	-	-	x

Figure 3.3: Decision Table for the Library Management System

3.4.1 Differences between the Decision tree and Decision table

a. Decision Trees are more easily comprehensible compared to decision tables when dealing with a small number of conditions.

b. In cases where multilevel decision-making is required, decision trees offer greater benefits. They can represent hierarchical decision-making more naturally than decision tables, which are limited to depicting a single decision for choosing the best course of action.

c. However, as the number of conditions and actions increases, decision trees can become exceedingly complex and difficult to understand. In such scenarios, it may be more preferable to use decision tables, especially when dealing with a high number of decisions, as decision tables can provide a more concise representation.

3.5 Formal Requirement Specifications

- A formal method is a mathematical approach used to specify hardware and/or software systems, assess whether a specification can be implemented, verify that an implementation adheres to its specification, and demonstrate system attributes without executing the system.
- The mathematical foundation of a formal method is established through its specification language. A formal specification language comprises two sets: the syntactic domain and the semantic domain, along with a satisfaction relation between them. The syntactic domain refers to the structure of the specifications, while the semantic domain denotes their intended meaning. The satisfaction relation indicates whether a given specification satisfies a system model.
- In formal methods, if the satisfaction relation holds for a given specification and system model, then syn is considered the specification of sem, and sem is deemed to be the realization of syn.

3.6 Axiomatic Specification

First-order logic finds application in axiomatic system specification to articulate the pre- and post-conditions, delineating the operations of the system through axioms. Pre-conditions denote the prerequisites that must be fulfilled before invoking an operation. Essentially, pre-conditions encapsulate the requirements concerning a function's input parameters.

On the other hand, post-conditions outline the conditions that must be satisfied upon the execution of a function. They serve as constraints on the outcomes generated by the function execution to be deemed successful.

3.7 Algebraic Specification

The algebraic specification technique is utilized to define an object class or type based on the relationships among the operations defined on that type. This approach gained prominence through Guttag's work in the 1980s, particularly in the definition of abstract data types. Over time, several algebraic specification notations have emerged, including those rooted in languages like OBJ and Larch.

In essence, algebraic specifications depict a system as a heterogeneous algebra, where a heterogeneous algebra comprises diverse sets for which various operations are specified.

3.8 Executable or 4GL Process

When a system's specification is clearly defined or documented using a programming language, it becomes feasible to execute the specification immediately without the need to develop implementation code. However, executable specifications are typically slow and inefficient. 4GLs (4th Generation Languages) exemplify executable specification languages. These languages have achieved success because they establish and transfer a large granularity commonality to program code across various data processing applications.

The effectiveness of 4GLs lies in software reuse, where common abstractions are identified and parameterized. Extensive testing has shown that rewriting 4GL programs in 3GLs (3rd Generation Languages) results in up to 50% less memory usage and a tenfold reduction in program execution time.

3.9 Summary

This chapter is dedicated to the Requirement Gathering and Analysis phase of the SDLC.

Prior to commencing the design activity, it is crucial to dedicate significant time and effort to produce a high-quality SRS document. Any inaccuracies in the specification can prove to be costly as they impact all subsequent phases of development.

The Requirements Analysis and Specification phase is divided into two parts: requirements collection and analysis, and requirements specification. The objective of requirements analysis is to thoroughly comprehend the exact user needs and to identify and resolve any inconsistencies, anomalies, or gaps in these requirements.

The requirements are systematically organized into an SRS document during the requirements specification activity.

There are numerous advantages to formalizing the requirements. However, one fundamental drawback of formal specification techniques is their difficulty in application. Nevertheless, with the development of appropriate frontends, formal approaches may become more user-friendly in the future.

To illustrate some of the challenges associated with formal specification, axiomatic and algebraic procedures were examined as examples of formal specification methodologies.

3.10 Keywords

- SRS: The Software Requirements Specification (SRS) document stands out as one of the most crucial and challenging documents to produce in the software development life cycle. Its complexity arises from the diverse range of users it aims to serve.
- Formal Requirement Specification: Formal specifications add rigor to the development process. Creating a rigorous specification is often more significant than the formal definition itself. A rigorous specification clarifies various aspects of system behavior that may not be apparent in an informal specification.
- Functional Requirement: Functional requirements encompass a set of high-level demands, each of which involves receiving input from the user and producing output in return. Additionally, each high-level requirement may consist of multiple functions.
- Decision Tree: A decision tree illustrates the processing logic involved in decision-making and the subsequent actions taken. The branches of a decision tree represent conditions, while the leaf nodes depict the actions to be taken based on the outcomes of the condition testing.
- Decision Table: A decision table presents sophisticated processing logic in a tabular or matrix format. The criteria to be assessed are listed in the table's upper rows, while the actions to be executed when the associated conditions are met are specified in the lower rows. Each column in the table represents a rule, indicating that if a condition is met, the corresponding action should be taken.
- Algebraic Specification: Algebraic specification defines an object class or type in terms of the relationships between the operations defined on that type. Initially popularized by Guttag, algebraic specifications express abstract data types using various notations, including those based on the OBJ and Larch languages.

- Axiomatic Specification: Axiomatic specification formulates the specification in terms of pre- and post-conditions. Pre-conditions must be satisfied before the execution of a system, while post-conditions must be met after the system's execution.

3.11 Self-Assessment Questions

- What types of issues related to requirements does an analyst typically anticipate and address before initiating the creation of the SRS document? Please provide an example of each.
- How does Requirement Gathering differ from Requirement Analysis?
- What are the different categories of system requirements, and what distinguishes Functional from Non-functional requirements?
- Outline five desirable qualities of a well-crafted Software Requirements Specification (SRS) document.
- What are the objectives of the requirements analysis and specification phase, and how are these processes conducted and by whom?

3.12 Case study

Case Study: Requirement Gathering and Requirement Analysis

Company Overview: XYZ Technologies is a software development enterprise specializing in crafting tailored business applications for clients across diverse sectors. Recently, they embarked on a project from a healthcare institution to construct a novel electronic medical records system.

Objective: The aim of this case study is to scrutinize the process of requirement gathering and requirement analysis for the electronic medical records system project.

Requirement Gathering:

a) What methods were employed for requirement gathering?

b) Did all stakeholders actively participate in the process?

c) Was the documentation of requirements conducted efficiently?

Recommendations:

a. Employ a blend of interviews, surveys, and workshops for requirement gathering to ensure a comprehensive grasp of user needs.

b. Engage key stakeholders like doctors, nurses, administrators, and IT personnel in requirement gathering to encompass diverse viewpoints.

c. Utilize collaborative tools and techniques such as prototyping and storyboarding to visualize and validate requirements.

d. Maintain clear and concise documentation of all requirements, encompassing both functional and non-functional aspects, to serve as a reference throughout the project.

Requirement Analysis:

a) How were the gathered requirements analyzed?

b) Were any conflicts or ambiguities detected in the requirements?

c) Were the requirements prioritized based on their significance?

Recommendations:

a. Implement techniques like requirement validation, verification, and traceability to ensure clarity, completeness, and consistency in all requirements.

b. Conduct requirement reviews and walkthroughs with stakeholders to pinpoint any conflicts, ambiguities, or gaps in the requirements.

c. Assign priorities to requirements based on their criticality, impact on system functionality, and alignment with project objectives.

d. Develop a requirement traceability matrix to establish a transparent linkage between requirements, design, development, and testing phases.

Questions to be considered:

a) How did the company ensure the comprehensive capture of all requirements?

b) Were there any encountered difficulties throughout the requirement gathering and analysis phases?

c) How were modifications in requirements handled throughout the project's progression?

Recommendations:

a) Develop a strong communication framework with stakeholders to facilitate continuous collaboration and input.

b) Organize routine requirement review meetings to adapt to alterations and attend to emerging requirements.

c) Introduce a change management protocol to assess, prioritize, and integrate new requirements while effectively managing project scope and timelines.

Conclusion:

Requirement gathering and analysis represent pivotal stages in any software development endeavor. XYZ Technologies should prioritize the involvement of all stakeholders, utilize effective techniques, and uphold clear documentation to ensure the accuracy and comprehensiveness of requirements. By confronting challenges and integrating recommended strategies, the company can significantly enhance the success of the electronic medical records system project.

References:

- I. Sommerville: Software Engineering, 9th Edition, Pearson Education, 2017.
- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.
- Roger S. Pressman: A Practitioner's Approach, 7th Edition, McGraw Hill Education, 2009.

Unit 4

Software Design

Learning Objectives:

- To ascertain the actions involved in software development.
- Determine the goods that need to be created in the first and detailed design stages.
- Recognize the main differences between activities involving analysis and design.
- To identify the essential components developed during the software design stage.
- Describe the essential qualities that make a software design good.
- To identify a design document's essential aspects for comprehension.

Structure:

Introduction

The design document is produced during the software design phase using the client requirements listed in the SRS document. The SRS document is converted into the design document during the design phase's activities, which are referred to as the design process. It is difficult to create an effective software design in a single step, It requires many iterations and a series of processes. The design procedure can be divided into two key categories:

a. preliminary(or high-level) design
b. And detailed design.

4.1 Types of Design Activity

Two design tasks, high-level and detailed design, have quite different meanings and purviews based on the technique used. High level design involves determining several modules, as well as the control linkages and interface definitions between them. The result of the high-level design is referred to as the software architecture or program structure. A high-level design has been represented using a wide variety of notations. A structure chart, which resembles a tree, is a common technique to describe the control structure in a high-level architecture. However, other notations, like the Warnier-Orr [1977, 1981] diagram or the Jackson diagram [1975], can also be utilized. The various modules' data structures and algorithms are designed during the detailed design phase. The module-specification document is the standard name for the product of the detailed design phase.

4.1.1 Difference between Analysis and Design

The analysis's conclusions are broad and don't account for platform- or implementation-specific issues taking into consideration. Typically, a graphical formalism is used to document the analytical model. In the function-oriented method we will examine, data flow diagrams (DFDs) will be used to describe the analytical model and a structure chart will be used to document the design. However, for the object-oriented approach, the unified modeling language (UML) will be used to define both the design model and the analysis model. It would normally be extremely difficult to construct the analytical model using a computer language.
After a number of modifications, the analysis model yields the design model. The design model, in contrast to the analysis model, takes into account a number of choices made about the system's

implementation. The design model needs to be accurate enough to be implemented with ease using a computer language.

4.1.2 Outcome of the Design Phase

Items created during the software development phase. The components that follow have to be designed during the design process for a concept to be executed easily in a conventional programming language.

a. Various modules are required to realize the design solution.

b. The detected modules' control relationships. The call relationship or invocation relationship between modules is another name for it.
c. Inter action between modules. The interface between modules identifies the particular data items sent between them.
d. Individual module data structures.
e. Algorithms required for each module's implementation. To visualize the system being built and anticipate every feature that will be required.

4.2 Important Characteristics of the Design Document

Creating a precise description of a strong software architecture that works in a variety of problem areas is challenging. The idea of a "good" software design might change depending on the application being developed. Nonetheless, the majority of scholars and software engineers concur on a few essential qualities that every effective software design for widespread applications should have. The following are these traits:

a. **Correctness:** All of the features listed in the SRS document should be effectively implemented in a well-designed system.
b. **Understandability:** A good design is simple to grasp.
c. **Effectiveness:** It should be effective.
d. **Changeability:** It should be simple to change.

4.3 Cohesion

The majority of researchers and engineers agree that successful software design requires a clean decomposition of the problem into modules and a clear organization of these modules within a

hierarchy. High cohesion and low coupling are essential properties of effective module decomposition. Cohesion measures a module's functional strength, with a functionally independent module exhibiting high cohesion and minimal connectivity with other modules. Functional independence is defined as a coherent module performing a single task or function, interacting with other modules as little as possible. Different types of cohesion are illustrated in the figure below (Figure 4.1).

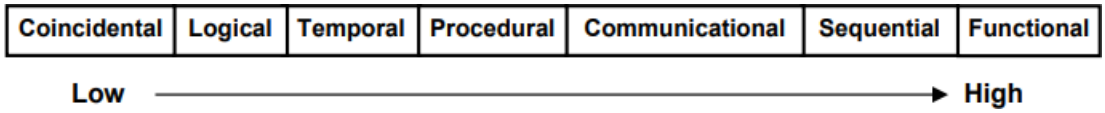

Figure 4.1: Types of Cohesion

Coincidental Cohesiveness: Coincidental cohesiveness refers to a module that performs a series of unrelated tasks with only the slightest connections between them. This type of module comprises a random assortment of operations, often grouped together without any strategic planning or consideration. For example, in a transaction processing system (TPS), a module might arbitrarily combine methods like get-input, print-error, and summarize-members. This grouping has no relevance to the actual structure of the problem being addressed.

Logical Cohesion: A module is considered to have logical cohesion if all of its components perform similar functions, such as error handling, data input, or data output. For example, a module that contains a set of print functions to generate various output reports demonstrates logical cohesion.

Temporal Cohesiveness: A module is considered to have temporal cohesion if it includes functions that are related by the requirement to be executed simultaneously. For instance, the set of functions responsible for the initialization, start-up, and shutdown of a process demonstrates temporal cohesion.

Procedural Cohesiveness: A module is said to have procedural cohesion if its set of functions are all part of a specific sequence of steps within an algorithm that must be performed in a precise order to achieve a particular goal, such as decoding a message.

Communicational Cohesion: A module has communicational cohesion if all of its functions refer to or update the same data structure, such as a set of functions designed to operate on an array or a stack.

Sequential Cohesiveness: When the elements of a module make up a series and the output from

one element is fed into the next, the module is said to have sequential cohesiveness. The get-input, validate-input, and sort-input methods, for instance, are combined into one module in a TPS.

Functional coherence is achieved when various components within a module collaborate to accomplish a single objective. For instance, a module that incorporates all necessary functionalities for handling employees' payroll exemplifies functional coherence. If a module exhibits functional cohesiveness, it can be succinctly described in a single, comprehensive sentence.

4.4 Coupling

Coupling between two modules measures their level of interdependence or interaction. A functionally independent module exhibits high cohesion and minimal interaction with other modules. When two modules exchange a substantial amount of data, their interdependence increases. The complexity of their interfaces determines the degree of their connection. Although no methodologies exist for precisely and statistically estimating the coupling between two modules, categorizing the different types of coupling helps assess their degree of interdependence. Each of the modules can be coupled in any of five different manners (Figure 4.2).

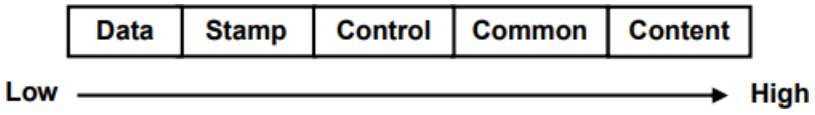

Figure 4.2: Types of Coupling

Data coupling occurs when two modules communicate through a parameter. An example is a simple data item, such as an integer, a float, or a character, passed between two components. This data item should be used to solve a problem, not for control purposes.

Stamp coupling happens when two modules interact through a composite data item, such as a record in Pascal or a structure in C.

Control coupling occurs when data from one module influences the sequence of instructions executed in another module. For instance, when a flag set in one module is used to control the execution flow in another, this represents an example of control coupling.

Common coupling occurs when two modules interact by accessing the same global data.

Content coupling occurs when two modules share code, such as when a segment of one module is directly accessed from another.

4.5 Layered Modular Design Approach

When selecting the best design from a pool of viable options, it's crucial to choose one that is easy to understand. A design that is straightforward to grasp tends to be simpler to implement, test, debug, and maintain, ultimately reducing lifecycle costs. A layered and modular design approach is particularly beneficial as it enhances comprehension and manageability.

4.5.1 Modularity

A Modular design is an essential aspect of effective problem-solving, where a complex issue is divided into smaller, manageable modules with minimal connections between them. This approach simplifies the divide-and-conquer strategy, allowing each module to be understood independently, especially when interactions between modules are either nonexistent or minimal. The design's complexity is thus significantly reduced. Determining the modularity of a design solution is challenging without measurable criteria, but we can assess it based on the cohesion within individual modules and the coupling between them. A design solution is considered highly modular if its modules exhibit strong internal cohesion and minimal inter-module dependencies. This effective decomposition of tasks relies on software designs that feature good cohesion and low coupling between components (Figure 4.3).

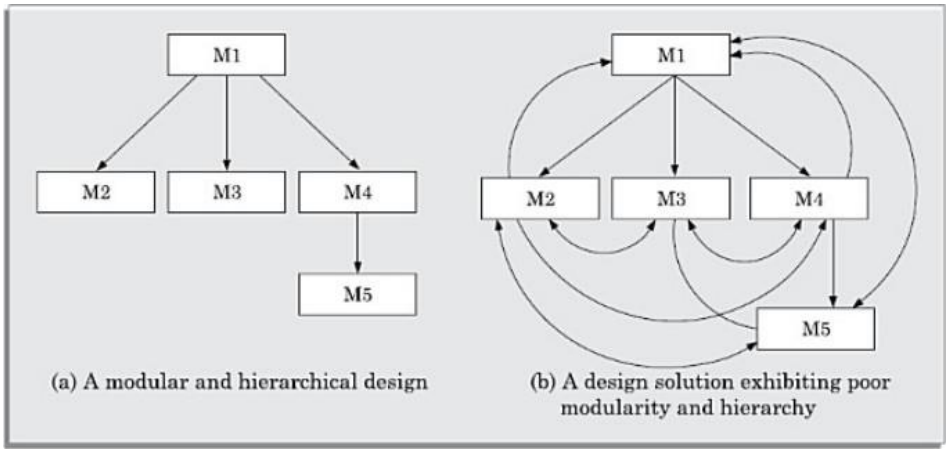

Figure 4.3: Two design solutions for the same problem

As we can say, the 1st design solution is easy to understand as compared to another one.

4.5.2 Layered Design Solution

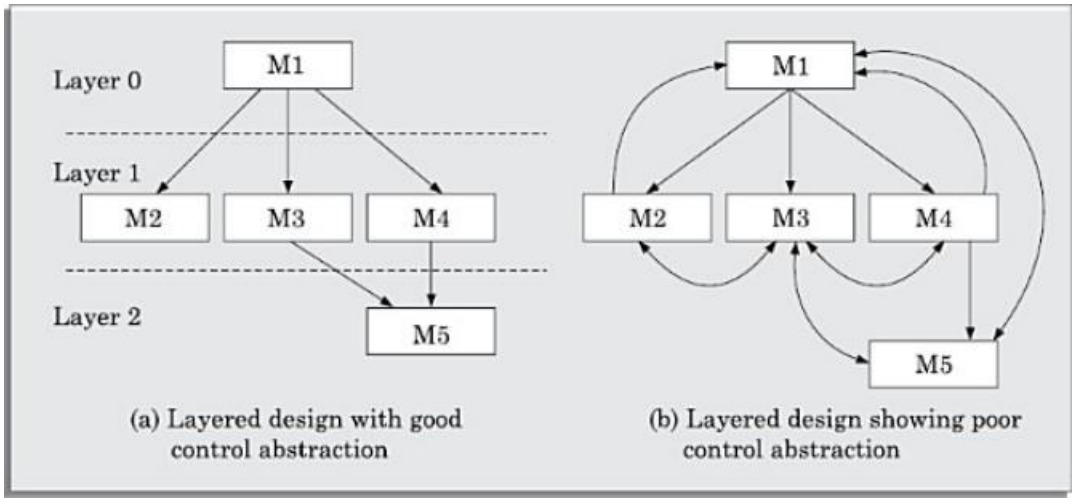

Figure 4.4: Example of good and bad layer abstraction

A layered design is one in which a tree-like diagram with distinct layering is produced when the call relations between various modules were graphically depicted. The modules are placed in a hierarchy of layers in layered design solutions. Only the modules in the layer directly below another module can be called upon to perform certain actions. The higher layer modules can be compared to managers who instruct (invoke) the lower layer modules to complete specific duties. A layered design provides control abstraction because modules in a lower layer do not need to understand or interact directly with modules in higher layers. This simplification allows for easier comprehension of how each module works by only considering the modules in the immediately lower layer. When an error occurs in a module, the potential issues can often be traced to a small subset of modules, making the debugging process more straightforward (Figure 4.4).

4.6 The Important Terminology of Layered Architecture

a. **Superordinate:** A module that manages or controls another module is considered superordinate in a control hierarchy.

b. **Subordinate:** A module that is managed or controlled by another module is regarded as subordinate.

c. **Visibility:** Module B is deemed visible to module A if A directly invokes B.

d. **Control Abstraction:** In a layered design, a module should only invoke the functionalities of modules that are directly beneath it in the hierarchy.

e. **Depth and Width:** The depth and width of a control hierarchy provide insights into the hierarchy's overall span of control and the number of layers it includes, respectively.

f. **Fan-out:** The fan-out of a module refers to the total number of other modules it directly controls.

g. **Fan-in:** Fan-in indicates the number of modules that directly invoke a specific module. High fan-in is typically indicative of good design, as it suggests effective code reuse. In the example provided, module M1 has a fan-in of 0, module M2 has a fan-in of 1, and module M5 has a fan-in of 2.

For example, in Figure 4, the fan-out of module M1 is 3. A design with very high fan-out numbers is generally not considered optimal, as it often indicates that the module lacks cohesion. A module with a significant fan-out (more than 7) is likely to implement multiple distinct functions rather than a unified function.

4.7 Outcome of the Design Module

For a design to be effectively implemented in a typical programming language, several components must be developed:

a. The design requires the creation of numerous modules to realize the solution.

b. The design specifies the connections that link the control structures of these modules.

c. These connections between modules are also referred to as call connections or invocation connections.

d. Interaction between multiple modules needs to be clearly defined.

e. The precise data items exchanged between modules are specified through their interfaces.

f. The data architectures of the various modules must be designed.

g. The algorithms required for the implementation of each module should be developed separately.

4.8 Summary

- This chapter focuses on the Design Phase of the SDLC method.
- Software design typically occurs in two stages: high-level design and detailed design.
- During high-level design, the system's essential parts (modules) and their relationships are determined.
- The data structure and algorithms are identified during detailed design. We emphasized the importance of design clarity as a crucial criterion for determining design quality.
- We defined design clarity as the effective application of decomposition and abstraction concepts. • Later, we classified these in terms of cohesiveness, coupling, layering, control abstraction, fan-in, fan-out, and so on.

4.9 Keywords

- Cohesion: Documenting the Software Requirements Specification (SRS) is often one of the most important and challenging tasks in the software development life cycle. The diverse range of users who rely on the SRS contributes to this complexity.
- Coupling: Formal specifications bring greater rigor to the process. Developing a rigorous specification often proves more important than the formal definition itself, as it clarifies system behaviors that may not be obvious from an informal specification.
- Fan-out: A module's fan-out measures the total number of other modules it directly controls.
- Fan-in: Fan-in refers to the number of modules that directly invoke a specific module.
- Modularity: This principle involves breaking down the problem into smaller, manageable components and implementing them separately. Modularity facilitates high cohesion and low coupling, enhancing the overall design quality.

4.10 Self-Assessment Questions

1 Do you agree with the statement, "Higher development and maintenance costs would come from a poorly understood design solution?" Explain why.

2 In the context of software design, what do the terms coupling and cohesion mean? How do these concepts help in producing an effective system design?

3 How does modular design work? How can you tell if a particular design is modular or not?

4 What are the qualities of good software design?

5 Is it true that when you improve the cohesion of your design, the coupling in the design decreases? Justify your answer using appropriate examples.

4.11 Case Study

Case Study: Importance of the Design Phase in SDLC

Introduction: ABC Solutions is an IT consulting firm that provides custom software development services to various clients. The company has recently completed a project for a retail client to develop an e-commerce platform.

Objective: The objective of this case study is to highlight the importance of the design phase in the Software Development Life Cycle (SDLC) and its impact on project success.

Importance of the Design Phase:

a) Why is the design phase crucial in SDLC?

b) What are the key deliverables of the design phase?

c) How does the design phase influence the overall project outcome?

Questions to Consider in the Design Phase:

a) What are the user requirements, and how can they be translated into a functional design?

b) What architectural and technical considerations should be addressed during the design phase?

c) How can potential risks and constraints be identified and mitigated through design decisions?

Recommendations:

1 Engage in user interviews, surveys, and usability tests to collect comprehensive requirements and integrate them into the design.

2 During the design phase, take into account factors such as scalability, security, performance, and compatibility to prevent potential issues later on.

3 Conduct a detailed risk analysis and incorporate suitable risk mitigation strategies into the design.

Project Success:

a) How does a well-designed system contribute to project success?

b) What are the risks associated with neglecting the design phase?

c) How can the design phase impact project timelines and budgets?

Conclusion: The design phase is crucial to the success of a software development project. ABC Solutions should recognize its significance, address key issues during this phase, and use the provided recommendations to create a well-designed, high-quality solution. By prioritizing the design phase and implementing effective design practices, the company can improve project outcomes, reduce risks, and deliver superior software solutions to its clients.

4.12 References

- I. Summerville: Software Engineering9thEdition, Pearson Education, 2017.
- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.
- RogerS. Pressman: A Practitioner's Approach, 7thEdition, McGraw Hill Education, 2009

Unit 5

Software Structure

Learning Objectives:

- To determine the purpose of the structured design.
- To describe the purpose of a structural chart.
- To explain the differences between a flow chart and a structure chart.
- To give scenarios of the actions taken during the transform analysis process.
- To describe the meaning of transaction analysis.
- To describe what DFD is.

Structure:

Introduction

Contemporary software development methodologies like function-oriented design and object-oriented design diverge significantly from each other. However, despite their distinct approaches, they often complement each other more effectively when used together. Object-oriented methodology, although relatively newer and still evolving, is gaining traction for developing large-scale programs due to its numerous advantages. Conversely, function-oriented designing, with its established principles and methodologies, boasts a considerable following in the field.

5.1 Function-Oriented Design

The key features of a conventional function-oriented design approach include:

The system is perceived as an entity responsible for executing a diverse range of tasks. Each function originates from this overarching view of the system and is gradually elaborated into more intricate functions. For example, consider the function "create-new library-member," which entails generating a record for a new member, assigning a unique membership number, and generating a bill for their membership fee. Sub-functions like creating a membership record, assigning a membership number, and printing a bill can be incorporated into this function. Furthermore, these functions can be further divided into more specific sub-functions.

The system state is centralized and shared among various functions. For instance, data such as member records are accessible for reference and updating by numerous functions, including adding new members, deleting existing members, and editing member records.

5.1.1 Types of Function-Oriented Design

a) Structured design by Constantine and Yourdon (1979)
b) Jackson’s structured design by Jackson (1975)
c) Warnier-Orr methodology (1977, 1981)
d) Step-wise refinement by Wirth (1971)
e) Hatley and Pirbhai’s Methodology (1987)

5.2 Structure Analysis

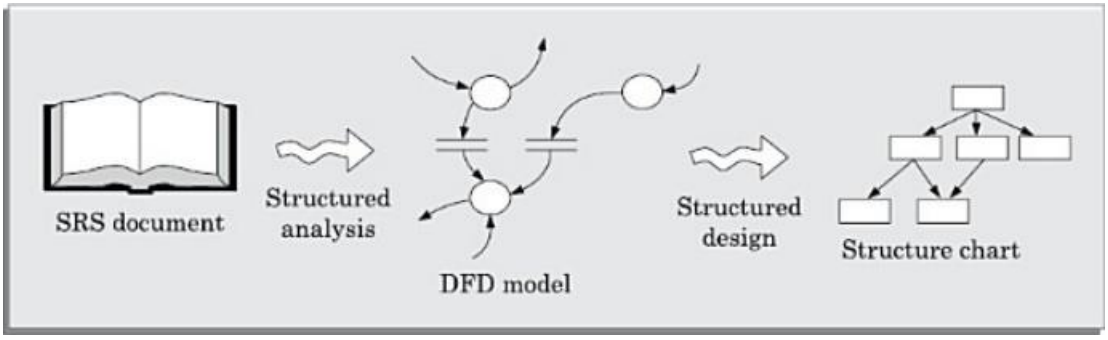

Figure 5.1: Structure Analysis and Structure Design

The figure 5.1 illustrates the schematic responsibilities of Structured Analysis (SA) and Structured Design (SD). Here are some key points from the diagram:

- The SRS document is transformed into a Data Flow Diagram (DFD) model during structured analysis.
- The DFD model is converted into a structure chart during structured design.

Each function required by the system is analyzed and hierarchically decomposed into more detailed functions during structured analysis. Conversely, structured design maps all functionalities identified through structured analysis to a module structure, often referred to as the high-level design or software architecture for the given challenge. A structure chart illustrates this mapping. Typically, the high-level design step precedes the detailed design stage.

During the detailed design stage, algorithms and data structures for the individual modules are designed. The detailed design can be directly implemented using a standard programming language as a working system. Consequently, users can quickly comprehend the results of structured analysis. In fact, in structured analysis, functions and data are labeled using the user's vocabulary, enabling the customer to verify that it meets all their needs.

Gane and Sarson (1979) and DeMarco and Yourdon (1978) made significant contributions to the development of structured analytic methodologies. Structured analysis techniques are founded on the following fundamental principles:

a. Decomposition from the top-down. b. Utilizing the divide and conquer principle, wherein each high-level function is split into detailed functions. c. Employing Data Flow Diagrams (DFDs) to graphically represent the analytic outcomes.

5.3 DFD

A Data Flow Diagram (DFD), also known as a bubble chart, is a hierarchical graphical model of a system illustrating the various processing activities or functions performed by the system, as well as the data interchange between those processes. In this model, each function is depicted as a processing station or process that takes input data and produces output data. The system is represented by its input data, the various processing operations performed on these data, and the system's output data. DFD models represent functions using a relatively small number of primitive symbols.

5.3.1 Importance of DFD in a good software design

The popularity of the DFD technique is most likely due to its relatively simple formalism, making it easy to grasp and apply.

A DFD model illustrates numerous sub-functions hierarchically, starting with a collection of high-level functions performed by a system. In essence, any hierarchical paradigm is easy to comprehend. As a hierarchical model begins with a very simple and abstract representation of a system, different aspects of the system are gradually introduced through multiple hierarchies, facilitating easy understanding by the human mind.

Similarly, the approach of data flow modeling adheres to a straightforward set of intuitive notions and guidelines. DFD is a sophisticated modeling technique that can be utilized not only to depict the results of structured analysis of a software problem but also for various other purposes, such as illustrating the flow of documents or items within an organization.

5.3.2 A Drawback of DFD

DFD models come with various drawbacks, including the following major ones:

a. DFDs leave room for error. In the DFD paradigm, a bubble's function is determined by its label. However, a concise label may not fully represent all the functionality of a bubble. For example, a bubble labeled "find-book-position" may lack specificity, failing to address various aspects such as handling absent or inaccurate input information. Additionally,

such a bubble may not convey information about what happens if the required book is unavailable.

b. DFDs do not define control aspects. The order in which inputs and outputs are consumed and produced by a bubble is not specified in a DFD model. This lack of representation is crucial for modeling real-time systems.

c. The process of decomposition to achieve successive levels, as well as the final level of decomposition, is highly subjective and dependent on the analyst's discretion. Consequently, for the same problem, multiple feasible DFD representations may exist. Furthermore, determining which DFD representation is superior or preferable over another can be challenging.

d. The data flow modeling technique lacks clear guidance on precisely breaking down a particular function into its subfunctions, relying instead on the analyst's judgment to carry out decomposition.

5.3.3 Common Errors During DFD Designs

a. In the context diagram, it's a common mistake for novices to draw more than one bubble. The system should be depicted as a single bubble, while external entities occur at all levels of DFDs, which may confuse newcomers.

b. Only the context diagram should represent all external entities interacting with the system. External entities should not appear at any other levels of the DFD. It's typical to encounter either too few or too many bubbles in a DFD. Ideally, each diagram should contain 3 to 7 bubbles, with each bubble further decomposed into 3 to 7 sub-bubbles. Novices often leave various levels of DFDs imbalanced.

c. Data arrows should only connect a data store to bubbles.

d. A data store cannot be linked to another data store or an external entity.

e. The DFD model should encompass all system functionalities. No system function described in its SRS document should be overlooked.

f. Only system functions indicated in the SRS document should be represented. Designers should not assume system functionality that is not described in the SRS document and then attempt to represent it in the DFD.

g. The names of data and functions must be understandable. Some students and practicing engineers use symbolic data names like a, b, c, etc., which can make it difficult to understand the DFD model.

5.4 Structure Design

The purpose of structured design is to convert a DFD representation of the structured analysis results into a structured chart. Structured design provides two methods to guide the transformation of a DFD into a structure chart:

- Transform analysis
- Transaction analysis

The standard procedure involves starting with the level 1 DFD, converting it into a module representation using either the transform or the transaction analysis method, and then progressing to the lower-level DFDs. It is essential to determine whether transform or transaction analysis is applicable to a particular DFD at each level of transformation. The following subsections describe these methods.

5.5 Transform Analysis

Transform analysis identifies the key functional components (modules) and their high-level inputs and outputs. The process involves segmenting the DFD into three sorts of segments:

- Input
- Logical reasoning
- Output

The input section of a DFD encompasses operations that convert physical input data (e.g., a character from a terminal) to logical input data (e.g., internal tables, lists, etc.). Each input part is referred to as an afferent branch. Conversely, the output section of a DFD converts output data from logical to physical form, with each output component termed as an efferent branch. The central transform constitutes the remaining section of a DFD.

In the subsequent step of transform analysis, a structure chart is created by delineating a functional component for the central transform, as well as the afferent and efferent branches. These are depicted beneath a root module that would invoke these modules. Identifying the maximum level of input and output transformations necessitates knowledge and experience. One approach is to track the inputs until reaching a bubble whose output cannot be deduced solely from its inputs.

Central transforms do not check input or add information to it; rather, they may sort input or filter data.

In the third step of transform analysis, the structure chart is modified by adding sub-functions required by each of the high-level functional components. Various levels of functional components can be incorporated, and the process of factoring involves splitting functional components into subcomponents. This includes adding read and write modules, error-handling modules, initialization and termination processes, identification of customer modules, etc. The factoring process continues until all bubbles in the DFD are represented on the structure chart.

5.6 Transactional Analysis

The transaction is a crucial component facilitating users in accomplishing significant tasks. In the development of transaction processing programs, transaction analysis plays a pivotal role. In a transaction-driven system, the path through the Data Flow Diagram (DFD) varies based on the input data item, leading to one of several potential paths. In contrast, a transform-centered system involves similar processing procedures for each data item.

A transaction can be defined as any distinct method of handling input data. Identification of transactions can be straightforward by checking the input data. The number of transactions is determined by the number of bubbles representing input data on the DFD. It's worth noting that some transactions may not require input data, which can be identified through analysis.

For each transaction, it's essential to trace it to its corresponding result. All bubbles traversed by a transaction are grouped into the same module on the structure chart. Initially, a root module is drawn on the structure chart, and subsequently, a module is created for each identified transaction. Each transaction is tagged to denote its type, aiding in the separation of the system into transaction modules and a transaction-center module through transaction analysis.

5.7 Structure Chart

A structure chart serves as a visual representation of the software architecture, illustrating the system's components, dependencies (such as which modules call each other), and the flow of parameters between modules. It's notably straightforward to construct a structure chart using a computer language. However, it's important to note that procedural details, like how specific functionalities are generated, are not reflected in a structure chart. Instead, the focus is primarily on the modular design of the software and the connections among different modules.

The basic building blocks used in creating structure charts include:

a. Rectangular boxes: These boxes represent modules within the system.

b. Module invocation arrows: These arrows indicate the transfer of control from one module to another, pointing in the direction of control flow.

c. Dataflow arrows: Directed arrows labeled with data names represent the flow of data from one module to another, pointing in the direction of data movement.

d. Library modules: A rectangle with double edges signifies a library module.

e. The diamond sign represents a selection process.

f. Repetition: A loop around the control flow arrow symbolizes repetition.

5.8 Summary:

- Structured analysis/structured design (SA/SD), a function-oriented software design technique, combines elements from various significant design methodologies.
- Good designs meeting multiple goodness criteria can be created using methodologies like SA/SD, which consists of structured analysis and structured design components.
- Structured analysis aims to provide a functional breakdown of the system, often depicted using data flow diagrams (DFDs). Implementing DFD representations using conventional programming languages can be challenging.
- Structure chart representations can be created by systematically transforming DFD representations. Implementing the structure chart representation using a normal programming language is relatively straightforward.

5.9 Keywords

- Structure Charts: A structure chart represents the software architecture, including multiple modules, module dependencies (which modules call which other modules), and the parameters exchanged between modules. It's straightforward to construct using a programming language. However, procedural aspects, such as how a specific functionality is produced, are not depicted in structure charts, as their primary focus is on the module structure and interaction among different modules.
- Structure Analysis: The aim of structured analysis is to capture the system's detailed structure as experienced by the user. This ensures that users can easily understand the

results of structured analysis. In structured analysis, functions and data are named using the user's vocabulary.

- DFD: Data Flow Diagrams (DFDs) are a powerful modeling approach not only for illustrating the results of structured analysis of a software problem but also for various other purposes, such as displaying the flow of documents or items within an organization.
- Transactional Analysis: Transaction analysis is a valuable alternative to transform analysis for creating transaction processing programs. Transactions allow users to execute specific tasks using software, such as "issue book," "return book," "search book," and so on.
- Transform Analysis: Transform analysis identifies the key functional components (modules) and their input and output data. During the initial phase of transform analysis, the DFD is divided into three sorts of sections: Input, Processing, and Output.

5.9 Self-Assessment Questions

1. How do you define "structured analysis" and "structured design"?
2. What are the main objectives of "structured analysis" and "structured design"?
3. Explain the role of the DFD model in understanding how a software system works.
4. What do "transform analysis" and "transactional analysis" mean to you?
5. What are the primary drawbacks of using a data flow diagram (DFD) for structured analysis?

5.11 Case Study: Function-Oriented Design in Software Development

Introduction: XYZ Solutions specializes in building enterprise-level applications for various industries. Recently, they completed a transportation management system project for a logistics company.

Objective: This case study aims to demonstrate the application of function-oriented design in software development and its benefits for the transportation management system project.

Recommendations:

1. Educate the development team and stakeholders about the concept and advantages of function-oriented design.
2. Ensure project requirements and goals align with the principles of function-oriented design.

3. Establish clear guidelines and standards for implementing function-oriented design in the project.

Questions to be considered: a) How were the functions of the transportation management system identified and defined? b) What methodologies or techniques were employed to achieve function-oriented design? c) What were the benefits and challenges encountered during the implementation of function-oriented design?

Recommendations:

1. Conduct a comprehensive analysis of the transportation management system's requirements to identify its core functions.
2. Utilize structured techniques such as structured analysis, data flow diagrams, and entity-relationship diagrams to model and define the system's functions.
3. Conduct regular reviews and walkthroughs to validate the function-oriented design and ensure alignment with project objectives.
4. Establish metrics to measure the effectiveness and efficiency of the function-oriented design implementation.

Benefits of Function-Oriented Design:

- How did function-oriented design contribute to the success of the transportation management system project?
- What were the specific benefits observed in terms of system functionality and maintainability?
- How did function-oriented design help in managing future changes and enhancements?

Recommendations:

a) Encourage the use of function-oriented design in future projects to leverage its benefits.

b) Invest in training and upskilling the development team in function-oriented design principles and methodologies.

c) Continuously evaluate and improve function-oriented design practices based on lessons learned from the transportation management system project.

Conclusion: Function-oriented design is a valuable approach in software development, offering clarity, modularity, and maintainability to software systems. By applying function-oriented design principles and methodologies, XYZ Solutions successfully developed a robust transportation

management system. The company should continue to embrace function-oriented design in future projects and refine its practices to further improve software quality and maintainability.

5.11 References:

- I. Somerville: Software Engineering, 9th Edition, Pearson Education, 2017.
- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.
- Roger S. Pressman: A Practitioner's Approach, 7th Edition, McGraw Hill Education, 2009.

Unit - 6

Object- Oriented Design

Learning Objectives:

- Describe what is a model.
- Describe how models can be useful.
- Describe what UML is & the origins of UML, and its acceptance in the industry.
- Recognize the many sorts of views captured by UML diagrams.
- To comprehend the many forms of UML diagrams

Structure:

Introduction

The Object-Oriented Software Development methodology enjoys widespread popularity and is extensively utilized by both industry professionals and academic scholars. Object technologies have evolved significantly since their inception in the early 1980s, progressing steadily from their modest beginnings. The adoption of the object-oriented method has surged due to the multitude of benefits it offers.

During the 1990s, object technology gained prominence, reaching an advanced stage of development. Given the pervasive use and popularity of object technologies in various sectors, it is imperative to have a well-developed understanding of this technology. Proficiency in programming languages such as Java or C++, which are object-oriented, is essential for developing high-quality Object-Oriented Software.

6.1 Object-Oriented Design

The fundamental concepts of object-oriented programming are depicted in the figure. We will delve into a detailed discussion of these concepts in the forthcoming subsection (Figure 6.1).

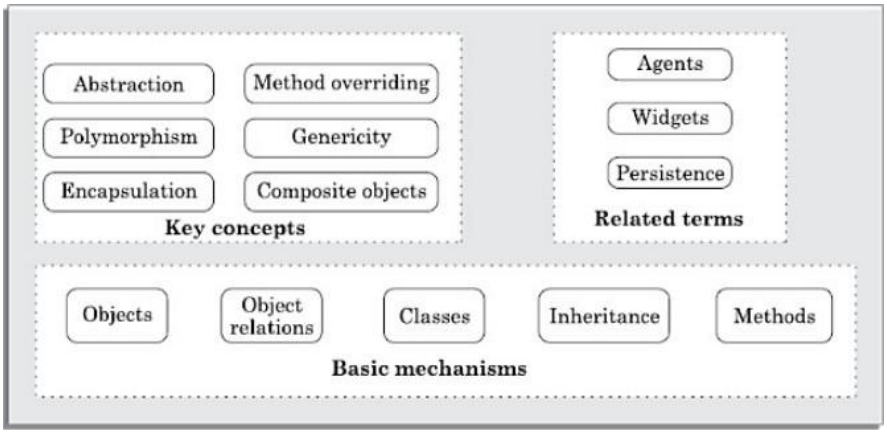

Figure 6.1: Object-oriented design concept

6.1.1 Object

An object represents a real-world entity with associated attributes, which are private to the object, and methods that operate on its data. Real-world entities such as books, cars, houses, people, and accounts are examples of objects. Object-oriented approaches facilitate the understanding of software systems by envisioning them as interactions between sets of objects. For instance, in library automation software, activities like issuing a book and stamping the return date involve interaction between the book object and the issue register object. Each object contains relevant data and methods to manipulate that data. For example, a library member object might include private data such as the member's name, membership number, address, phone number, email address, date of membership, membership expiry date, and outstanding books. Methods to operate on this data could include "issue-book," "find-books-outstanding," "find-books-overdue," "return-book," and "find-membership-details."

6.1.2 Class

A class serves as a blueprint for creating objects. Once a class is defined, multiple objects of that class can be created, all sharing similar properties and methods. For instance, the Library Member class defines properties such as member Name, email, joining Date, expiry Date, etc., and methods like issue-book, return-book, and find-book. A class is a user-defined data type.

6.1.3 Methods

Methods represent operations that objects can perform, defined within a class. For example, methods in a Library Member class could include issue-book, return-book, and find-book.

6.1.4 Encapsulation

Encapsulation refers to the practice of binding related attributes and methods within a class. It allows data to be kept private and accessed only through the methods defined in the class (Figure 6.2).

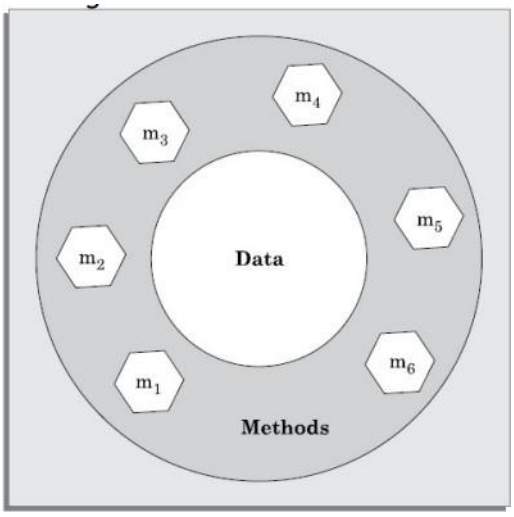

Figure 6.2: Encapsulation

6.1.5 Inheritance

Inheritance is used to create a new class using the existing class. The existing class is known as a "super class", and the newly derived class is known as "sub-class". The features of the base class are said to be inherited by the derived class. The classes Faculty, Students, and Staff, for example, have been derived from the basic class, Library Member (Figure 6.3).

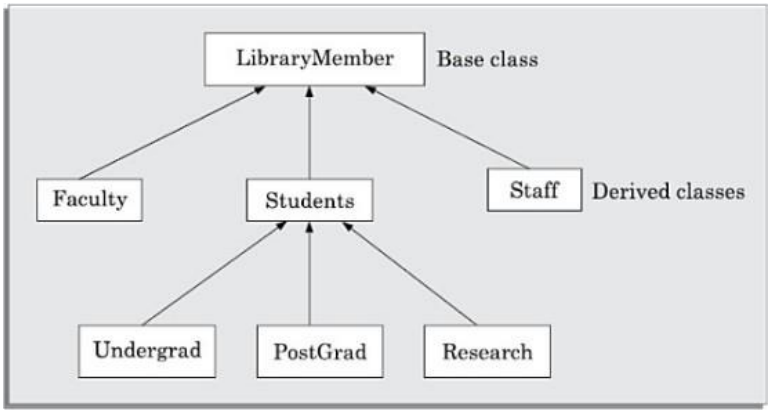

Figure 6.3: Library Management System

6.1.**6 Over loading**

Method overloading refers to the practice of defining the same method name with different input parameter lists. This allows multiple methods with the same name but different parameter types or numbers.

6.1.7 **Over riding**

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The method signature remains the same, but the implementation differs in the subclass.

6.1.**8 Polymorphism**

Polymorphism refers to the ability of a single method to exhibit different behaviors in different contexts. This can be achieved through method overloading or method overriding.

6.1.9 **Abstraction**

Abstraction involves hiding the complex implementation details of a class and exposing only the necessary functionalities to interact with the class. When creating a class, abstraction allows developers to focus on defining the essential features and behaviors without exposing unnecessary details.

6.2 Unified Modeling Language (UML)

UML is a modeling language used to visualize software systems. It provides a set of notations, such as rectangles, lines, and ellipses, to create visual representations of systems. UML has its structure and semantics, and while it is not a system development method itself, it can be used to describe the results of object-oriented analysis and design obtained through other methods.

6.2.1 The Emergence of UML

In the late 1980s and early 1990s, various object-oriented design techniques and notations emerged, leading to confusion due to the lack of standardization. UML was created to standardize these notations and provide a unified approach to object-oriented modeling. It consolidated different methodologies and notations used at that time, including Object Management Technology, Booch's methodology, Object-Oriented Software Engineering, Odell's methodology, and Shlaer and Mellor methodology.

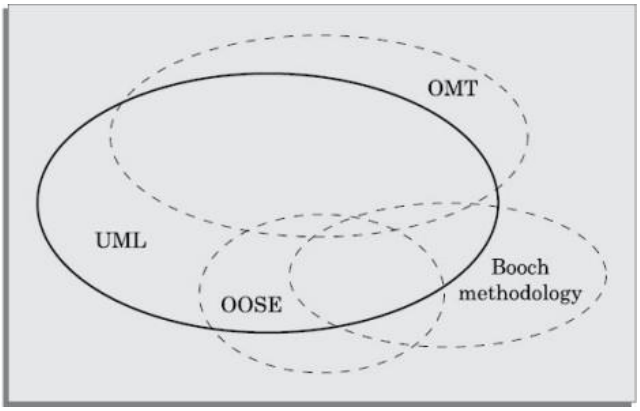

Figure 6.4: Impact of different modeling techniques on UML

UML is adopted by OMG(Object Management Group) and accepted in the wide softwareindustry.UML can be used in both small and large problems. Now UML is used worldwide for modelling system design (Figure 6.4).

6.3 UML Diagrams

UML is utilized to create nine types of diagrams that capture five distinct views of a system. These diagrams offer multiple perspectives on the software system, aiding in developing a thorough understanding. The models derived from UML can then be refined to guide the actual implementation of the system.UML diagrams can represent the following five system views (Figure 6.5):

- The user's perspective
- Structural perspective
- Behavioral perspective
- Implementation perspective
- Environmental perspective

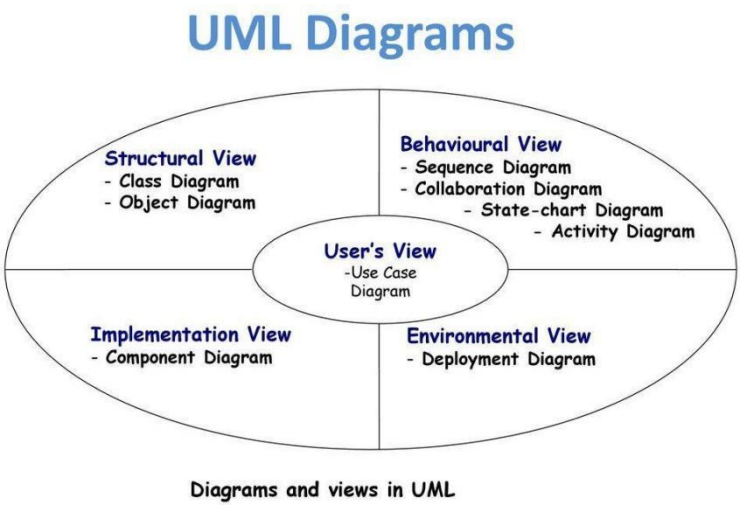

Figure 6.5: UML Diagrams and Different Views

6.3.1 Structural View

The structural view defines the necessary objects (classes) required for understanding and implementing a system. It also illustrates the relationships between these classes (objects). Since the structure of a system remains constant over time, the structural model is also referred to as the static model.

6.3.2 Behavioral View

The behavioral view describes how elements interact with each other to comprehend system behavior. It captures the system's time-dependent (dynamic) behavior.

6.3.3 Implementation View

This view depicts the primary elements of the system and how they are interconnected, focusing on the implementation aspects of the system.

6.3.4 Environmental View

The environmental view illustrates how various components are implemented on different hardware pieces, providing insights into the system's deployment environment.

6.3.5 User View

The user view outlines the system's functionalities accessible to users. It represents the external users' perspectives on the system's operations, offering a black-box view where internal structure, dynamic behavior of components, and implementation details are abstracted. Unlike other views, the user's view is a functional model, while the others are object models. The user's view is the primary focus, with all other perspectives expected to align with it, forming the core of any user-centric development approach.

6.4 Use Case Diagrams

A set of "usecases" defines the use case model for any system. Usecases display the different ways that users can interact with a system. A simple way to find all of a system's usecases is to simply ask, "What can the user do with the system?" Thus, the usecases for the Library Information System could be:

- issue-book
- query-book
- return-book
- create-member
- add-book, and soon.

Usecases are similar to high-level functional requirements. Usecases break down system behavior into transactions, with each transaction performing a useful function from the user's perspective. Completing a transaction may involve a single message or multiple exchanges between the user and the system (Figure 6.6).

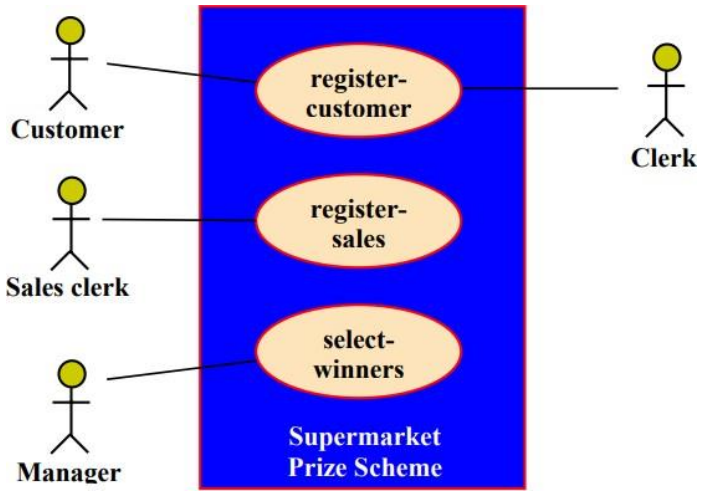

Figure 6.6: Use Case Diagram for Supermarket

The use case model for the Supermarket Prize Scheme. We can find three use cases: "register-customer", "register-sales", and "select-winners". The text description for use case "register-customer" is presented as an example.

U1 text description: “register-customer” Using this use case, consumer can register himself by supplying required information.

Scenario1: Primary series

1. Customer: “Select the Register Customer option”.
2. System: present a pop up for you to input your name, address, and phone number.
3. Customer: fill in the blanks with the correct data.

Scenario2: “at the final step of the main line sequence”

1.The system suggests “the customer has already registered”.

Scenario3: “at the fourth step of the main line sequence”

1. The system displays a message indicating that some input data was not submitted and prompts you to enter the missing value.

U2: “register-sales”: Using this use case, the clerk can record the details of a customer's purchase.

Scenario1: Main line Sequence

1. The clerk chooses register sales option.
2. System: shows a prompt for customer's id and purchase data.
3. Clerk: inputs the necessary information.

4: The system provides a notification indicating that the sale has been successfully registered.

U3: Choose a Winner: The manager may produce the winner list applying this use case.

Scenario2: Main sequence

1. "The manager selects the select-winner option".
2. System: "displays the gold coin and the list of surprise gift winners".

6.5 Class Diagrams

A class diagram shows a system's static structure. It displays the structure of a system rather than how it operates. A system's static structure is made up of many class diagrams and the dependencies between them. A class diagram's basic components are classes and their relationships—"generalization", "aggregation", "association", and "other types of dependencies". We will now go through the UML syntax for representing the classes and their relationships (Figure 6.7).

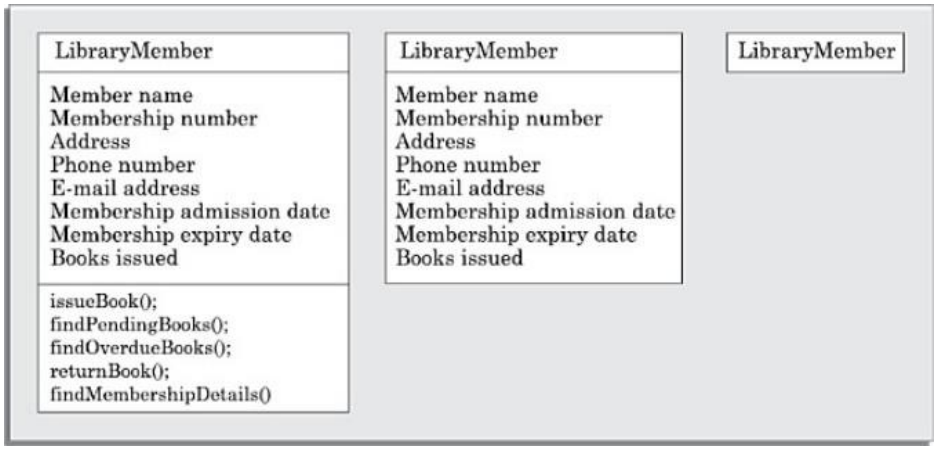

Figure 6.7: Different Representations of a Class

6.5.1. Class

The classes define entities with similar characteristics, such as attributes and methods. Solid outline rectangles with sections are used to represent classes. Classes must contain a name compartment with the name printed centred in boldface. Class names are usually written in mixed case, starting with an uppercase letter, and are typically chosen as singular nouns.

6.5.2. Attributes

An attribute is a named property of a class that represents the type of data an item can have. Attributes are shown with their names and can optionally include a type specification, a default value, and constraints. The attribute type is specified by appending a colon and the type name after the attribute name. The first character of an attribute name is usually a tiny letter. For example, **book name: String**

6.5.3. Operation

An operation is a function implementation that can be accessed from any object of the class to affect its behavior. Invoking an operation on an object alters the object's data or state. A class can have any number of operations, including none. Typically, the first letter of an operation name is lowercase. Italics are used for abstract methods. An operation's arguments (if any) may have a type defined, which can be 'in','out', or 'input'. A single return type expression can be used as the return type for an operation. For example: **issue book (in book name):Boolean**

6.5.4. Association

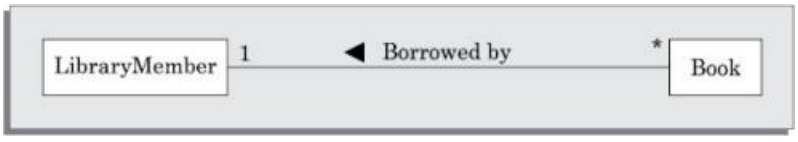

Figure 6.8: Association between 2 Classes

A connection between classes is described by an association. Object connection or link refers to the associative relationship between two objects. Associations are represented by links. The association of the two groups is illustrated by drawing a straight line between them. Along with the association line is written the name of the organization. An arrowhead may be added to the

association line to indicate the direction in which the association should be read. The arrowhead should not be misinterpreted as representing the location of a pointer that is implementing an association. The multiplicity is noted as an individual number or as a value range on either side of the associated connection. The multiplicity of a class specifies how many instances of that class are linked with one another.

In the above figure 6.8, an asterisk denotes a wild card that has the meaning of one or many. The figure denotes many books can be borrowed by a Library Member.

6.5.5 Aggregation

In the concept of Aggregation, related classes are not merely connected but also exhibit a whole-part relationship. This means that the aggregate object not only maintains references to its components, enabling access to their methods, but also takes responsibility for their creation and deletion. For example, a book registry exemplifies aggregation since it aggregates book objects, allowing books to be added to or removed from the registry as required.

In UML, an empty diamond symbol at the aggregate end of a relationship signifies aggregation. The diagram provided offers an example of such an aggregation connection. It demonstrates how a document can be conceptualized as a collection of paragraphs, with each paragraph potentially containing multiple lines. The notation "1" marked at the diamond end and "*" marked at the other indicates that a single document can encompass multiple paragraphs. However, if the intention were to specify that a text comprises precisely ten paragraphs, the number "9" would replace the "*" (Figure 6.9).

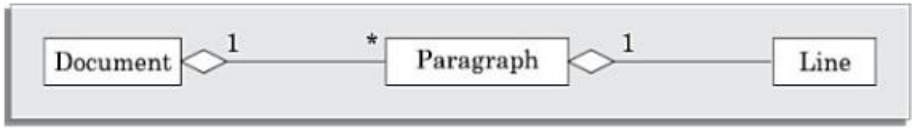

Figure 6.9: Aggregation Relationship

6.5.6 Composition

Composition is a stronger form of aggregation in which the parts depend on the whole for existence. This indicates that parts' existence is unable to exist without whole. In a nutshell, the vital connection of the whole and the life of each element are the same. The parts develop when the whole develops, and the parts are destroyed when the whole is destroyed. An order object is an example of composition because no item in the order can be changed after it is placed. If any changes to the order products need to be made after the purchase, the entire order must be cancelled, and a new order with the modified products should be placed. In this scenario, when an order object generates, every order item inside it is also created, and when an order object is destroyed, all order items inside it are also removed. That is, aggregate (order) has same life as components (order items). A full diamond drawn at the composite end represents the composition relationship (Figure 6.9).

Figure 6.9: Composition Relationship

6.6 Sequence Diagram

A sequence diagram is a graphical representation that depicts the interaction between objects in a system. It is typically read from left to right and consists of objects represented by rectangle boxes connected by vertical dashed lines at the top of the diagram. The name of the object is written inside the box, followed by a colon and the class name, which is underlined to signify that any instance of the class can be referred to.

Objects appearing at the top of the sequence diagram exist before the start of the use case execution. However, if an object is created during the execution and contributes to the interaction (such as through a method call), it should be shown at the position where it was created on the diagram.

The vertical dashed line representing an object's lifeline indicates the object's existence at different points in time. If an object is destroyed during the execution, its lifeline may be crossed at that point, indicating its termination. The activation symbol, depicted as a rectangle drawn on an object's lifeline, signifies the times when the object is active. As long as the activation symbol is present on the lifeline, the object is considered active.

Each message is labeled with the message name (Figure 6.11).

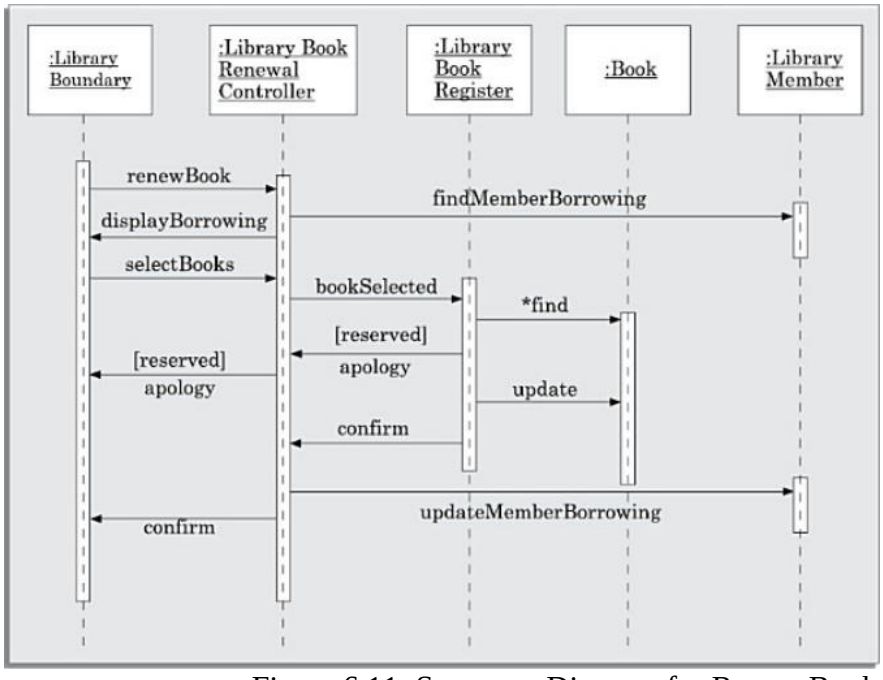

Figure 6.11: Sequence Diagram for Renew Book

6.7 State Chart

A state chart diagram is commonly employed to illustrate how an object's state evolves over time. These diagrams are useful for showcasing how an object's behavior varies across different executions of use cases. However, state chart diagrams may not be suitable when depicting behavior involving multiple objects collaborating. In such cases, sequence or collaboration diagrams are more appropriate for modeling this type of interaction.

The finite state machine (FSM) formalism serves as the foundation for state chart diagrams. A finite set of states that correspond to the states of the modelled object make up an FSM. When particular occurrences take place, the object's state changes.

The following are the basic elements of a state chart diagram:

Initial state: “The initial state is shown as a filled circle”.

Final State: “A filled circle inside a larger circle represents the final state”.

State: “Rectangles with rounded corners are used to depict them”.

A transition in a state diagram is depicted by an arrow connecting two states. The event triggering the transition is typically labeled alongside the arrow. Additionally, a guard condition may be included to determine whether the transition can take place. A guard is a Boolean logic condition, and the transition can only occur if the guard evaluates to true.The syntax for the transition label is shown in three parts—[guard]event/action (Figure 6.12).

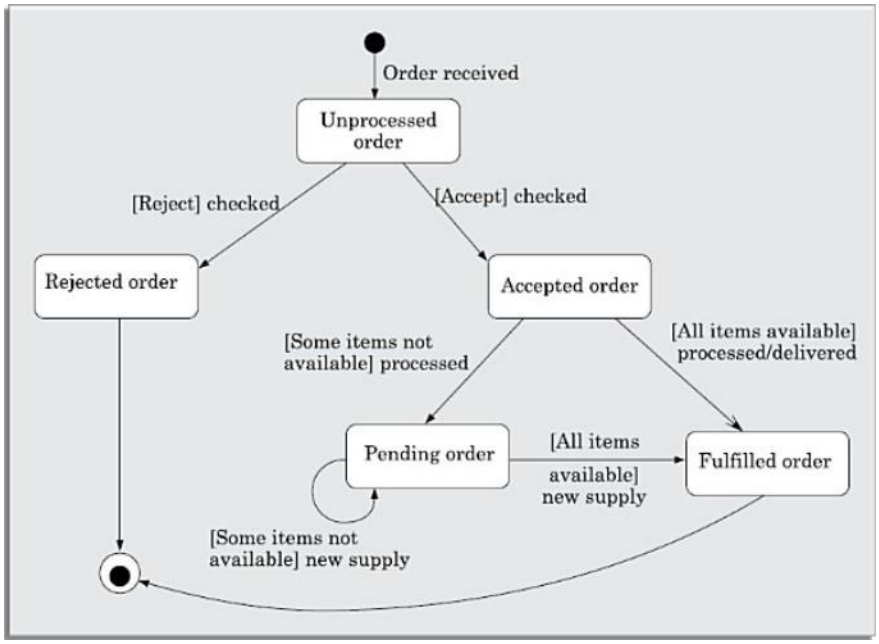

Figure 6.12: State Chart diagram for order

6.8 Summary

- ❖ We began this chapter by reviewing some key ideas related to object orientation.
- ❖ One of the key benefits of object orientation is greater software development team productivity.
- ❖ The increased productivity in object-oriented projects largely stems from the reuse of predefined classes and partial reuse through inheritance. Additionally, the conceptual simplicity offered by the object-oriented approach contributes significantly to this enhanced productivity.
- ❖ Object modeling plays a crucial role in system analysis, design, and understanding. The Unified Modeling Language (UML) has gained widespread popularity and is progressively becoming a standard in object modeling.
- ❖ We talked about the syntax and semantics of some common sorts of diagrams that maybe created with UML. We will go over an object-oriented system development process that employs UML as a model documentation tool.

6.9 Keywords

- **Use Case:** A series of" use cases" makeup the use case model for any system. Use cases intuitively reflect the various ways that consumers can utilize technology.
- **Sequence Diagram:** A sequence diagram depicts item interaction as a two-dimensional chart. The lifeline of an object is shown as a vertical dashed line in a sequence diagram. A narrow between the lifelines of two objects separates each message. The message are displayed from top to bottom in chronological order. In other words, the order of the messages may be seen by reading the graphic from top to bottom.
- **State Chart:** A state chart diagram is typically used to represent how an object's state changes over time. State chart diagrams are effective in illustrating how an object's behavior varies a cross numerous use case executions.
- **Aggregation:** Aggregation is a sort of connection in which the classes involved represent a whole-part relationship. The aggregate is in charge of forwarding messages to the appropriate sections. As a result, the aggregate assumes responsibility for delegation and leadership.

- **Association:** For objects to be able to communicate with one another, associations are required. An association describes how two classes are related. Object connection or link refers to the associative relation between two objects. Examples of associations are links. A link connects instances of an object mentally or physically.

6.10 Self-Assessment Questions

1. How to identify the use cases of a system? Identify the Use Case of LIS (Library Information System).
2. Which UML diagrams capture the important components of the system and their dependencies? Justify the ATM cash withdrawn Situation.
3. Bill contains many items. Each item describes some commodity, the price of a unit, and the total of this price. Create a Use Case diagram for this scenario.
4. How will you identify the Use case of the Library Management system?
5. What does the aggregation relationship between classes represent? Give example

6.11 Case Study

Introduction: In a software development project, the implementation of different use case views is crucial for capturing and understanding the functional requirements of the system. However, there are challenges in effectively implementing these views. Let's explore a case study and provide recommendations to address the issues.

Problem: A software development team is struggling to implement different use case views for their project. The challenges they face include:

1. **Incomplete and ambiguous use case descriptions**: The use case descriptions lack the necessary details and are often ambiguous, leading to misinterpretation and confusion among team members.
2. Lack of user-centric perspective: The use cases fail to focus on the needs and goals of the system's end-users, resulting in a misalignment between the system's functionality and user expectations.
3. Inconsistent and disconnected use case diagrams: The use case diagrams do not effectively represent the relationships between the use cases and the actors. They lack clarity and coherence, making it difficult to understand the overall system behavior.

Recommendations:

1. Thorough Requirements Gathering: Engage stakeholders, including end-users and domain experts, to gather comprehensive requirements by asking questions such as:

- What are the key goals and objectives of the system?
- Who are the primary actors interacting with the system?
- What are the main functions or tasks that the system should perform?

2. Apply User-Centered Design Principles: Ensure that the use cases are defined from a user-centric perspective by asking questions like:

- How will the system fulfill the needs and expectations of the end-users?
- Are there any specific user roles or personas that need to be considered?
- What are the main user interactions and workflows within the system?

3. Create detailed Use Case Descriptions: Document use cases with clear and detailed descriptions by addressing questions such as:

- What are the preconditions and postconditions of each use case?
- What are the primary and alternative flows of events?
- Are there any exceptional or error conditions that need to be handled?

4. Validate and Refine Use Case Descriptions: Collaborate with stakeholders and subject matter experts to validate and refine the use case descriptions, considering questions like:

- Are the use case descriptions accurate and complete?
- Do they reflect the intended system behavior and user requirements?
- Are there any missing or conflicting use cases that need to be addressed?

5. Improve Use Case Diagram Representations: Enhance the clarity and coherence of use case diagrams by addressing questions such as:

- Do the use case diagrams accurately represent the relationships between actors and use cases?
- Are the use cases organized and grouped in a logical and meaningful manner?
- Can the use case diagrams be easily understood and interpreted by all stakeholders?

6. Utilize UML Tools and Templates: Utilize UML modeling tools and predefined templates to facilitate the creation and documentation of use case views, ensuring standard notations and symbols for consistency and professionalism.

7. Conduct Regular Reviews and Walkthroughs: Regularly conduct reviews and walkthroughs of the use case views with stakeholders and development team members to gather feedback, clarify ambiguities, and ensure a shared understanding of the system requirements.

By implementing these recommendations, the software development team can overcome challenges and create comprehensive and well-defined use cases that accurately represent the system's functional requirements and align with user expectations.

6.12 References

- Summerville I: Software Engineering9th Edition, Pearson Education, 2017.
- Rajib Mall: Fundamentals of Software Engineering, 4thEdition, Prentice Hall of India, 2014.
- Roger S. Pressman: A Practitioner's Approach, 7thEdition, McGraw Hill Education, 2009

Unit - 7

Code Review

Learning Objectives:

- Determine the importance of coding standards.
- Understand the difference between coding standards and coding guidelines.
- To clarify what code review entails.
- Discuss the importance of properly documenting software.
- Make a difference between internal and external documents.

Structure:

Introduction

Coding commences once the design phase concludes, and the design documents undergo thorough review. Each module specified in the design documentation is coded and subjected to unit testing during this coding process. Unit testing involves testing each module separately from the others; once a module's coding is complete, it undergoes independent testing.

Following the completion of coding and unit testing for all modules, the integration and system testing phase commences. The design document produced at the end of the design phase serves as the input to the coding phase. This document encompasses both the high-level system design, typically represented in the form of a module structure (e.g., a structure chart), and the detailed design. The functional architecture is typically articulated through module requirements, which delineate the data structures and algorithms for each module. Accordingly, various modules identified in the design document are coded in accordance with their module requirements during the coding phase.

7.1 Coding Standard

Developers must adhere to coding standards, which are verified during code inspection. Any code that deviates from these standards is rejected during code review and must be rewritten by the responsible programmer. While coding standards provide basic recommendations regarding coding style, the actual implementation is left to the discretion of individual developers.

Effective software development companies often establish their own coding standards and rules tailored to their organization and the nature of the products they develop.

7.1.2 The following are some examples of coding standards.

a. The rules for limiting the use of global variables specify which types of data can and cannot be declared as global.

b. The headers preceding the codes for various modules contain organization-wide standard information. This includes specifying the format in which the header information is organized. Examples of standard header data include the module's name, the date of development, the author's name, modification history, module synopsis, supported functions and their input/output parameters, and module accesses/modifications of global variables.

c. Naming rules for global variables, local variables, and constants suggest that global variable names start with a capital letter, local variable identifiers begin with small letters, and constant names begin with uppercase letters.

d. Error return standards and handling of exceptions mechanisms ensure uniform reporting of error situations by different functions in a program and their handling within an organization. For instance, when an error condition is encountered, different functions should consistently return a 0 or 1.

7.1.3 Coding Guidelines

a. Employ a coding style that is neither overly complex nor too difficult to understand. Simple code is easier to comprehend. Inexperienced developers may enjoy crafting intricate and unreadable code, but clever coding can obscure the code's meaning and hinder maintenance.

b. Be mindful of the side effects of function calls, such as parameter changes provided by reference, alterations to global variables, and I/O operations. Unclear side effects make it challenging to understand code, especially when unexpected changes occur, such as partial updates to global variables or file I/O operations, which may not be evident from the function's name and header information.

c. Avoid using the same identifier for multiple purposes. Programmers often use a single identifier to represent multiple temporary entities, such as using a temporary loop variable to compute and store the final result. While this may save memory, it can lead to confusion and should be avoided.

d. Ensure proper documentation of the code. As a general guideline, there should be at least one comment line for every three source lines.

e. Limit the size of functions to no more than ten source lines. Long functions tend to be more difficult to understand and are prone to containing multiple functionalities, increasing the likelihood of bugs.

f. Use go to statements sparingly. Overuse of go to statements can make a program disorganized and harder to comprehend.

7.2 Coding Walkthrough

A code walkthrough is an informal code examination method conducted after the module has been encoded, properly compiled, and syntax errors have been rectified. A select few members of the

development team are provided with the code to study and understand a few days prior to the walkthrough meeting. Each member selects specific test cases and individually traces the execution of the code, examining each statement and function execution. The primary objectives of the walkthrough are to identify algorithmic and logical issues in the code. Participants make notes of their observations in preparation for a walkthrough meeting with the module's coder.

While code walkthroughs are informal analysis techniques, several concepts have evolved over the years to enhance the effectiveness of this simple yet useful method. These guidelines are rooted in personal experience, common sense, and various subjective considerations. Therefore, they should be regarded as examples rather than rigid rules. Some of these guidelines include:

- The team conducting the code walkthrough should ideally comprise three to seven members.
- The focus of the discussion should be on identifying errors rather than devising solutions to the identified problems.
- Managers should refrain from attending code walkthrough meetings to foster teamwork and prevent the perception among engineers that they are being evaluated.

7.3 Code Inspection

In contrast to code walkthroughs, the purpose of code inspection is to identify common types of problems stemming from ignorance and poor programming practices. Instead of manually simulating code execution as done in walkthroughs, code inspection scrutinizes the code for specific types of errors. For instance, it's more likely to catch errors like altering a formal parameter in a procedure while it's called with an actual constant parameter through code inspection rather than mere hand simulation of procedure execution.

During code inspection, adherence to coding standards is also verified alongside typical faults. Successful software development companies compile data on common errors made by their engineers and identify the most frequently occurring ones. This list of common faults serves as a reference during code inspection to detect potential issues.

Here are some common programming faults that can be uncovered during code inspection:

- Using uninitialized variables.
- Entering loops incorrectly.
- Loops that fail to terminate.

- Assignments that are incompatible.
- Array indices that exceed limits.
- Insufficient storage allocation and deallocation.
- Mismatch between actual and formal parameters in function calls.
- Incorrect usage of logical operators or inappropriate operator precedence.
- Incorrect modification of loop variables.
- Comparison of floating-point variables without proper consideration of precision, and so forth

7.4 Documentation

When developing various types of software products, it's crucial not only to focus on executable files and source code but also on creating various documents such as user manuals, software requirements specifications (SRS) documents, design documents, test documents, installation manuals, and more. All these documents play essential roles in ensuring excellent software development practices. Effective documentation serves several functions:

- Improved accessibility and reliability of a software product: Good documentation reduces the effort and time required for maintenance tasks.
- Assisting users in efficiently using the system: Documentation helps users navigate and understand how to effectively utilize the software.
- Aiding in managing labor turnover: Good documentation helps mitigate the impact of employee turnover. Even if an engineer leaves the organization, a new engineer can quickly acquire the necessary knowledge from well-documented materials.
- Efficiently tracking project progress: Creating comprehensive documentation assists project managers in tracking the progress of the project. They can assess quantifiable progress based on completed work and the evaluation of necessary paperwork.

Different sorts of software documents can be broadly grouped as follows:

- Internal Documentation
- External Documentation

7.4.3 Internal Documentation

Here are examples of essential types of internal documentation:

a. **Comments within the source code**: Including comments within the source code to explain complex logic, algorithms, or any other information necessary for understanding the code.

b. **Meaningful variable names**: Choosing variable names that accurately describe their purpose or usage within the code, making it easier for developers to understand the functionality.

c. **Headers for packages and functions**: Providing clear headers for packages and functions that describe their purpose, input parameters, return values, and any other relevant information.

d. **Indentation of code**: Properly indenting the code to visually represent the structure and hierarchy of the code, making it easier to follow and understand.

e. **Code structure**: Organizing the code into logical modules and functions, each serving a specific purpose, to improve readability and maintainability.

f. **Enumerated types**: Using enumerated types to define a set of named constants, improving code clarity and preventing errors from using incorrect values.

g. **Constant identifiers**: Utilizing constant identifiers to represent fixed values that are used repeatedly throughout the code, enhancing code readability and maintainability.

h. **User-defined data types**: Defining custom data types to encapsulate related data and operations, improving code organization and abstraction.

Internal documentation encompasses various code comprehension features provided within the source code, such as module headers, comments, variable names, code structure, and more. While comments are commonly emphasized, studies suggest that meaningful variable names contribute more to code comprehension. Strong software development organizations establish robust internal documentation practices through the implementation of coding standards and guidelines.

7.4.1 External Document

Various types of supporting documents, such as user manuals, documents describing the software requirements, design documents, test documents, etc., are used to offer external documentation. All of these documents are created in quick time and with high standards due to a disciplined approach to software development. Compatibility with the code is an essential requirement for any acceptable external documentation. If any of the documents are incompatible, it will be very hard for someone to understand the software. To put it differently, all documents developed for a product should be up-to-

date, and any modifications to the code should be reflected in the relevant external documents. Even just a few old documents might cause misunderstanding and inconsistency. Proper understandability by the group of people for whom the document was created is an essential requirement for external documents. Gunning's fog index is highly effective for doing this. We will talk about this next.

7.5 Gunning Fog's Index

Gunning's fog index, devised by Robert Gunning in 1952, serves as a metric to evaluate a document's readability. This metric indicates the number of years of formal education necessary for a person to comfortably comprehend the document. For instance, if a text has a fog index of 12, it suggests that individuals who have completed 12 years of schooling should have no difficulty understanding it. The calculation of Document D's Gunning's fog index involves two distinct elements:

Firstly, the fog index is determined by the average number of words per sentence, computed as the total number of words in the document divided by the total number of sentences. This aspect acknowledges the common belief that lengthy sentences are harder to grasp.

Secondly, the fog index considers the occurrence of complex phrases within the document. A syllable refers to a unit of sound in a word that can be uttered independently. For example, the word "sentence" contains three syllables ("sen," "ten," and "ce"). Words with more than three syllables are considered complex, and an abundance of such terms reduces the document's readability.

For instance, in the sentence "The Gunning's fog index is based on the premise that using short sentences and simple words makes a document easy to understand," the fog index is computed as follows:

0.4 * (23/1) + (4/23) * 90 = 23

This calculation yields a fog index of 23 for the given sample sentence.

Suppose a user manual is intended for factory employees with an eighth-grade education level. In that case, it is imperative to ensure that the document's Gunning's fog index does not exceed 8 to facilitate comprehension.

7.6 Summary

- We covered the coding phases of the software development process in this chapter.
- The majority of software development companies create their coding standards and demand that their programmers follow them. Contrarily, coding standards offer broad advice to programmers

about excellent programming practices, but it is up to individual engineers to decide how to put the principles into practice.

- ❖ Compared to testing, code review is a more effective method of reducing faults since it identifies problems while testing identified failures.

7.7 Keywords

- **Code Review:** A model's code is reviewed once the module has been successfully compiled and any syntax problems have been eradicated. Code reviews are extremely low-cost methods of reducing coding errors and producing high-quality code. Typically, two sorts of reviews are performed on a module's code. Code inspection and code walkthrough are the two sorts of code review approaches.
- **Code Inspection:** The purpose of code inspection is to find some common mistakes brought about by oversight and bad programming. To put it another way, unlike code walkthroughs, where code execution is manually simulated, code inspections look for certain types of problems in the code.

  Code Walkthrough: A code walkthrough is an informal method of code analysis typically used after a module has been encoded, successfully compiled, and any syntax mistakes have been rectified.

  Coding Standards: Coding standards are sets of guidelines specifying various conventions to be followed while coding, including aspects such as variable names, code layout, error return conventions, and more.

7.8 Self-Assessment Questions

1. Describe the difference between coding standards and coding guidelines, and explain the importance of establishing and enforcing these rules in a software development organization. List five essential coding standards and recommendations.
2. Clarify the variance between code inspection and code walkthrough, and compare the benefits of each approach.
3. Calculate the Fog index for this question and outline its significance in software documentation.
4. Define the disparities between external and internal documentation for a software product, emphasizing their significance to the organization.

5. Identify the kinds of errors detectable during a code walkthrough.

7.9 Case Study

Introduction: In the realm of software engineering, adhering to coding standards is crucial for ensuring code quality, readability, maintainability, and collaboration within development teams. This case study delves into the challenges associated with inconsistent coding practices and proposes recommendations to address these issues effectively.

Problem: A software development team grapples with several challenges stemming from inconsistent coding practices and a lack of adherence to coding standards:

1. Inconsistent code formatting: Developers follow disparate formatting conventions, leading to code that is hard to read and comprehend. Inconsistent indentation, spacing, and naming conventions impede codebase maintenance and modification.
2. Lack of code documentation: The absence of consistent code documentation makes it challenging for others to understand its purpose, functionality, and usage, hindering code reuse and collaboration.
3. Inefficient code organization: The codebase lacks proper modularization and organization, resulting in monolithic code files or classes. This complicates code navigation and maintenance, diminishing productivity and increasing error-proneness.

7.9 Recommendations:

1. Define and communicate coding standards: Establish a set of coding standards delineating guidelines for code formatting, naming conventions, documentation, and organization. Ensure all team members are familiar with these standards and comprehend their significance.
2. Automate code formatting: Employ automated tools to enforce consistent code formatting throughout the project. Tools like Prettier or ESL int can automatically format the code base according to the defined coding standards.
3. Provide code review and feedback: Foster a culture of code reviews wherein team members assess each other's code for adherence to coding standards. Utilize code review tools to offer feedback and suggestions for improvement.
4. Create code documentation templates: Develop templates or guidelines for documenting code, encompassing function/method descriptions, parameter details, and usage examples. Make code documentation a standard part of the development process.

5. Implement modular code organization: Encourage the adoption of modular programming techniques and design patterns to promote code organization and maintainability. Decompose complex code into smaller, reusable modules with well-defined responsibilities.
6. Conduct coding standard training sessions: Conduct training sessions or workshops to educate developers on the importance of coding standards and how to implement them effectively. Cover topics such as code formatting, documentation, and code organization.
7. Establish code quality metrics: Define and monitor code quality metrics such as code coverage, complexity, and adherence to coding standards. Utilize code analysis tools to identify areas needing improvement and measure progress.
8. Continuously improve coding standards: Regularly review and update coding standards based on evolving industry best practices and team feedback. Encourage developers to propose enhancements and contribute to refining coding standards.

Implementation of these recommendations empowers the software development team to enhance code quality, readability, and maintainability by enforcing consistent coding practices. Adherence to coding standards fosters improved collaboration, reduced errors, and enhanced overall efficiency in the development process.

7.11 References:

- I. Sommerville: Software Engineering, 9th Edition, Pearson Education, 2017.
- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.
- Roger S. Pressman: A Practitioner's Approach, 7th Edition, McGraw Hill Education, 2009.

Unit -8

Validation and Verification

Learning Objectives:

- Exploring the Testing Lifecycle
- Unveiling Black Box Testing
- Boundary Value Analysis Demystified
- Test-First Development Unveiled
- Understanding Testing Variants

Structure:

Introduction

A program undergoes testing by subjecting it to a set of test inputs, also known as test cases, to observe its behavior and verify if it functions as expected. During this process, any conditions leading to failure are documented for further debugging and adjustment if the program deviates from its intended behavior.

The primary objective of the testing procedure is to identify every defect present in the software application. However, even after completing the testing phase for most real-world systems, it remains impossible to guarantee that the program is entirely free of errors. This limitation stems from the vast input data domain inherent in many software applications. Testing cannot cover every possible configuration that input data might assume.

Despite this inherent limitation, the significance of testing cannot be overstated. It serves as a crucial mechanism for uncovering weaknesses in a software program. Thus, testing serves as an effective means of reducing system flaws and enhancing customer trust in the built system.

The testing process has two primary aims:

1. To validate to both the software's creators and users that it meets their criteria. This means that every requirement outlined in the requirements document for customized software should undergo at least one test. For generic software products, this entails testing all system features and combinations of features intended for inclusion in the final release.
2. To pinpoint instances where the software's behavior deviates from expectations, is undesirable, or fails to adhere to its specifications. These deviations typically result from programming errors. Defect testing aims to identify and rectify unfavorable system behaviors such as crashes, unintended interactions with other systems, inaccurate calculations, and data corruption.

8.1 Verification and Validation

Validation: Are we building the right product?' 'Verification: Are we building the product right?'

Validation involves ensuring that a fully developed system aligns with its requirement specifications, while verification focuses on confirming whether the output of each stage of software development matches that of the preceding phase. Therefore, validation aims to guarantee an error-free result, while verification concentrates on detecting problems during each phase.

The verification and validation processes center on confirming that the software being constructed adheres to its specifications and delivers the expected functionality desired by the stakeholders funding the project. These checks commence as soon as the requirements are established and persist throughout the development lifecycle. Verification verifies that the software meets its declared functional and non-

functional requirements, while validation ensures that the program meets client expectations, extending beyond compliance with specifications to fulfill customer needs. Validation is critical because requirements specifications may not always accurately reflect the true desires or preferences of system users.

The overarching goal of verification and validation methods is to ascertain that the software system is "fit for purpose," meaning it is suitable for its intended use. The required level of confidence is determined by the system's purpose, user expectations, and the prevailing market environment.

8.2 Fault and Failure

Fault

A Fault is caused by a mistake made by a developer during any of the development operations and is among the various types of errors that might occur in a program. For instance, calling the incorrect function is one example of an error. In the field of program testing, the terms error, fault, bug, and defect are considered synonymous, although the program testing community may use them interchangeably.

However, it's important to note that the term "fault" has a slightly different connotation in the realm of hardware testing than the terms error and bug.

Failure:

A program failure fundamentally signifies an incorrect behavior displayed by the program during its execution. An erroneous behavior is defined as either an incorrect outcome produced or an inappropriate activity performed by the program. Each failure is a consequence of a defect in the program, meaning every software failure can be traced back to some defect in the code.

The number of possible failure modes for a program is vast. Out of the numerous ways in which an application can fail, three instances are randomly selected.

8.3 Debugging

The following are some of the approaches popularly adopted by programmers for debugging:

Brute force method: This approach involves adding print statements throughout the program to print intermediate values, hoping that some of these values will aid in identifying the incorrect statement. When combined with a symbolic debugger, the technique becomes more systematic, as the values of different variables can be easily verified, and breakpoints and watchpoints can be set conveniently to test variable values.

Another variant of this approach is single-stepping with a symbolic debugger, where the developer mentally computes the desired result after each source instruction and checks whetherit is computed by single-stepping through the program.

Backtracking: This is also a frequently used strategy. Starting with the statement where an error symptom is seen, the source code is traced backward until the error is discovered. However, as the number of source lines to be traced back grows, so does the number of alternative backward paths, which can become unmanageably enormous for complicated programs, limiting the applicability of this approach.

Cause elimination method: Once an error has been identified, the signs of the failure (e.g., an attribute has an incorrect value when it should be positive, etc.) are logged. Based on the failure symptoms, the causes that may have contributed to the symptom are determined, and tests are performed to rule these out. The fault tree analysis is an associated method to recognize the error from the error symptom.

Programming slicing: This method is comparable to backtracking. However, by establishing slices, the search space is minimized. A program slice for a specific variable and at a specific statement is the collection of source lines before this statement that potentially influences the value of that variable. The notion that an inaccuracy in the value of a variable might be produced by the statements on which it is data dependent is exploited by program slicing.

8.4 Types of Testing

A commercial software system typically undergoes three levels of testing:

1. **Development testing:** This phase involves testing the system to identify flaws and defects. The testing process usually includes system designers and programmers who aim to uncover any issues within the system. Development testing ensures that the software meets the specified requirements and functions as intended.

2. **Release testing:** In this stage, an independent testing team evaluates the entire system before it is released to users. The primary objective of release testing is to ensure that the system meets the requirements of system stakeholders and functions properly under various conditions. The testing team verifies that the software is stable, reliable, and ready for deployment.
3. **User testing:** User testing involves users or potential users testing the system in their respective environments. The term "user" can refer to internal marketing groups who assess whether the software is ready for advertisement, release, and sale. Acceptance testing, a type of user testing, occurs when the customer formally evaluates the system to determine if it should be accepted from the system supplier or if further development is necessary.

8.4.1 Development Testing

Testing can be conducted at three different levels of detail during development:

A. Unit Testing: Unit testing involves testing individual program units or object classes. The functionality of objects or methods should be tested during unit testing. After a module has been coded and properly reviewed, unit testing begins. This phase tests various units (or modules) of a system in isolation. To test a single module, a complete environment that includes everything required for module operation is necessary. Modules required to provide the required environment are typically not available until they have also been unit tested; stubs and drivers are designed to offer the complete environment for a module.

B. Integration Testing: Integration testing occurs after all (or at least some) of the modules have been unit tested. The goal of integration testing is to find flaws in module interfaces (call parameters) and ensure that the various modules of a program properly interface with one another. During integration testing, several system elements are integrated in a planned manner using an integration plan. The integration plan details the steps and order in which modules are joined to realize the entire system.

C. Performance Testing: Performance testing is an essential component of system testing. It is performed to ensure that the system meets the non-functional requirements listed in the SRS document. Several types of performance testing correlate to different types of non-functional requirements. The forms of performance testing to be performed on a system depend on the various non-functional requirements of the system as described in its SRS document.

Performance testing includes stress testing, volume testing, configuration testing, compatibility testing, regression testing, recovery testing, maintenance testing, documentation testing, and usability testing.

Each of these testing methods serves a specific purpose in ensuring the quality and reliability of the software being developed.

8.4.2 Release Testing

During the development process, release testing and system testing have two key distinctions:

1. **Team Involvement:** Release testing should be conducted by a separate team that was not involved in system development. This helps in gaining an unbiased perspective and ensures that the system is tested from an external viewpoint. On the other hand, the development team's system testing focuses on finding faults in the system (defect testing).
2. **Testing Goals:** The goal of release testing is to ensure that the system meets its requirements and is fit for external usage. This is also known as validation testing. In contrast, the system testing conducted by the development team aims to find defects in the system and ensure its internal functionality.

Black Box Testing:

Black box testing is a method where test cases are derived from the system specification without any knowledge of its internal design or coding. Two common approaches for designing black box test scenarios are Equivalence Class Partitioning and Boundary Value Analysis.

Equivalence Class Partitioning:

This approach divides the input values into equivalence classes, ensuring that the program's behavior remains consistent within each class. The criteria for creating equivalence classes include defining valid and invalid ranges for input values.

Example: For a program computing the intersection point of two straight lines, equivalence classes include parallel lines, intersecting lines, and coincident lines. Test cases are then selected from each class, such as (2, 2), (2, 5), (5, 5), (7, 7), and (9, 9).

Boundary Value Analysis:

Boundary value analysis focuses on testing input values at the boundaries of different equivalence classes. This helps uncover programming errors that often occur at these boundaries due to oversight or incorrect processing.

Example: Test cases for a function computing the square root of integers between 0 and 5000 should include values like 0, -1, 5000, and 5001.

These testing methods contribute to ensuring the quality and reliability of the software under development.

White Box Testing:

White-box testing is a critical aspect of unit testing, offering various methodologies to ensure thorough testing of the code base. These methodologies focus on analyzing the internal structure of the code and designing test cases accordingly.

Fault-Based Testing:

Fault-based testing techniques aim to uncover specific types of defects by focusing on a predefined fault model. For example, mutation testing is a fault-based approach that involves introducing small changes (mutations) to the source code to evaluate the effectiveness of the test cases in detecting these mutations.

Coverage-Based Testing:

Coverage-based testing techniques aim to execute or cover specific program parts to ensure thorough testing. Common coverage metrics include statement coverage, branch coverage, multiple condition coverage, and path coverage. These methodologies focus on achieving comprehensive coverage of the code base to ensure that all code paths are exercised during testing.

A white-box testing strategy is considered stronger than another if it covers all program items covered by the weaker strategy and additionally covers at least one program element not covered by the weaker strategy.

User Testing:

In practice, there are three forms of user testing:

1. **Alpha Testing:** Users collaborate with the development team to test the product at the developer's site. This form of testing allows for early feedback and collaboration between users and developers.

2. **Beta Testing:** A release of the software is made available to users for experimentation. Users are encouraged to explore the software and report any faults they encounter to the system creators. Beta testing helps in gathering feedback from a wider user base before the official release of the software.
3. **Acceptance Testing:** Consumers test the system to determine its readiness for acceptance from system developers and deployment in the customer environment. Acceptance testing ensures that the system meets the requirements and expectations of the end users before deployment.

These user testing methods play a crucial role in ensuring that the software meets user expectations and is ready for deployment in real-world environments.

1. Acceptance:

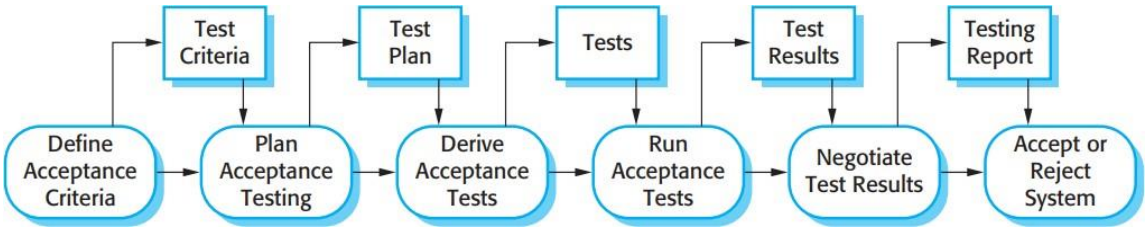

Figure 8.1: Acceptance Testing

The figure 8.1 illustrates the six phases of the acceptability testing procedure.

Set Acceptance Standards:

Before signing the system contract, it's crucial to establish acceptance criteria agreed upon by both the customer and developer. However, defining criteria at this stage can be challenging due to potential requirements changes as the project progresses.

Acceptance Testing for Plans:

This phase determines the resources, time, and budget allocated for acceptance testing, along with the testing schedule. The acceptance test plan should cover the required coverage of requirements and the sequence of evaluating system features. It should also address potential hazards to the testing process and strategies to mitigate them.

Construct Acceptance Tests:

Tests are designed to determine if the system meets the acceptance requirements, including its performance, functional, and non-functional aspects. These tests aim for comprehensive coverage of the system's needs, although creating wholly objective acceptance criteria can be challenging.

Conduct Acceptance Testing:

The system undergoes the agreed-upon acceptance tests. While testing in the actual usage environment is ideal, it may not always be feasible. Setting up a testing environment might be necessary, and automating this process can be challenging due to the involvement of end-user interactions.

Discuss Test Results:

It's unlikely that the system will function flawlessly and pass all defined acceptance tests without any issues. Therefore, the software engineer and client need to agree on whether the system is suitable for usage and how any discovered problems will be addressed.

Accept/Reject System:

During this phase, the customer and developers meet to decide whether to approve or reject the system. Further development may be necessary to address any identified issues before acceptance testing can be repeated.

8.5 Summary:

This chapter has emphasized the Testing phase of SDLC. Proper testing is crucial for identifying flaws in a program, although it cannot guarantee error-free software. Black-box and white-box testing are two common testing methods, with functional testing being a type of black-box testing. System testing includes functional and performance testing, ensuring compliance with requirements.

8.6 Keywords:

- Testing
- Development Testing
- Unit Testing
- Acceptance Testing
- Regression Testing

8.7 Self-Assessment Questions:

1. Integration testing involves testing the integration of different components or modules of a software product. Various techniques can be applied to integration testing, such as big-bang, top-down, bottom-up, and mixed approaches.
2. Performance testing evaluates the system's performance against specified non-functional requirements. Types of performance testing include stress testing, volume testing, configuration testing, compatibility testing, regression testing, recovery testing, maintenance testing, documentation testing, and usability testing.
3. Acceptance testing involves testing the software from the user's perspective to determine its readiness for deployment and use in the operational environment. Alpha testing involves users collaborating with the development team, beta testing involves users experimenting with the software, and acceptance testing involves formal testing to determine whether the software meets acceptance criteria. Test cases for these tests are created based on the specific objectives of each test, and they may differ in scope and focus.
4. Usability testing typically falls under system testing. Usability is tested by evaluating the user interface and ensuring it meets user needs and expectations. Techniques such as user surveys, heuristic evaluations, and usability testing sessions are used to assess usability.
5. Even if software has completed unit, integration, and system testing, it cannot be guaranteed to be error-free. Testing can only demonstrate the presence of defects, not their absence. Therefore, while comprehensive testing increases confidence in the software's quality, it does not eliminate the possibility of undiscovered defects.

8.8 Case Study:

Company XYZ faced challenges related to inadequate test coverage, test environment management, and test data management. Recommendations include prioritizing critical non-functional requirements, establishing consistent test environments, creating diverse test datasets, leveraging automation and tooling, and fostering collaboration and communication among stakeholders to address these issues effectively.

8.9 References

- Sommerville, I: "Software Engineering 9th Edition", Pearson Education, 2017.
- Rajib Mall: "Fundamentals of Software Engineering", 4th Edition, Prentice Hall of India, 2014.
- Roger S. Pressman: "A Practitioner's Approach", 7th Edition, McGraw Hill Education, 2009.

Unit: 9

Software Project Management

Learning Objectives:

- Determine a software project manager's job responsibilities and skills.
- Identify the essential project planning activities.
- Determine and appropriately order the various project-related estimations performed by a project manager.
- Define Software Project Management Plan(SPMP).
- Identify and describe two metrics for estimating the size of software projects. Recognise the various project-parameter estimating techniques.

Structure:

Introduction

The success of any software development project hinges on effective project management. Historically, many projects have fallen short due to poor management techniques rather than a lack of skilled engineers or resources. Therefore, understanding the latest techniques in software project management is crucial.

Software project management encompasses various strategies and skills aimed at facilitating teamwork among engineers. Before delving into project management techniques, it's essential to identify who should oversee project management. Typically, a project manager, often a seasoned team member, acts as the administrative supervisor, handling project leadership and technical direction for small projects. For larger projects, a different team member takes on the role of technical leader, responsible for decisions regarding tools, approaches, and problem-solving strategies.

This chapter begins by exploring why managing software projects is particularly challenging compared to other types of projects. It then outlines the primary duties of a software project manager, including project planning, scheduling, estimating, risk management, and configuration management.

9.1 Software Project Management

Software engineering inherently involves software project management due to organizational constraints like budget and schedule. The project manager's role is to ensure the project adheres to these limitations while delivering high-quality software. While good management doesn't guarantee success, poor project management often leads to late deliveries, increased costs, or unmet client expectations.

9.2 Roles and Responsibilities of Project Manager

Software project managers oversee overall project management and success, a role encompassing diverse tasks from team morale boosting to customer presentations. These tasks can be categorized into project planning and project monitoring and control activities. Project planning involves organizing tasks before development begins, while monitoring and control activities ensure adherence to plans and adjust as necessary during development.

Skills of Project Manager

Successful project managers require theoretical knowledge of project management methodologies, alongside qualitative judgment, decision-making skills, and effective

communication. Experience plays a crucial role in areas such as tracking project progress, client engagement, managerial presentations, and team development.

9.3Project Planning

Project planning begins after a project is deemed feasible and involves careful estimation of project characteristics such as cost, timeframe, and effort. Accurate estimations are crucial for subsequent tasks like scheduling, staffing, and risk management. Project managers also develop staffing plans, manage risks, and create miscellaneous plans like quality assurance and configuration management (Figure 9.).

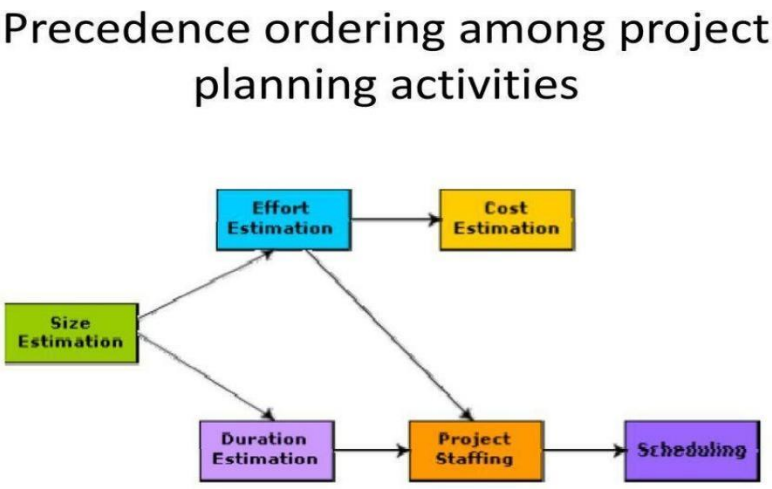

Figure 9.1: Project Planning Activities

9.3.1 Sliding Window Planning

Precise planning at the outset of a significant project is often challenging, especially when projects span several years. During such projects, parameters like scope and staff frequently change, derailing original plans. To address this, project managers adopt a staggered planning approach known as sliding.

At the project's onset, the manager has incomplete knowledge, which improves gradually as the project progresses through development phases. As understanding grows, project parameters are periodically re-estimated, allowing for more accurate planning.

9.3.2 The SPMP Document of Project Planning

Following project planning, project managers compile their ideas into a Software Project Management Plan (SPMP) document. The SPMP document covers various topics, including:

1. Introduction
2. Project Estimates
3. Schedule
4. Project Resources
5. Staff Organization
6. Risk Management Plan
7. Project Tracking and Control Plan
8. Miscellaneous Plans

This structured approach aids in organizing the SPMP document effectively.

9.3.3 Metrics for Project Size Estimation

Accurate estimation of project characteristics depends on correctly evaluating project size. Currently, Function Point (FP) and Lines of Code (LOC) are commonly used metrics for determining project size.

Lines of Code (LOC)

LOC is a straightforward statistic for project size but has limitations:

- It only measures coding activity, neglecting effort required for other lifecycle tasks.
- It depends on individual coding preferences and does not necessarily correlate with code quality.
- It penalizes the use of higher-level programming languages and code reuse.
- It measures lexical complexity but doesn't address logical and structural complexities.
- It's difficult to estimate accurately at the project's outset.

Function Point

Function Point metric, proposed by Albrecht, addresses many issues of the LOC metric. It's advantageous as it can determine software size according to problem definition and can be applied once the product is fully built.

9.4 Project Estimation Techniques

Estimating project characteristics like size, effort, time, and cost is essential for project planning. Estimation approaches fall into three categories: Empirical Estimating Method, Heuristic Methods, and Analytical Estimation Methods

Empirical Techniques

Empirical estimation techniques rely on educated guesses based on previous experience with similar projects. Expert judgment and Delphi cost estimation are two common empirical estimating methodologies.

Expert Judgment Method

Expert judgment involves an expert providing an educated approximation of the project's magnitude based on extensive analysis. While this method is common, it is susceptible to human errors and biases. Group estimation, where estimates are provided by a team of experts, reduces individual biases but may still be influenced by group dynamics.

Delphi Cost Estimation

The Delphi cost estimation method involves a team of experts and a coordinator. Estimators anonymously submit their estimates to the coordinator, who then compiles and share a summary of responses. This process is repeated multiple times to refine estimates and minimize individual biases.

Heuristic Techniques

Heuristic approaches describe relationships among project parameters mathematically. These models can be single-variable or multivariable. Single-variable models use previously estimated basic features to estimate additional parameters, while multivariable models consider multiple independent parameters to provide more accurate estimates.

Analytical Estimation Techniques

Analytical estimation approaches start with basic assumptions about the project and derive results based on these assumptions. Examples include Halstead's software science, which is useful for estimating software maintenance efforts.

9.5 Scheduling, Staffing Estimation, Organization, and Team Structure

Scheduling

Scheduling project tasks involves determining when and by whom each task will be completed. Project managers identify major activities, break them down into tasks, determine task interdependencies, estimate task durations, and create an activity network to represent the information. Critical path analysis helps identify tasks critical to the project's duration.

Staffing Level Estimation

Determining the project's manpower requirements involves understanding patterns such as the Rayleigh distribution curve, as observed by Norden. Putnam's research further investigated software project staffing, relating the number of lines of code to effort and time using the Rayleigh-Norden curve.

Organization and Team Structure

Software development organizations can be structured functionally, project-wise, or matrix-wise. In a functional format, development personnel are grouped by functional areas, while in a project format, teams are organized around specific projects. Matrix format combines elements of both functional and project structures, allowing for flexibility in resource allocation.

Functional Format vs Project Format

Functional and project formats offer different approaches to organizing development teams. Functional format groups engineers based on functional areas, while project format organizes teams around specific projects, borrowing resources as needed (Figure 9. 2).

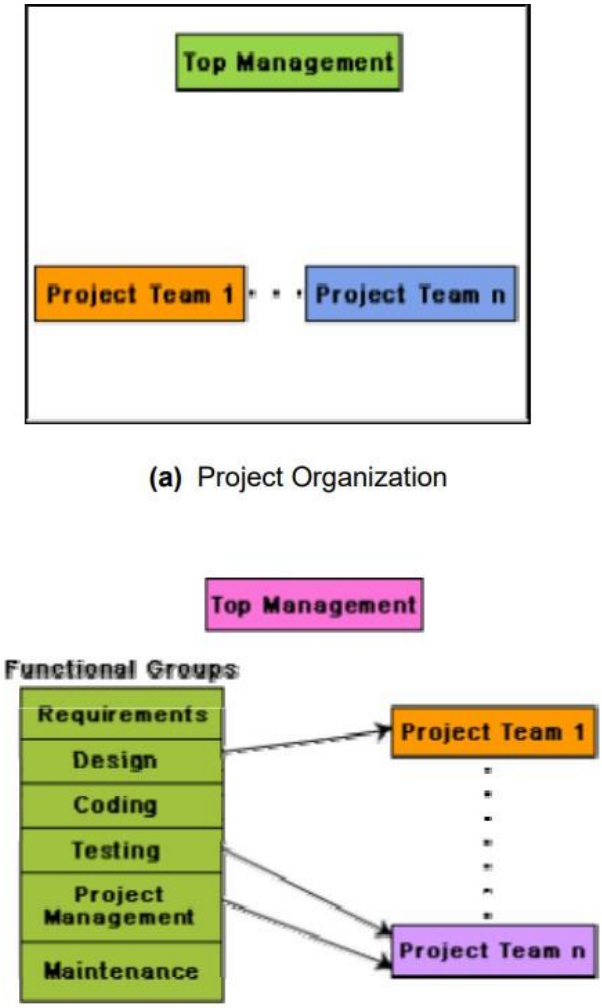

Figure 9. 2: Functional and Project Organization

In a functional style, different teams of programmers handle distinct phases of a project. For example, one team may focus on requirements specification, another on design, and so forth. As the project advances, the partially completed product is transferred from one team to the next. Consequently, this functional structure requires extensive communication among teams, as each successive team must fully understand the work of the preceding teams. This underscores the importance of creating high-quality documentation following each activity.

In contrast, the project format involves assigning a group of engineers to the project at its inception, and they remain involved until the project is completed. Consequently, the same team

oversees every aspect of the project's lifecycle. Compared to the project format, the functional model demands more inter-team communication since each team must comprehend the work done by preceding teams.

A. Matrix Organisation Structure:

	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Figure 9.3: Matrix Organization structure

A matrix organization aims to blend the advantages of both project and functional structures. In this setup, the pool of functional specialists can be reassigned to different projects as necessary. Thus, a matrix illustrates how various functional specialists are allocated to different projects. Consequently, in a matrix structure, the project manager must delegate authority over the project to various individual functional managers (Figure 9.3).

Depending on the balance of power between functional managers and project managers, matrix organizations can be categorized as strong or weak. In a strong functional matrix, functional managers have the authority to assign employees to projects, and project managers must accept the individuals they are given. Conversely, in a weak matrix, the project manager has control over the budget, the authority to reject employees from functional groups, and even the option to hire external employees.

Two significant issues often encountered in a matrix organization are:

B. Conflict over staffing arises between the functional manager and the project managers. C. Workers who are in firefighting mode are frequently shifted as emergencies develop in other projects.

Team Structure

Team structure pertains to the organization of individual project teams. There are several ways project teams might be organized, with three primary formal team structures: head programmer, democratic, and mixed team organizations, among others. Different team arrangements are often necessary for solving problems of varying complexities and sizes.

a. **Chief Team Structure**

In a chief team structure, a senior engineer acts as the team's principal programmer and provides technical leadership. The lead programmer divides tasks into manageable portions, assigns them to team members, and consolidates and verifies their outputs. While effective for well-understood problems due to the lead programmer's authority, this arrangement can stifle creativity and morale since team members are constantly under the chief programmer's supervision. Additionally, the chief programmer team is susceptible to single-point failure due to the excessive authority and responsibility placed on the lead programmer.

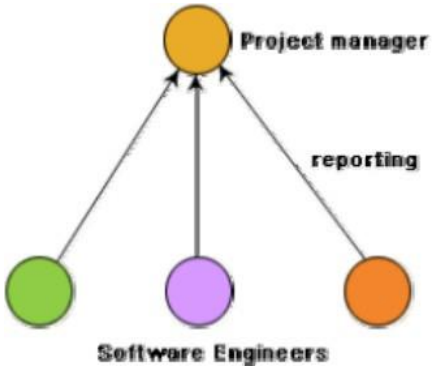

Figure 9.4: Chief Team Structure

This group is arguably the most effective at finishing straightforward and short tasks because the lead programmer can come up with a workable design and instruct the programmers to code

various modules of his design solution. Let's say a company has completed numerous straightforward MIS initiatives. The chief programmer team structure can then be used for a subsequent MIS project of a similar nature. When a singlepersonhasthenecessaryintellectualcapacitytocompletetheassignment, the chief programmer team structure performs well. However, even for straightforward and well-understood issues, a company must exercise caution while implementing the chief programmer structure. The chief programmer team structure shouldn't be used until early project completion is more important than other considerations like team morale, individual growth, life-cycle cost, etc.

a. Democratic Team:

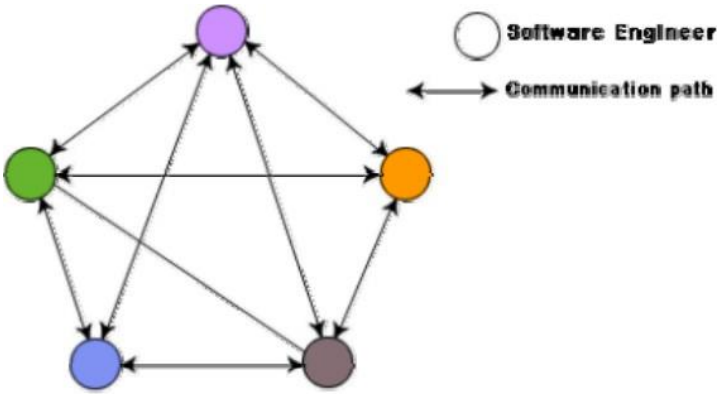

Figure 9.5: Democratic Team Structure

As the name suggests, this team structure does not impose a formal team hierarchy. An administrator typically provides leadership in this area. Different individuals take on different technical leadership roles at various points. A democratic workplace fosters more motivation and job satisfaction.

As a result, it experiences decreased staff turnover. Additionally, a democratic team structure is excellent for less well-known issues because a team of engineers is more likely to come up with answers than a single person, as in a chief programmer team. For projects needing fewer than five or six engineers, as well as projects focused on research, a democratic team organization is appropriate. Purely democratic organizations frequently degrade into chaos when dealing with

large-scale tasks. Programmers can exchange and critique one another's work in a democratic team environment, which promotes egoless programming (Figure 9.5).

b. Mixed Control Team Organization:

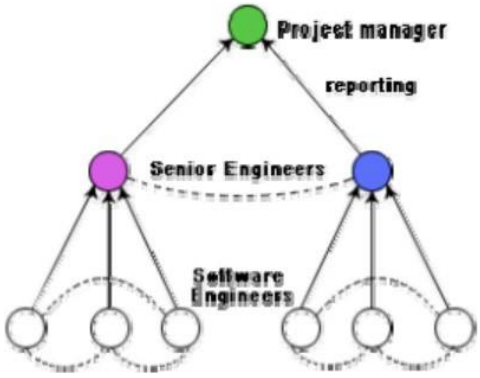

Figure 9.6: Mixed Team Structure

This team incorporates concepts from both the chief-programmer organization and the democratic organization. This team's organisational structure combines democratic principles with hierarchical reporting. Figure 9.6 illustrates the reporting structure using solid arrows and the democratic links using dashed lines. For big team sizes, the mixed control team organization works well. The senior engineers' democratic system is employed to break down the issue into manageable chunks. At the programmer level, each democratic arrangement makes an effort to solve a specific problem .

Therefore, this team structure is ideally suited to manage extensive and intricate programs. This team structure is often utilized in software development firms and is very well-liked.

9.6 Risk Management

A wide range of risks can impact a software project. To systematically identify significant risks, they must be classified into categories. Three types of risks can harm a software project:

a. **Project Risk**: Project risks encompass budgetary, schedule, personnel, resource, and customer-related issues. The risk of project schedule delay is significant due to the intangible nature of software, making monitoring and controlling extremely challenging. Unlike in manufacturing projects where progress is visible, the invisibility of the product under development contributes to schedule slippage risks.

b. **Technical Risk**: Technical risks involve potential issues with design, implementation, interfacing, testing, and maintenance. Ambiguous or incomplete specifications, changing requirements, technical uncertainty, and obsolescence are examples. Most technical risks stem from a lack of understanding among the development team.

c. **Business Risks**: These risks involve the possibility of developing a product that no one wants, as well as the risk of losing budgetary or personnel commitments.

9.7 SCM

Software Configuration Management (SCM) involves efficiently managing a software's configuration throughout its lifecycle. The configuration represents the status of all project deliverables at any given time.

Software Configuration Management Activities

Configuration management is typically handled by a project manager using automated tools. These tools aid in resolving challenges and tracking the state of deliverables, facilitating controlled changes to components. Configuration management involves two main activities: configuration identification and configuration control.

Software Configuration Management Tool

SCCS and RCS are popular configuration management tools found on UNIX systems. They are used to control and manage various versions of text files efficiently. However, they do not support binary files and rely on saving deltas to minimize disk space usage.

9.8 Summary

In this chapter, the main duties of a project manager for software were explored, categorized into project planning and project monitoring and control. Project planning involves activities such as personnel, scheduling, estimating, risk analysis, configuration planning, and process customization. Project monitoring and control are crucial once development begins, relying on systematic techniques and subjective judgment.

9.9 Keywords

- **Team Structure**: Different arrangements of project teams, including democratic, chief programmer, and mixed control teams, are crucial for effective teamwork depending on project complexity.

- **SCM**: Software Configuration Management ensures efficient management of a software's configuration throughout its lifecycle.
- **Risk Management**: Identifying and mitigating potential negative events or situations that could impact project progress.
- **Staffing**: Selecting quality software engineers is essential for project success.

9.9 Self-Assessment Questions

1. Putnam's model can be used to calculate the change in project cost as project time increases, but it has drawbacks such as oversimplification and sensitivity to parameter values.
2. Adding extra labor to an already delayed project can cause further delays due to communication overhead, task dependencies, and resource contention.
3. Project scheduling tasks include identifying activities, breaking them into tasks, determining interdependencies, estimating time, creating an activity network, identifying the critical path, and assigning resources.
4. Different organizational structures like functional, project-wise, and matrix-wise have their pros and cons. For a complex satellite-based mobile communication product, a project-wise structure may be suitable to ensure focused efforts and clear accountability.
5. Choosing between democratic and chief programmer team organizations depends on project complexity and team dynamics. For a large software product development team, a democratic structure might foster collaboration and creativity, while a chief programmer structure could provide clear leadership but might stifle creativity.

9.11 Case Study

Problem: Company XYZ faced challenges due to lack of clear project goals and scope, inefficient resource allocation, poor communication, and collaboration.

Recommendations:

1. Clearly define project goals and scope through thorough requirements analysis and stakeholder collaboration.
2. Efficiently allocate and utilize resources by conducting resource planning and implementing project management tools.
3. Foster effective communication and collaboration through clear channels and regular status meetings.

4. Implement project management methodologies and tools to plan, track, and monitor project tasks and progress.
5. Regularly monitor project progress and manage risks proactively through robust project monitoring and risk management practices.

Conclusion: By implementing these recommendations, Company XYZ can improve its software project management practices, leading to successful project delivery, improved outcomes, and increased customer satisfaction. Learning from challenges and applying effective project management strategies is essential for project success.

9.12 References

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- Summerville, I: Software Engineering 9th Edition, Pearson Education, 201723
- Roger S. Pressman: A Practitioner's Approach, 7th Edition, McGraw Hill Education, 2009
- Craig Larman: Applying UML and Patterns An introduction OOAD and the Unified Process, 3rd Edition, Pearson Education, 2015

Unit: 10

Software Maintenance

Learning Objectives:

- Describe why software maintenance is essential and list the various kinds of software maintenance.
- Describe what software reverse engineering entails.
- What are products made with legacy software? Determine the issues with their up keep.
- List the variables that affect software maintenance operation and identify the software maintenance process models.
- Describe what software reengineering entails. Calculate an approximation of a software product's maintenance costs.

Structure:

Introduction

Many software organizations are beginning to priorities software maintenance as a key task. Given the velocity of hardware obsolescence, the inherent longevity of software products, and the user community's hunger for the existing software products to operate on newer platforms, run in newer contexts, and/or have increased capabilities, this is not surprising. Maintenance is required when a software product handles some low-level tasks and the hardware platform is modified. Additionally, whenever a software product's support environment changes, it must be updated to accommodate the newest interface. For instance, when the operating system changes, a software product may require maintenance. Therefore, through maintenance activities, every software product continues to develop after its initial development.

10.1 Characteristics of "Software Maintenance"

In this section, we categorize various maintenance efforts into a few classes and discuss some general traits and unique issues related to maintenance initiatives. Many firms are prioritizing software maintenance due to the rapid obsolescence of hardware, the longevity of software products, and the user community's demand for software products to operate on newer platforms or have increased capabilities.

Software maintenance can be broadly divided into three categories:

1. **Corrective Maintenance**: This type of maintenance aims to fix flaws or defects found while the software system is in use. It involves identifying and addressing issues reported by users or discovered during testing.
2. **Adaptive Maintenance**: When clients require a software product to run on new platforms, integrate with new hardware or software, or adapt to changes in the operating system, adaptive maintenance becomes necessary. It involves modifying the software to ensure compatibility with the evolving environment.
3. **Perfective Maintenance**: Perfective maintenance is aimed at improving the software by updating its functionalities, adding new features, or enhancing existing ones based on user feedback or changing requirements. This type of maintenance focuses on optimizing the software's performance and user experience.

Through maintenance activities, every software product continues to evolve and meet the changing needs of users and technological advancements.

10.2 Software Reverse Engineering

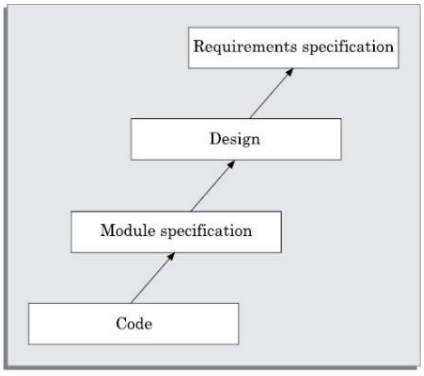

Figure10.1: Process Model for Reverse Engineering

Software reverse engineering is a technique aimed at extracting the design and requirements specification of a product's code through investigation. The main goals of reverse engineering are to create documentation for legacy systems and to improve system understandability to facilitate maintenance work. As older software products often lack structure and documentation, reverse engineering becomes crucial for their maintenance (Figure10.1).

During the initial stage of reverse engineering, aesthetic improvements are typically made to the code without altering its functionality. This involves enhancing code readability, structure, and understanding. Pretty printer programs can be utilized to reformat the code and improve its layout. Legacy software systems often have complex control structures and unclear variable names, making comprehension difficult. Assigning descriptive names to variables, data structures, and functions can greatly improve code documentation and readability. Additionally, complex nested conditionals can be simplified using simpler conditional statements or case statements to enhance clarity and maintainability.

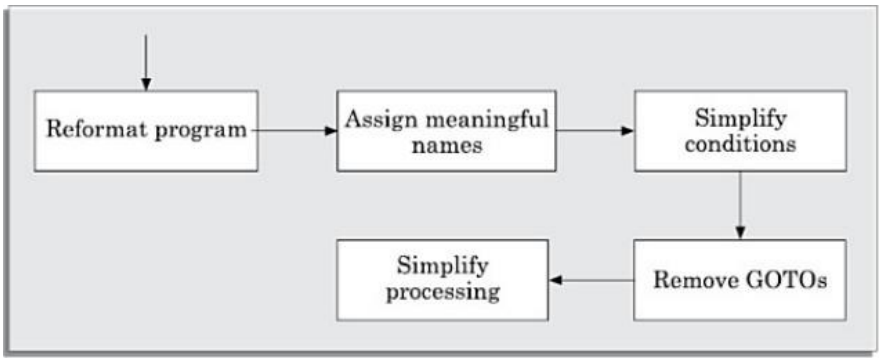

Figure 10.2: Cosmetic Changes before reverse engineering

After implementing cosmetic changes to legacy software, the next step involves extracting the code, design, and requirements specification. This process is schematically depicted in Fig. A comprehensive understanding of the code is necessary to extract the design. Technologies exist that can automatically generate dataflow and control flow diagrams. It is also essential to extract the structure chart, which illustrates module invocation sequences and data transfer between modules. Once the entire code has been thoroughly understood and the design extracted, the Software Requirements Specification (SRS) document may be prepared (Figure 10.2).

10.3 Maintenance Process Model

Two fundamental forms of software maintenance process models can be delineated. The first model suits small rework projects, wherein code updates are promptly executed, and corresponding documentation is subsequently amended. The maintenance process is illustrated graphically in the figure. Commencing with the collection of adjustment needs, the project progresses to requirement analysis for devising code change strategies. In this phase, involving at least some members of the original development team proves beneficial in mitigating team turnover, particularly for projects entailing unstructured and inadequately documented code.

Having access to a functioning old system at the maintenance site significantly streamlines the workload for maintenance engineers. It provides them with a clear understanding of the old system's functioning, enabling them to compare the performance of their modified system with the original. Additionally, debugging the re-engineered system becomes easier as program traces from both systems can be compared to identify issues (Figure 10.3).

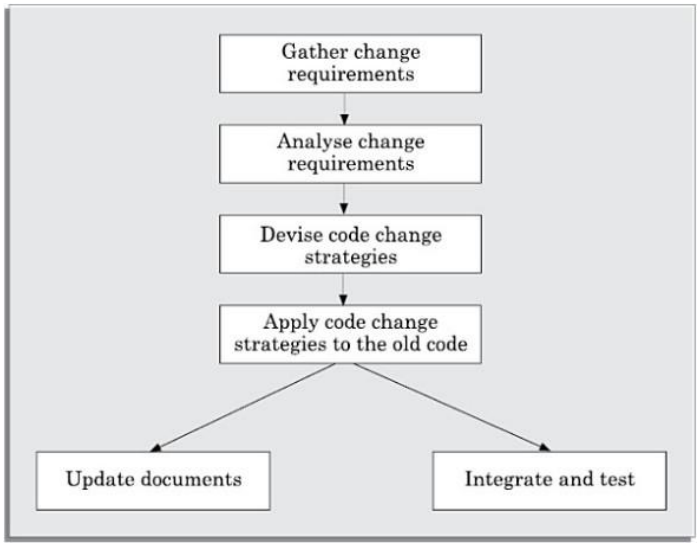

Figure 10.3: Maintenance Process Model-1

The second process model for software maintenance is appropriate for projects that require a considerablequantityofrework.Areverseengineeringcycleisfollowedbyaforwardengineering cycle to depict this method. This method is also known as software reengineering. The figure depicts this process model. For legacy products, the reverse engineering cycle is essential. The old code is analyzed (abstracted) during reverse engineering to retrieve module specifications. The module specs are then analyzed to create the design. The original requirements specification is created by analyzing (abstracting) the design. The change requests are then applied to the existing requirements specific action to produce the new requirements specification (Figure 10.4).

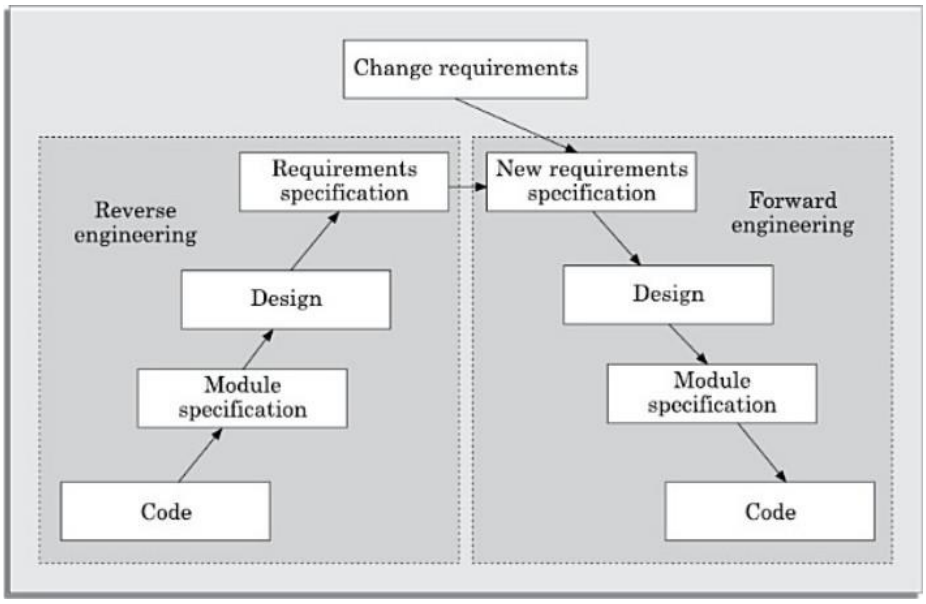

Figure 10.4: Maintenance Process model-2

Reverse-engineered components are extensively reused throughout the design, module specification, and coding phases. This approach offers a notable advantage as it yields a more structured design compared to the original product, along with improved documentation and often enhanced productivity.

A more efficient design is responsible for the efficiency gains. However, this strategy is more expensive than the first (Figure 10.5).

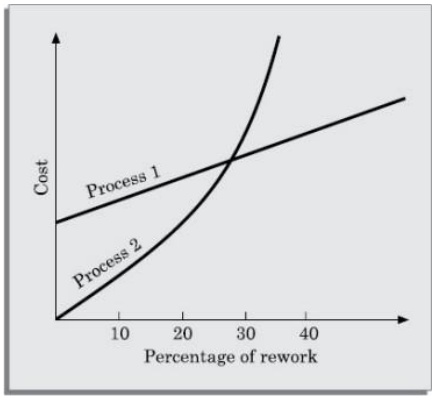

Figure 10.5: Empirical Estimation of Maintenance Cost and Rework

Based on empirical research, Procedure 1 is deemed superior when the rework quantity is below 15%. However, apart from rework quantity, various other factors can influence the choice between Process Model 1 and Process Model 2:

1. In cases of products exhibiting a high failure rate, reengineering may be the more favorable option.
2. Reengineering might be preferred for legacy products characterized by insufficient design and code structure.

10.4 Maintenance Cost Estimation

Maintenance efforts are generally known to consume approximately 60% of a typical software product's entire life cycle cost. However, maintenance costs vary greatly between application domains. Maintenance costs for embedded systems might be 2 to 4 times the development cost. As part of his COCOMO cost estimation model, Boehm [1981] provided a method for predicting maintenance costs. Boehm's maintenance cost estimation is based on a metric known as “Annual Change Traffic” (ACT).

Boehm defined “ACT as the percentage of a software product's source instructions that change in a normal year, either by addition or deletion”.

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

“$KLOC_{added}$ is the total number of kilo lines of source code added during maintenance”. “$KLOC_{deleted}$ represents the total number of KLOC destroyed during maintenance”.

As a result, altered code should be counted in both code added and code deleted. The maintenance cost is calculated by multiplying the annual change traffic (ACT) by the total

Development cost:

“maintenance cost=ACT*development cost”

Many maintenance cost estimation models yield only approximate results because they fail to consider various factors, such as engineers' expertise and familiarity with the product, hardware requirements, software complexity, and other relevant elements.

10.5 Summary

- ❖ We covered several fundamental principles related to software maintenance tasks in this chapter.
- ❖ Maintenance typically constitutes the most expensive phase of the software life cycle. Therefore, investing time and effort in building the product with a focus on maintainability can often prove cost-effective in reducing maintenance expenses.
- ❖ We talked about reverse engineering tasks and then two maintenance process models.
- ❖ We also spoke about how these two process models could be used in maintenance initiatives. In estimating maintenance projects, we highlighted the key points.

10.6 Keywords

- **Reverse Engineering:** Software reverse engineering is the technique of extracting from an investigation of a product's code the design and the requirements specification. Reverse engineering's two main goals are to create the necessary documentation for a legacy system and to make maintenance work easier by making a system more understandable. Since older software products are largely unstructured and lack sufficient documentation, reverse engineering is becoming more and more crucial. Even well-designed products eventually degrade due to repeated maintenance efforts, turning them in to legacy software.
- **Maintenance Cost:** It is common knowledge that the cost of maintenance activities accounts for roughly 60% of the overall cost of a typical software product's life cycle. However, maintenance costs differ significantly between different application domains.
- **Corrective Maintenance:** A software product needs corrective maintenance to fix any defects found while the system was being used.
- **Adaptive: "**When users require a software product to run on new platforms, new operating systems, or to integrate with new hardware or software, maintenance may be necessary".
- **Perfective: "**A software product requires maintenance to support the new features that consumers want it to support, to alter various system operations in response to user requests, or to improve the system's performance".

10.7 Self-Assessment Questions

1. What constitutes software reverse engineering, and what purpose does it serve within the software development domain?
2. What are the primary aims and objectives associated with software reverse engineering?
3. Could you elaborate on the distinction between static and dynamic techniques employed in reverse engineering?
4. What are some prevalent tools and methodologies utilized in software reverse engineering?
5. What ethical and legal considerations must be taken into account when engaging in software reverse engineering practices?
6. How is software reverse engineering utilized in the context of modernizing legacy systems?
7. In what capacity does reverse engineering contribute to cybersecurity and the analysis of vulnerabilities?
8. Can you provide examples or case studies demonstrating the practical applications of software reverse engineering?
9. What obstacles and constraints typically arise during software reverse engineering endeavors?
9. How does software reverse engineering support efforts related to software maintenance and improvement?

10.8 Case Study

Introduction: Company ABC is a software development company that recently faced challenges in managing software maintenance and controlling maintenance costs. Let's explore their case, address some questions, provide recommendations, and conclude with the key take aways.

Problem:

1. **Increasing maintenance costs**: Company ABC experienced a steady increase in software maintenance costs over time. The cost of fixing defects, enhancing functionality, and adapting to changes became a significant portion of their overall budget, impacting profitability.
2. **Inefficient bug tracking and resolution**: The company faced challenges in effectively tracking and resolving software defects. This resulted in longer resolution times, increased customer dissatisfaction, and higher maintenance costs.
3. **Lack of proactive maintenance**: Company ABC primarily relied on reactive

maintenance, addressing issues as they arose. This approach led to a backlog of unresolved issues and limited focus on preventive maintenance, resulting in recurring problems and additional costs.

Recommendations:

1. **Implement effective bug tracking and resolution processes:**

- Utilize bug-tracking tools to capture and prioritize software defects. Assign clear responsibilities and timelines for bug resolution.
- Establish a well-defined process for bug triaging, categorization, and resolution. Regularly communicate progress and updates to stakeholders.
- Conduct root cause analysis to identify underlying issues causing defects and implement preventive measures.

2. **Proactive Maintenance and Continuous Improvement:**

- Implement a preventive maintenance strategy, including periodic code reviews, refactoring, and performance optimization.
- Regularly assess software functionality and performance to identify potential issues and address them proactively.
- Encourage and prioritise user feed back to identify usability issues and implement improvements.

3. **Embrace automated testing and monitoring:**

- Implement automated testing frameworks to identify and prevent regression issues.
- Utilise continuous integration and deployment practices to automate testing and quickly identify defects.
- Monitor software performance and usage patterns to identify potential areas for improvement and prioritize maintenance efforts.

4. **Prioritize and manage change requests effectively:**

- Implement a change management process to handle change requests efficiently.
- Assess change requests based on their impact, benefits, and alignment with business goals before approving them.
- Communicate the impact of change requests on maintenance costs and ensure proper allocation of resources.

Conclusion: Software maintenance is a critical aspect of the software development lifecycle. By implementing effective maintenance strategies, such as proactive maintenance, efficient bug tracking and resolution, and automated testing and monitoring, companies can reduce maintenance costs, improve software quality, and enhance customer satisfaction. Company ABC can benefit from embracing these recommendations, as they provide a proactive approach to software maintenance, minimize recurring issues, and enable continuous improvement. By investing in preventive maintenance and addressing issues promptly, organizations can optimize maintenance costs and ensure the long-term success and sustainability of their software applications.

10.9 References:

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- Summerville I: Software Engineering 9thEdition, Pearson Education, 201723
- Roger S. Pressman: A Practitioner's Approach,7thEdition, McGraw Hill Education, 2009
- Craig Larman: Applying UML and Patterns, An introduction OOAD and the Unified Process, 3rd Edition, Pearson Education, 2015

Unit -11

Software Reliability

Learning Objectives:

- Identify the Challenges in Quantifying Software Reliability.
- Identify the Measures of Software Reliability.
- Types of Software Product Failures.
- Software Product Reliability Growth Models.
- Identify the Key Characteristics of ISO 9001 Certification.
- Explain the SEI CMMI Model's Core Process Areas for a Software Organization.

Structure:

Introduction

Customers not only desire extremely reliable items, but they may also demand a quantifiable guarantee of the product's reliability before purchasing various product categories. This demand is especially true for safety-critical and embedded software solutions. However, as we will describe in this chapter, determining the trustworthiness of any software product is quite challenging. One of the key challenges is being observer-dependent when trying to quantitatively assess a software product's reliability. That is, various groups of users may arrive at different estimations of product reliability. Additionally, various other issues, such as often fluctuating reliability values due to bug fixes, make accurately measuring a software product's reliability challenging. In this chapter, we delve into these concerns.

Although no completely appropriate metric for measuring software product dependability exists, we will cover some measures that are currently employed to quantify software product reliability. We'll also look at reliability growth modeling and how to anticipate when (and if) a particular level of reliability will be reached. Furthermore, we will investigate the statistical testing strategy to estimate dependability. We are additionally going through multiple elements of software quality assurance (SQA). In recent years, software quality assurance, also known as SQA, has become one of the most talked-about topics in software companies. SQA's main objective is to assist organizations in creating high-quality software products in a repeatable manner. When the company's software development process is person-independent, it is considered repeatable.

The success of a project is not dependent on any of the project's team members. SQA additionally handles important issues such as the quality of produced software and the cost of development.

11.1 Software Reliability

A software product's reliability mainly indicates its dependability or trust worthiness. Alternatively, a software product's dependability can be defined as the likelihood of the product working "correctly" over a specific length of time. A software package with an enormous amount of bugs is unreliable. However, there is no clear relationship between observed system reliability and the number of latent flaws in the system. For example, deleting mistakes from areas of the software that are infrequently performed has little effect on the product's perceived reliability.

It has been experimentally observed that a significant portion of a typical program's execution time is spent executing only a small fraction of the program's instructions, often referred to as the program's core. Removing a substantial number of product problems from the least used sections of a system would typically result in only a minor gain in product reliability. Therefore, a product's reliability depends not only on the quantity of latent faults but also on their precise placement and the execution profile of the product.

11.2 Reliability Metrics

Different categories of software products may have varying dependability requirements. Therefore, the level of reliability necessary for a software product must be described in the SRS (software requirements specification) document. Some measurements that can quantitatively indicate how reliable a software product is are required. A good reliability metric should rely on the observer, allowing diverse observers to reach an agreement on the system's reliability level.

Six reliability measures can be used to measure the reliability of software products:

- ROCOF (Rate of Occurrence of Failure)
- MTF (Mean Time To Failure)
- MTTR (Mean Time To Repair)
- MTBR (Mean Time Between Failures)
- POFOD (Probability of Failure on Demand)
- Availability

11.3 Reliability Growth Modelling

A reliability growth model is a mathematical model that depicts how software dependability improves as defects are discovered and corrected. Such models can be used to estimate when (or if) a specific level of reliability will be reached. Jelinski and Moranda Model is one such reliability growth model, which is a step-function model where reliability is considered to rise by a constant increment each time an issue is found and rectified. However, this simplistic model of reliability, which assumes that all errors contribute equally to reliability growth, is highly impractical (Figure11.1).

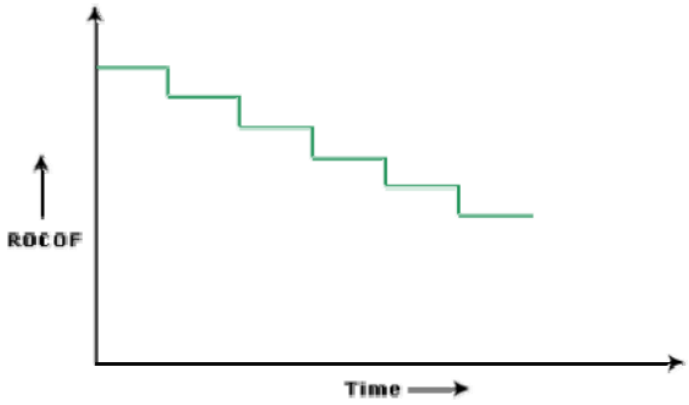

Figure 11.1 : Setup Function Model of reliability growth

The Model of Littlewood and Verrall

The model proposed by Littlewood and Verrall accommodates negative reliability growth, reflecting the reality that when a repair is performed, additional defects may be introduced. It also acknowledges that when errors are addressed, the average improvement in dependability per repair diminishes. This model considers the contribution of an error to improved reliability as an independent random variable with a Gamma distribution. This distribution represents the phenomenon where error fixes that contribute significantly to reliability growth are addressed initially. As the testing process advances, the rate of return diminishes (Figure11.2).

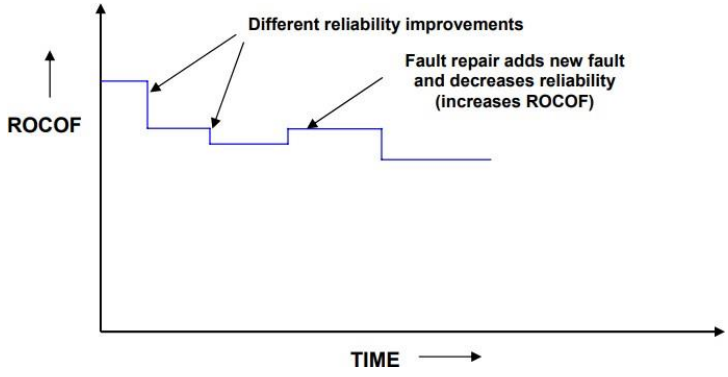

Figure11.2: Random Step Function Model of Reliability Growth.

11.4 Software Quality

A high-quality product is typically defined by its alignment with the purpose for which it was created. For software products, meeting the requirements outlined in the SRS document is often considered the benchmark for quality. However, the traditional definition of quality as "fitness for purpose" falls short when it comes to software. Simply fulfilling functional requirements may not suffice if the user interface is impractical or if the code is overly complex and unmaintainable. Therefore, the modern definition of quality encompasses various factors, including:

- **Portability**: Software portability refers to its ability to run easily across different operating systems, machines, and with other software products.
- **Usability**: A high-quality software product is user-friendly, catering to both expert and novice users.
- **Reusability**: Different parts of the software should be easily reusable to create new products.
- **Correctness**: The software should meet all the requirements specified in the SRS document.
- **Maintainability**: Defects should be easily repairable, new features should be integrable, and adjustments to functionality should be straightforward.

11.5 ISO 9001

The ISO 9000 series comprises three standards: 9001, 9002, and 9003, with ISO 9001 being the most relevant to software development organizations. It outlines requirements for organizations involved in designing, developing, producing, and servicing items. ISO 9001 focuses on several key areas:

- **Management Responsibility**: Management must implement an effective quality policy and ensure that roles and responsibilities impacting quality are documented.
- **Quality System**: A quality system must be maintained, documented, and kept up to date.
- **Contract Review**: Before entering into a contract, organizations must review it to ensure understanding and capability.
- **Design Control**: The design process must be well-regulated, and design changes must be managed.

- **Document Control**: Suitable protocols must be in place for document approval, distribution, and changes.
- **Purchasing and Purchaser-Supplied Product**: Purchased materials must be examined for compliance, and purchaser-supplied products must be controlled appropriately.
- **Product Identification**: The product must be identifiable at every stage of the process.
- **Process Control**: Processes must be adequately managed, and a quality strategy must include a list of quality requirements.
- **Inspection, Testing, and Nonconforming Product Control**: Testing must be effective, and nonconforming products must be controlled to prevent harm.
- **Corrective Action**: Errors must be rectified, and the system must be improved to prevent future occurrences.
- **Handling and Quality Records**: Proper handling, storage, and documentation of quality processes are essential.
- **Quality Audits**: Audits of the quality system must be performed to ensure effectiveness.
- **Training**: Training needs must be identified and provided to personnel.

11.6 SEI CMM and Six Sigma

The SEI Capability Maturity Model (CMM) and Six Sigma methodologies are aimed at improving the quality of software development and business processes, respectively. SEI CMM categorizes organizations into five maturity levels, ranging from initial ad hoc processes to optimized, continuously improving processes. On the other hand, Six Sigma focuses on optimizing processes to achieve near-perfection, with the goal of minimizing defects and improving efficiency. The DMAIC (Define, Measure, Analyze, Improve, Control) and DMADV (Define, Measure, Analyze, Design, Verify) processes are employed in Six Sigma projects to drive improvement and innovation.

11.7 Summary

This chapter has covered the significance of software reliability, the challenges in measuring it accurately, and various approaches to ensuring software quality, including ISO 9001 and SEI CMM. Additionally, the chapter discussed the key concepts of Six Sigma and its applicability to improving processes across different industries.

11.8 Keywords

- **Reliability**: The dependability or trustworthiness of a software product.
- **Reliability Metrics**: Quantitative measures used to assess the dependability of software products.
- **Software Quality**: The suitability of a software product for its intended purpose, considering factors beyond functional requirements.
- **Six Sigma**: A methodology aimed at optimizing processes to achieve near-perfection and minimize defects.
- **ISO 9000**: A series of standards providing guidelines for establishing quality management systems.
- **SEI CMM**: The Capability Maturity Model developed by the Software Engineering Institute to assess and improve an organization's software development processes.

11.9 Self-Assessment Questions

1. The responsibility for quality assurance efforts in a software development organization typically lies with the quality assurance (QA) team. Their primary tasks includes:
    - Developing and implementing standardized quality assurance processes.
    - Planning and executing testing methodologies to ensure software meets quality standards.
    - Tracking and managing software defects effectively.
    - Conducting regular audits and reviews to identify areas for improvement.
2. Ensuring that products are of high quality is the responsibility of various stakeholders within a software development organization, including:
    - Product managers: They define quality standards and requirements for products.
    - Development teams: They are responsible for implementing quality standards during the development process.
    - QA teams: They verify that the products meet quality standards through testing and validation.
    - Customer support teams: They gather feedback from customers and identify quality issues.
3. Six measures to assess programmer reliability could include:
    - Defect density: Number of defects found per lines of code.

- Mean time to failure: Average time between system failures.
- Failure rate: Number of failures experienced over a given period.
- Mean time to repair: Average time taken to fix defects.
- Code churn: Rate of code changes over time.
- Customer-reported issues: Number of issues reported by end-users. These measures provide insights into different aspects of reliability but may not fully capture all dimensions of system reliability, such as user experience and performance under stress conditions.

4. The Jelinski and Moranda Model and Littlewood and Verrall's Model are both reliability growth models used in software engineering. The former is based on the assumption that defects are removed at a constant rate during testing, while the latter allows for negative reliability growth to account for the introduction of defects during repairs. Each model has its advantages and disadvantages, and the choice depends on the specific characteristics of the software development process.
5. ISO 9001 certification and SEI CMM-based quality evaluation offer advantages and disadvantages:
    - ISO 9001 certification provides a standardized framework for quality management, enhancing credibility and marketability. However, it may be generic and not tailored specifically to software development processes.
    - SEI CMM offers a more targeted approach to improving software development processes, with a focus on maturity levels and continuous improvement. However, it may require significant resources and time to implement effectively.

11.10 Case Study

Introduction: Company XYZ, a software development company, faced challenges related to reliability and quality management in its software products. Let's explore their case, address some questions, provide recommendations, and conclude with the key takeaways.

Problem: Company XYZ experienced a high defect rate and frequent customer complaints about reliability and quality issues, negatively impacting customer satisfaction and trust.

Recommendations:

1. Implement comprehensive quality assurance processes:
    - Establish standardized processes for testing and defect tracking.
    - Incorporate quality checkpoints throughout the development lifecycle.

- Implement a defect tracking system to address issues effectively.

2. Focus on test automation:
    - Invest in automation frameworks to increase test coverage and efficiency.
    - Automate repetitive and critical test cases for consistent results.
3. Prioritize reliability testing:
    - Allocate dedicated resources and time for reliability testing activities.
    - Perform stress testing and monitor system behavior under different scenarios.
4. Continuously improve through feedback and data analysis:
    - Gather feedback from customers and analyze data to identify improvement opportunities.
    - Conduct regular retrospectives to review and improve quality management practices.

Conclusion: By implementing these recommendations, Company XYZ can enhance the reliability and quality of its software products, leading to improved customer satisfaction and long-term success. Prioritizing reliability and quality management is crucial for building robust and trustworthy software solutions.

11.11 Reference

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- Summerville, I: Software Engineering 9th Edition, Pearson Education, 201723
- Roger S.Pressman:APractitioner'sApproach,7thEdition,McGrawHillEducation,2009
- Craig Larman: Applying UML and Patterns An introduction OOAD and the Unified Process, 3rd Edition, Pearson Education, 2015

Unit -12

Computer Aided Software Engineering

(CASE) Tools

Learning Objectives:

- What does the CASE tool mean? Determine the main justifications for employing a CASE tool.
- What does the term "CASE environment" mean? Set off a CASE environment from a programming environment.
- Determine the characteristics of a CASE tool for prototyping.
- Determine the features that a decent CASE tool for prototyping should support.
- Determine the usual CASE tool supports that are required to carry out structured analysis and software design activities.
- Determine the assistance that CASE tools might provide during the code creation process.

Structure:

Introduction

We will discuss computer-aided software engineering (CASE) in this chapter and how using CASE tools can enhance software development. Input required for both development and maintenance. CASE has become a popular topic in the software industry recently. The most costly part of any computer system is now the software. Even though hardware costs are decreasing rapidly and exceeding even the highest expectations, software costs are rising due to increasing labor expenses. Most managers are concerned about this situation. In this scenario, CASE technologies offer to reduce costs and efforts associated with software development and maintenance. As a result, most software project managers have developed a passion for CASE tool deployment and development. CASE solutions promise to free software engineers from the monotony of repetitive tasks and to produce better products more quickly.

12.1 Scope of CASE

We need to first define what a CASE tool and CASE environment are. In a more limited sense, a CASE tool can indicate any instrument used to automate some activity related to software development. CASE is a general word used to refer to any type of automated support for software engineering. There are presently several CASE tools accessible. Some of these instruments help with phase-related tasks. Project management and configuration management are examples of non-phase tasks, as are specification, structured analysis, design, coding, testing, etc. The following are the main goals while utilizing any CASE tool:

- To boost productivity.
- To assist in the cheaper production of higher-quality software

12.2 Benefits of CASE

The use of a CASE environment, or even standalone CASE tools, offers several advantages. Let's explore a few of these advantages:

- **Cost Savings**: Throughout all phases of development, a major advantage of using a CASE system is cost savings. Various studies measuring the effects of CASE place the effort to decrease costs between 30 and 40 percent.

- **Quality Improvements**: The use of CASE tools results in significant quality improvements. This is mostly because iterating through the various stages of software development is simple, and the likelihood of human error is much diminished.
- **Reliable Documentation**: CASE tools support the creation of reliable documentation. Since the crucial information about a software product is kept in a single location, redundancy in the data is avoided, greatly reducing the likelihood of conflicting documentation.
- **Reduction in Tedium**: The majority of the tedious tasks in a software engineer's work are eliminated using CASE tools. For instance, they don't need to carefully examine the DFDs' balancing because they can do so quickly and easily by pressing a button.
- **Cost Reduction in Maintenance**: CASE tools have enabled radical cost reduction in software maintenance initiatives. This results from the systematic information collection that occurs during the many stages of software development as a result of adhering to a CASE environment, in addition to the enormous value of a CASE environment in traceability and consistency checks.
- **Organizational Impact**: The introduction of a CASE environment affects a company's working style and causes it to become more organized and structured.

CASE Environment:

Even though individual CASE tools are helpful, a tool set's full potential can only be realized when it is linked to a shared framework or environment. The stage or stages of the software development life cycle that they concentrate on define CASE tools. To have a consistent picture of the information related to the software development artifacts, it is necessary for various tools covering various stages to integrate through a central repository. The definitions of all composite and basic data items are often contained in this single repository, which is also known as a data dictionary. All CASE tools in a CASE environment exchange common data among themselves through the central repository. Therefore, a CASE environment makes it possible to automate the software development process' step-by-step approaches. The figure depicts a schematic illustration of a CASE environment (Figure 12.1).

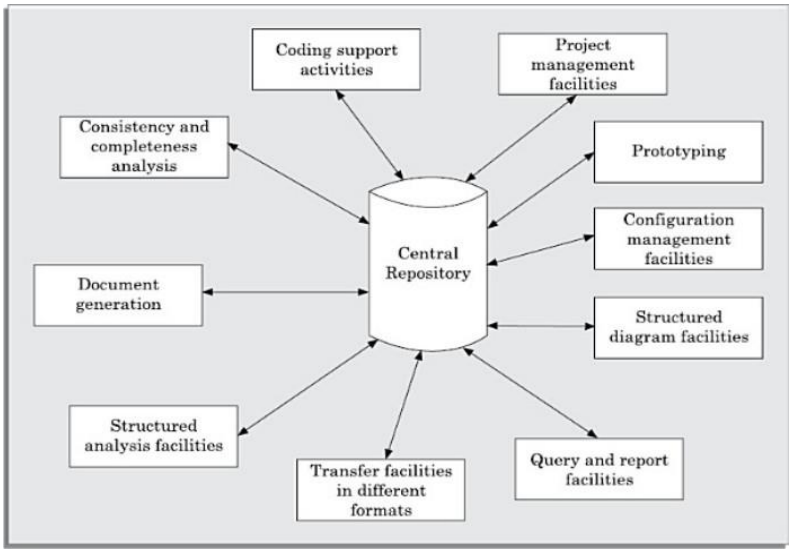

Figure 12.1: CASE Environment

A program editor, compiler, debugger, linker, etc., are all included in typical programming environments, such as Turbo C, Visual C++, etc. When you click on a compiler-reported error, the cursor is moved to the exact line or statement that the error is in, as well as into the editor.

12.3 CASE in the Software Life Cycle

Let's explore the various types of assistance that CASE offers throughout the various stages of a software lifecycle. A development methodology should be supported by CASE tools, which can also be used to enforce it and offer some consistency checks between different phases. The next section discusses some of the potential support that CASE tools may offer during the software development lifecycle.

Prototyping Support

Prototyping is beneficial for understanding the needs of complex software products, illustrating a notion, and marketing fresh concepts. A prototyping CASE tool should have the following prerequisites:

- User interaction definition.
- Establishment of the system's control flow.
- Data storage and retrieval requirements.
- Addition of logic to the processing.

Several independent prototyping tools are available. However, a tool that interfaces with the data dictionary can utilize the items within, assist in building the data dictionary, and guarantee consistency between the design data and the prototype. A decent prototyping tool should support the following features:

- Enable the user to design a GUI using a graphics editor.
- Compatibility with a CASE environment's data dictionary.
- Integration with external user-defined modules.
- User specification of the prototype's operations and order.
- Handling of input and output data management during mock-up runs.

Structured Analysis and Design

For structured analysis and design, a variety of diagramming approaches are employed. The CASE tool should support one or more of these methodologies and make it simple to create analysis and design diagrams. It should allow for consistency and completeness checks at all levels of the analysis hierarchy and forbid any contradictory operation. The tool should also enable easy movement between levels through design and analysis.

Code Generation

The CASE tool should support code generation by providing a traceable path from the source file to the design data. Concrete support requirements include:

- Creation of module skeletons or templates in well-known languages.
- Automatic generation of records, structures, and class definitions from the data dictionary's contents.
- Creation of database tables for relational database management systems.
- Provision of user interface code for various application types.

TestCase Generator

The CASE tool used to create test cases should support both design and requirement testing and produce ASCII-formatted test set reports that can be inserted directly into the test plan document.

12.4 Second-generation CASE tool

One of the key characteristics of second-generation CASE tools is their direct support for modified methodologies. To adapt a CASE tool to a specific methodology, the task would typically be performed by a CASE administrator or organization. Second-generation CASE tools also provide the following features:

- **Intelligent Diagramming Support:** Diagramming techniques are widely recognized for their advantages in system analysis and design. Second-generation CASE tools automatically and artistically lay out diagrams with the assistance of advanced CASE technologies.
- **Integration with the Implementation Environment:** Integration between design and implementation is facilitated using CASE tools.
- **Data Dictionary Standards:** Users should be able to combine multiple development tools in a single environment. It is improbable that any single provider will be able to offer a comprehensive solution. Additionally, desired tools would require system tuning, making the user act as a system integrator. This is feasible only if a data dictionary standard emerges.
- **Customization Support:** Users should have the capability to create new types of objects and connections. This feature could be utilized to develop unique techniques. Ideally, the rules of a methodology should be specified to a rule engine for carrying out essential consistency tests.

12.5 CASE Environment Architecture

The figure depicts the architectural layout of a typical modern CASE environment (Figure 12. 2).

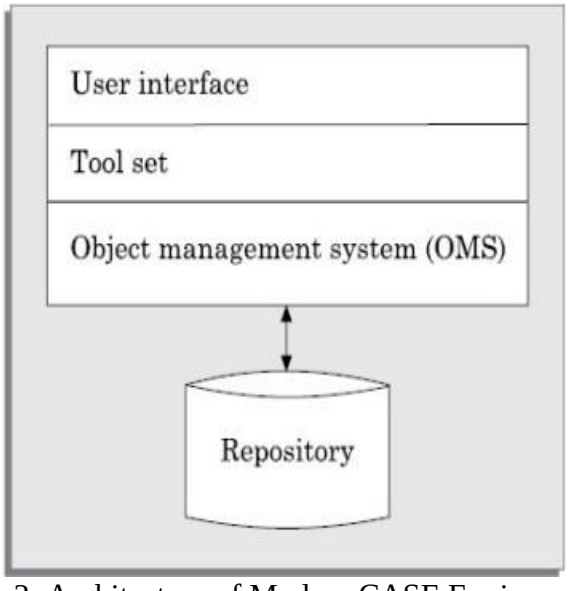

Figure 12. 2: Architecture of Modern CASE Environment

A modern CASE environment must include a user interface, tool set, object management system(OMS), and repository.

The User Interface

The user interface provides a uniform framework for accessing the many tools, making it easier for users to engage with the various tools and lowering the overhead of learning how to utilize the various tools.

Object Management System (OMS) and Repository

Various CASE tools portray a software application as a collection of components such as specifications, designs, text data, project plans, and so on. These logical entities are mapped by the object management system into the underlying storage management system (repository). Commercial relational database management systems are designed to support vast amounts of data formatted as simple, relatively short records. There are only a few categories of entities, yet there are a great number of examples. CASE tools, on the other hand, generate a vast number of entity and relation types, with just a few examples of each. As a result, the object management system handles the mapping into the underlying storage management system.

12.6 Summary

- We have outlined some of the most critical features of modern CASE tools and discussed emerging developments.
- CASE tools are becoming almost indispensable for large software projects, often involving teams of software developers.
- CASE technologies relying on distributed workstations are gaining popularity.
- We have highlighted some of the attractive aspects of distributed workstation-based CASE tools

12.7 Keywords

CASE Environment

A CASE environment supports the automation of software development procedures step by step. In contrast, a programming environment, unlike a CASE environment, is a comprehensive collection of tools that solely facilitate the coding phase of software development.

Data Dictionary Standards: Users should have the ability to integrate multiple development tools into a unified environment. It's highly unlikely that any single vendor can provide a complete

solution. Moreover, using a chosen tool would require system-specific adjustments. Therefore, the user would need to act as a system integrator, which is only feasible if a data dictionary standard emerges.

Code Generation: Regarding code generation, the typical expectation from a CASE tool is relatively modest. Having traceability from source files to design data is a feasible requirement.

Object Management System (OMS): Various CASE tools represent the software product as a collection of elements, such as specifications, designs, text data, and project plans. These logical entities are mapped by the object management system to the underlying storage management system (repository). While commercial relational database management systems are adept at handling vast amounts of data formatted as simple, relatively short records, CASE tools generate numerous entity and relation types with only a few examples of each. Consequently, the object management system is responsible for appropriately mapping these entities into the underlying storage management system.

12.8 Self-Assessment Questions

1. What do the terms CASE tool and CASE environment mean to you? Why do integration tools boost the tool's power? Use examples to explain.
2. What are some of the key features that a next-generation CASE tool should have?
3. Determine the CASE support available during a big maintenance operation involving legacy software.
4. Draw a schematic of a CASE environment and describe how the various tools are integrated.
5. What CASE tool support can be expected during code generation?

12.9 Case Study

Introduction

Company ABC is a software development company that recently implemented Computer-Aided Software Engineering (CASE) tools into their development process. Let's explore their CASE, address some questions, provide recommendations, and conclude with the key takeaways.

Problem:

- **Inefficient software development process:** Company ABC faced challenges with manual and time-consuming software development processes. There were inconsistencies, delays, and difficulties in managing complex software projects.

1. **Lack of collaboration and documentation:** The company struggled with collaboration among development teams and maintaining comprehensive documentation. This resulted in miscommunication, knowledge gaps, and difficulties in maintaining and updating software systems.
2. **Limited productivity and reusability:** The company's development teams lacked standardized methodologies and tools to improve productivity and reuse software components efficiently. This led to redundant efforts, slower development cycles, and increased costs.

Recommendations:

1. **Select and implement suitable CASE tools:**
    - Conduct a thorough evaluation of available CASE tools and select the ones that align with the company's development needs and goals.
    - Consider tools that offer features such as code generation, version control, documentation management, and collaboration capabilities.
    - Provide adequate training and support to the development team to ensure effective adoption and utilization of the selected CASE tools.
2. **Standardize development methodologies and processes:**
    - Define and implement standardized development methodologies and processes that integrate with the CASE tools.
    - Establish coding guidelines, documentation standards, and quality control measures to ensure consistency and maintainability of software systems.
    - Foster a culture of following established processes and continuously improving them based on feedback and lessons learned.
3. **Encourage collaboration and knowledge sharing:**
    - Utilize collaboration features of the CASE tool to facilitate communication, document sharing, and collaborative development.
    - Implement knowledge management systems to capture and share project-related information, lessons learned, and best practices.

- Foster a collaborative and inclusive environment where team members actively participate in discussions, share insights, and learn from each other.

4. **Promote code reusability and component-based development:**
    - Encourage modular and component-based development practices to facilitate code reusability.
    - Utilize CASE tools to create and maintain a repository of reusable software components, libraries, and templates.
    - Establish guidelines and processes for identifying, documenting, and sharing reusable components among development teams.

Conclusion:

Computer-Aided Software Engineering (CASE) tools can significantly improve software development processes by automating tasks, enhancing collaboration, and promoting standardization. By selecting and implementing suitable CASE tools, standardizing development methodologies, fostering collaboration and knowledge sharing, and promoting code reusability, Company ABC can overcome its challenges and achieve improved productivity, quality, and efficiency in software development. CASE tools enable companies to streamline their development processes, reduce manual effort, enhance collaboration, and ensure the delivery of high-quality software systems. By embracing CASE tools and associated best practices, organizations can stay competitive, meet customer expectations, and drive innovation in the software development industry.

12.10 References

1. Summerville, I: Software Engineering9thEdition, Pearson Education, 2017.
2. Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014.
3. Roger S. Pressman: A Practitioner's Approach, 7th Edition, Mc Graw Hill Education, 2009.