Bachelor of Computer Applications (BCA)

Software Engineering

(OBCACO303T24)

Self-Learning Material (SEM-III)

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
(Computer and Systems Sciences, JNU Jaipur)

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

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

COURSE COORDINATOR

Mr. Pawan Jakhar
(Computer and Systems Sciences, JNU Jaipur)

UNIT PREPARATION

Unit Writer(s)	**Assisting & Proofreading**	**Unit Editor**
Mr. Shish Dubey (Computer and Systems Sciences, JNU Jaipur) (Unit 1-4)	Mr. Kuldeep Sharma (Computer and Systems Sciences, JNU Jaipur)	Dr. Satish Pandey (Computer and Systems Sciences, JNU Jaipur)
Mr. Ram Lal Yadav (Computer and Systems Sciences, JNU Jaipur) (Unit 5-8)		
Ms. Heena Shrimali (School of Computer and Systems Sciences) (Unit 9-12)		

Secretarial Assistance
Mr. Mukesh Sharma

COURSE INTRODUCTION

This course on software engineering is designed to provide students with a comprehensive understanding of the field's importance, the development of its practices, and the impact of its methodologies. As the digital landscape evolves, the role of software engineering has become pivotal in crafting robust, efficient, and scalable software systems. This course aims to explore the historical progression of software engineering practices, highlighting how they have adapted to meet the increasing demands of technology and business.

This course has 3 credits and is divided into 12 Units. Starting with the fundamental importance of software engineering, students will gain insights into why structured approaches to software development are crucial for project success. The course will cover the emergence of software engineering as a discipline, examining key milestones that have shaped its practices and methodologies over the years. This historical context sets the stage for a deeper exploration of various software engineering concepts such as design, testing, maintenance, and project management.

An essential part of the course involves analyzing the impact of different software engineering methodologies. From waterfall to Agile, students will evaluate each method's strengths and weaknesses and their suitability for various types of projects. This analysis will help students understand how the selection of a methodology can significantly influence the success of a software development project.

Furthermore, the course will discuss the ongoing changes and trends within the field of software engineering, emphasizing the need for continuous learning and adaptation. It will also provide a detailed look at system engineering—an integral part of software engineering that focuses on designing and managing complex engineering projects.

Lastly, the course will explore the role of system analysts, who are key players in the software development process. Their responsibilities in translating business requirements into technical specifications make them vital to the successful implementation of software projects.

By the end of this course, students will have a thorough understanding of software engineering's essential aspects, enabling them to apply best practices and methodologies to real-world development scenarios.

Course Outcomes:

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

1. Identify, formulate, and solve complex engineering problems by applying principles of engineering, science, and mathematics
2. Apply engineering design to produce solutions that meet specified needs with consideration of public health, safety, and welfare, as well as global, cultural, social, environmental, and economic factors
3. Communicate effectively with a range of audiences
4. Develop and conduct appropriate experimentation, analyze and interpret data, and use engineering judgment to draw conclusions
5. Acquire and apply new knowledge as needed, using appropriate learning strategies
6. Develop efficient software using the latest tools and techniques. Use of computer aided designing and automated testing tools

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.
- Togothroughthevariouschangesthatoccurredinsoftwareengineeringconceptsand methodology.
- To understand system engineering.
- To understand the role of system analysts in the development process.

Structure:

Introduction: Software Engineering

Today's era is heavily reliant on technology, with computer-based systems integral to business operations. Software can range from simple applications for managing client information to complex systems tackling scientific problems. Due to its intangible nature, software development can become exceedingly complex without adhering to specific rules during the development

process. Software engineering addresses these challenges by providing a clear and concise framework for designing, developing, testing, and maintaining software applications.

1.1 History of Software Engineering

In the early stages of computer programming, applications were developed by individuals or small teams using low-level languages like assembly. As programming languages evolved, such as with the advent of high-level programming languages in the 1960s and 1970s, methodologies for software engineering began to take shape. The term "software engineering" gained prominence around 1968, marking a shift towards structured approaches to enterprise-scale application development. Object-Oriented Programming (OOP) emerged in the 1980s, revolutionizing software design and development. The 1990s saw the rise of Agile methodologies, which prioritize flexibility and adaptability in response to changing requirements, solidifying software engineering as a disciplined and mature field.

1.2 Importance of Software Engineering

Software engineering plays a crucial role in ensuring successful software development projects. Drawing an analogy to building a house without proper planning, software engineering emphasizes:

- **Proper requirement analysis**
- **Effective design principles**
- **Cost and time analysis**
- **Adaptability to changes**
- **Efficient time management**

By adopting software engineering practices, organizations benefit from:

- **Cost-effectiveness** in development, testing, and maintenance.
- **Predictability** in project timelines.
- **Maintaining quality** through systematic design and methodologies.
- **Reduced complexity** by breaking down problems into manageable units.
- **Improved communication** among team members and stakeholders.
- **Increased productivity** with clear design and requirements.
- **Ease of maintenance** for systematically designed projects.

- **Scalability** to accommodate new changes.

1.3 Changes in Software Development Practices

Software development practices have evolved significantly, driven by advancements in technology, changes in programming languages, and shifts in work culture. Modern practices include:

- Transition from **traditional waterfall** to **agile methodologies**, allowing for flexibility and stakeholder involvement.
- Adoption of **DevOps** to integrate development and IT operations, enhancing speed and quality.
- Embracing **cloud computing** for scalability and resource sharing.
- Implementation of **automated testing** tools like Selenium and Cucumber for efficiency.
- Utilization of **microservices** architecture for scalability and modular development.
- Integration of **container orchestration tools** like Docker for efficient deployment.
- Focus on **data security** through advanced practices and encryption.
- Leveraging **Machine Learning** and **Artificial Intelligence** for data-driven applications.
- Development of **Progressive Web Applications (PWA)** for enhanced web experiences.

1.4 System Engineering

System engineering encompasses the entire system development process, including hardware, software, and the overall system configuration. It focuses on system requirement analysis, design, development, testing, deployment, and maintenance. System engineers collaborate with various teams and stakeholders to ensure the system functions effectively in production environments, delivering quality software with high accuracy.

In conclusion, software engineering and its methodologies are crucial in modern software development, ensuring efficiency, quality, and adaptability in a rapidly evolving technological landscape. By embracing these principles and practices, organizations can navigate complexities, deliver superior products, and meet the demands of today's digital environment effectively.

1.5 Difference between Software Engineering and System Engineering

System engineering and software engineering are disciplines that are linked but have

uniqueness and areas of competence. Here are the main distinctions between the two:

Perspective &Scope:

System engineering offer salarger approach, focusing on the design, development, and management of complex systems that may contain hardware, software, firmware, and other components. It takes into account the relationships between these components as well as the system's integration with its surroundings.

It focuses on the software component of a broader system and handles issues such as coding, testing, and software design.

1.6 Role of System Analyst

1 The role of a system analyst is pivotal in ensuring the overall success of a project. System analysts play a crucial role in gathering requirements, analyzing them, documenting solutions, designing systems, and facilitating communication among stakeholders and development teams. Here's a breakdown of their responsibilities:

2 **Requirement Gathering:** System analysts initiate the project by collecting data about the problem through questionnaires, workshops, and stakeholder interviews. They navigate through existing systems if available, which can expedite the process. However, if no such model exists, requirement gathering becomes more challenging due to the need for extensive stakeholder interaction. During this phase, it's essential to reconcile any contradictions in requirements and ensure all conditions are addressed.

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

4 **Documentation:** System analysts meticulously document the proposed solution in the Software Requirement Specification (SRS) document. This document serves as an official guide for the development team, detailing functional and non-functional requirements, and outlining the implementation procedures. It's important to note that the SRS document may evolve as the project progresses and requirements change.

5 **System Design Preparation:** System analysts are responsible for preparing system designs that include data flow diagrams, use cases, integration points, and more. They

ensure that these designs accurately reflect the stakeholders' requirements and that system architecture and interfaces are clearly defined for the development process.

6 **Communication:** Acting as a bridge between stakeholders and development teams, system analysts facilitate effective communication throughout the project lifecycle. They interact with developers, testers, and deployment teams to ensure that all work is aligned with the requirements specified in the SRS document. Clear communication and comprehensive documentation are crucial to avoiding misunderstandings and ensuring project success.

7 **Recommendations:** System analysts not only identify business problems but also propose technology-based solutions to enhance business operations. They collaborate with stakeholders to recommend solutions that align with organizational goals and drive business growth.

1.7 Summary

Establishing the Boundaries of Software Engineering

In defining software engineering, two equivalent definitions were established based on years of programming experience and research discoveries:

1. **Definition 1:** Software engineering involves the methodical application of programming expertise and research findings to economically produce high-quality software. It is the engineering method used for software creation.
2. **Definition 2:** Software engineering techniques are crucial for developing large software products where teams of engineers collaborate. However, these principles are also beneficial for developing smaller programs.

Furthermore, the development of computer systems engineering integrates the creation of both software and hardware components. Software engineering operates within the framework of computer systems engineering.

In the upcoming chapter, various software engineering principles will be explored. It's important to note that a deeper understanding of these fundamentals is typically gained through experience in building large-scale programs. Students without prior programming experience may need to invest additional effort to grasp these concepts effectively.

Keywords

- **Software Engineering:** A disciplined approach encompassing system requirement analysis, model design, development, configuration, testing, deployment, and maintenance of software applications.
- **System Engineering:** A disciplined methodology that addresses the entire system, including hardware, software, and the development process. It ensures the integrated delivery of all developed components into a functional system.
- **System Analyst:** The backbone of the development process who interacts with stakeholders to gather system requirements. Effective communication and thorough requirement analysis by system analysts are critical for meeting product expectations.
- **Stakeholders:** Individuals or groups directly affected by the product or its development process. This includes end-users and team members interested in all aspects of the product.

1.8 Self-Assessment Questions

1. Why is software engineering important?
2. What are the different kinds of software development practices followed in the IT industry?
3. What are the key challenges of software engineering?
4. Why is system engineering different from software engineering?
5. What is the role of a System Analyst in the IT industry?

1.9 Case Study: Online Retail Platform

Introduction: The case study explores challenges faced by "eShopNow," an online retail platform that started small but encountered performance, security, and complexity issues as it grew.

Background: As "eShopNow" expanded rapidly, it struggled with performance bottlenecks, security vulnerabilities, and managing software complexity. A dedicated software engineering approach became necessary to address these challenges effectively.

Task: To resolve these issues and improve user experience, a suitable SDLC process needs to be recommended. This process should be capable of handling scalability, ensuring security, and managing complexity efficiently while enhancing overall user satisfaction.

By leveraging appropriate software engineering methodologies, "eShopNow" can streamline its development processes, fortify its security measures, optimize performance, and manage software complexity effectively, thereby enhancing its overall business operations and user experience.

Questions to be considered:

System Reliability and Performance:

1. Ensuring Reliability and Stability: Software engineering contributes to ensuring the reliability and stability of the e-shop platform through several measures:

- **Fault Tolerance:** Implementing redundancy and failover mechanisms to minimize service disruptions in case of component failures.
- **Monitoring and Alerting:** Setting up monitoring tools to detect performance issues, errors, and downtime proactively.
- **Load Balancing:** Distributing incoming traffic across multiple servers to optimize resource utilization and prevent overload.
- **Automated Recovery:** Implementing automated recovery processes to restore services quickly in case of failures.

2. Performance Optimization: To optimize the performance and responsiveness of the e-shop platform:

- **Code Optimization:** Conducting code reviews and performance profiling to identify and resolve bottlenecks in the codebase.
- **Caching:** Implementing caching strategies (e.g., database query caching, content caching) to reduce response times and improve scalability.
- **Database Optimization:** Optimizing database queries, indexes, and schema design to enhance data retrieval speed and efficiency.
- **Content Delivery Networks (CDNs):** Utilizing CDNs to cache static content closer to users, reducing latency and improving page load times.

3. Handling High Traffic Volumes and Minimizing Downtime: Software engineering helps in handling high traffic volumes and minimizing downtime by:

- **Scalable Architecture:** Designing the e-shop with a scalable architecture (e.g., microservices) that supports horizontal scaling to handle increased user traffic.
- **Performance Testing:** Conducting load testing and stress testing to assess system behavior under peak loads and optimize accordingly.
- **High Availability (HA) Configuration:** Setting up redundant systems and failover mechanisms to ensure continuous availability and minimal downtime during maintenance or unexpected traffic spikes.

User Interface and User Experience (UI/UX):

1. Designing an Intuitive and User-Friendly Interface: Software engineering plays a crucial role in designing an intuitive and user-friendly interface by:

- **User Research:** Conducting user research, personas, and usability testing to understand user preferences and behaviors.
- **Wireframing and Prototyping:** Creating wireframes and prototypes to visualize and iterate on interface designs before implementation.
- **Responsive Design:** Ensuring the interface is responsive and accessible across various devices and screen sizes.
- **Accessibility:** Incorporating accessibility features (e.g., screen reader compatibility, keyboard navigation) to ensure inclusivity for users with disabilities.

2. Considerations for Usability, Accessibility, and Visual Aesthetics:

- **Usability:** Focusing on intuitive navigation, clear information architecture, and minimal cognitive load for users.
- **Accessibility:** Adhering to accessibility standards (e.g., WCAG) to accommodate users with disabilities and ensure compliance.
- **Visual Aesthetics:** Applying principles of visual design, such as typography, color schemes, and visual hierarchy, to enhance user engagement and brand identity.

3. Gathering User Feedback and Continuous Improvement: Software engineering teams gather user feedback and continuously improve UI/UX by:

- **Feedback Loops:** Implementing feedback mechanisms like surveys, user interviews, and analytics tools to gather insights.
- **Iterative Design:** Using Agile methodologies to iteratively prototype, test, and refine interface designs based on user feedback.
- **A/B Testing:** Conducting A/B testing to compare different interface variations and determine optimal design choices.
- **Metrics Analysis:** Analyzing user behavior metrics (e.g., bounce rates, conversion rates) to identify usability issues and optimize the user experience.

Security and Data Protection:

1. Addressing Security Vulnerabilities and Protecting Customer Data: Software engineering addresses security vulnerabilities and protects sensitive customer data by:

- **Secure Coding Practices:** Following secure coding guidelines (e.g., OWASP Top 10) to prevent common security threats like SQL injection, cross-site scripting (XSS), etc.
- **Data Encryption:** Implementing encryption protocols (e.g., HTTPS, TLS) to secure data transmission and storage.
- **Authentication and Authorization:** Implementing strong authentication mechanisms (e.g., multi-factor authentication) and access controls to prevent unauthorized access.
- **Regular Security Audits:** Conducting regular security audits and vulnerability assessments to identify and mitigate potential risks proactively.

2. Compliance with Data Protection Regulations: To ensure compliance with data protection regulations such as GDPR or CCPA:

- **Data Privacy Policies:** Implementing clear data privacy policies and practices to inform users about data collection, processing, and storage.
- **User Consent Management:** Obtaining explicit consent from users for data processing activities and providing mechanisms for users to manage their consent preferences.
- **Data Access Controls:** Restricting access to sensitive data based on user roles and implementing logging and auditing mechanisms to track access.

Scalability and Flexibility:

1. Enabling Scalability for Growth: Software engineering enables the e-shop platform to scale and handle growth by:

- **Elastic Infrastructure:** Using cloud-based infrastructure (e.g., AWS, Azure) that allows for dynamic scaling based on demand.
- **Microservices Architecture:** Decomposing monolithic applications into microservices to facilitate independent scaling of services.
- **Horizontal and Vertical Scaling:** Implementing strategies for horizontal scaling (adding more instances) and vertical scaling (increasing resource capacity) based on workload requirements.

2. Architectural Considerations for Future Expansions and Integrations:

- **API Integrations:** Designing robust APIs to facilitate integrations with third-party systems (e.g., payment gateways, ERP systems) and enable new features.
- **Modular Design:** Adopting a modular architecture that allows for easy addition and removal of components to support future expansions.
- **Scalable Database Solutions:** Using scalable database solutions (e.g., NoSQL databases, sharding) to handle increasing data volumes and transaction throughput.

3. Adaptation to Changing Market Trends and Customer Demands:

- **Agile Development:** Embracing Agile methodologies to quickly adapt to changing market trends and customer feedback through iterative development cycles.
- **Continuous Deployment:** Implementing CI/CD pipelines to deploy updates and new features rapidly in response to market demands.
- **Customer Feedback Integration:** Incorporating customer feedback loops into the development process to prioritize feature development and improvements based on user needs.

Backend Operations and Integration:

To effectively integrate various backend systems such as inventory management, order processing, and payment gateways, software engineering employs several strategies. Firstly, it utilizes technologies like APIs (Application Programming Interfaces) to facilitate seamless communication and data exchange between different components of the e-shop

system. APIs standardize the interaction between disparate systems, allowing them to interact efficiently without direct dependencies.

Technologies and frameworks such as RESTful APIs, SOAP (Simple Object Access Protocol), and GraphQL are commonly used to ensure smooth communication and data flow within e-shop systems. These frameworks provide standardized protocols for data exchange, enabling backend systems to interact efficiently and securely.

Backend operations are optimized through software engineering practices to streamline processes, automate tasks, and reduce manual effort. Techniques such as automation scripts, batch processing, and workflow management systems are implemented. These tools automate routine tasks like data processing, order fulfillment, and inventory updates, thereby improving operational efficiency and reducing human error.

1 **Maintenance and Upgrades:**

Software engineering supports ongoing maintenance, bug fixing, and performance monitoring of the e-shop platform through several methodologies. Firstly, it employs robust version control systems such as Git to manage code changes systematically. Regular code reviews are conducted to ensure code quality, identify bugs, and maintain coding standards. Additionally, performance monitoring tools are used to track system performance metrics, identify bottlenecks, and optimize resource utilization.

Strategies for handling software upgrades and new feature releases involve agile development practices. Continuous Integration/Continuous Deployment (CI/CD) pipelines automate the build, test, and deployment processes, ensuring rapid and reliable updates to the platform. Compatibility with evolving technologies is ensured through regular updates of dependencies, frameworks, and libraries used in the e-shop system.

2 **Analytics and Insights:**

Software engineering contributes to capturing and analyzing relevant data about customer behavior, purchasing patterns, and market trends through various analytics tools and frameworks. Technologies like Google Analytics, Mixpanel, and custom-built analytics platforms are utilized to gather actionable insights from user interactions and transactions on the e-shop platform. These insights drive informed decision-making and enable data-driven optimizations to enhance user experience and operational efficiency.

References:

1.10 References:

- I. Sommerville: Software Engineering, 10th 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 -2
Software Development Life Cycle

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 encompass structured and sequential processes essential for effective software development. Among these methodologies, the Software Development Life Cycle (SDLC) models play a pivotal role, providing frameworks that guide the entire software development process. These models address several critical needs in software development:

2.1 Need for Software Development Life Cycle Models

a. **Approach:** SDLC models offer an organized approach to software development, specifying tasks, activities, and deliverables in a structured manner. This ensures that development teams follow a standardized procedure, promoting efficiency and consistency.

b. Analysis of User Needs: Emphasizing user needs analysis, SDLC models ensure that software products meet requirements and expectations. This thorough examination helps in delivering solutions that align closely with customer needs.

c. Project Planning and Management: By providing a roadmap, SDLC models facilitate effective project planning and management. They enable project managers to allocate resources, predict costs, and manage timelines and dependencies effectively.

d. Risk Management: SDLC models enhance risk management by identifying potential risks early in the development cycle. This allows teams to apply mitigation strategies, resolve issues proactively, and minimize the impact of risks on the final product.

e. **Quality Control:** Throughout development, SDLC models emphasize quality control through testing, code reviews, and quality checkpoints. These practices ensure that software meets required quality standards before deployment, enhancing reliability and performance.

f. Communication and Teamwork: SDLC models promote collaboration among various stakeholders including business analysts, testers, designers, and developers. By providing a common framework and vocabulary, they facilitate effective communication and teamwork.

g. Documentation: Documentation is a key focus of SDLC models, ensuring that all stages of development are well-documented. This documentation includes requirements, design choices, implementation details, and testing processes, which are crucial for future maintenance and updates.

h. **Scalability and Maintainability:** SDLC models consider scalability and maintainability during the design phase. By addressing these factors early, they ensure that software can handle future growth, adapt to changing requirements, and remain easily maintainable.

Overall, SDLC models contribute significantly to enhancing effectiveness, reliability, and success rates of software development projects by providing structure, reducing risks, fostering quality assurance, promoting collaboration, and ensuring accurate documentation.

2.2 Entry and Exit Criteria of SDLC Methods

Entry and exit criteria serve as checkpoints in SDLC models, ensuring that specific requirements are met before progressing to the next phase. Here's an overview of the entry and exit criteria for each stage, focusing on the Waterfall model:

a. **Requirement Gathering and Analysis:**

- **Entry Criteria:** Project initiation, understanding of the problem statement, and identification of key stakeholders.
- **Exit Criteria:** Approved and documented customer demands, functional and non-functional requirements, and a finalized requirements document.

b. **Designing:**

- **Entry Criteria:** Completed requirements documents, functional specifications, and approved architecture design methodology.
- **Exit Criteria:** System design documents including architecture diagrams, database schemas, interface requirements, and design review approvals.

c. **Coding or Development:**

- **Entry Criteria:** Approved system design documentation, adherence to coding standards and guidelines.
- **Exit Criteria:** Code reviews, unit test reports, and validated code modules ready for integration.

d. **Testing:**

- **Entry Criteria:** Test plans, test cases, and completion of coding phase.
- **Exit Criteria:** Executed test cases, test results, defect reports, and an approved test summary report.

e. **Configuration and Deployment:**

- **Entry Criteria:** Completion of testing phase, approval of test summary report, and a stable software build.
- **Exit Criteria:** Software deployed in intended environment, installation and user manuals prepared, and an approved deployment checklist.

f. **Maintenance and Support:**

- **Entry Criteria:** Successfully deployed software, user acceptance, and sign-off.
- **Exit Criteria:** Ongoing support plan in place, critical bugs resolved, and issues addressed.

It's important to note that these criteria may vary based on specific SDLC models and project requirements. Agile methodologies, for instance, might have more flexible entry and exit criteria due to their iterative nature.

In conclusion, entry and exit criteria in SDLC models act as critical milestones, ensuring each phase of development is completed satisfactorily and validated before proceeding. They play a vital role in managing project progress, setting clear objectives, and maintaining quality throughout the entire development lifecycle.

2.3 Different SDLC Models

a. There are various software development lifecycle models designed to cater to different user requirements and system design approaches. One of the commonly discussed models is the Classical Waterfall Model. This model divides the core software development activities—specification, development, validation, and evolution—into distinct phases like requirements definition, software design, implementation, and testing. Originating from more generalized systems engineering approaches, the waterfall model was initially proposed by Royce in 1970.**C**.

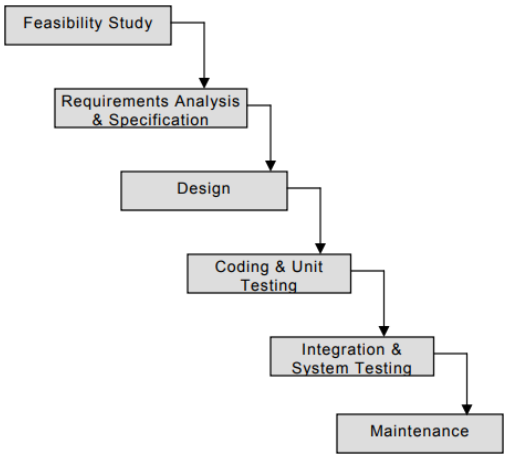

Figure1: The Classical Waterfall Model

In the software development process, each phase ideally produces one or more approved documents before moving on to the next phase. While the waterfall model suggests a linear progression, in reality, these phases often overlap and exchange data. Issues with requirements can surface during design, design flaws may be uncovered during coding, and so forth. This iterative feedback between phases highlights that software development is not strictly linear.

Documents created in each phase may require revisions to accommodate changes discovered throughout the process. Iterations can be costly and time-consuming due to the effort involved in

creating and approving documents. Therefore, it's common to freeze early development phases like specification after a limited number of iterations, deferring unresolved issues to later phases or working around them in the programming stage. However, prematurely freezing requirements can lead to systems that do not fully meet user needs or are poorly structured due to workarounds in implementation.

Throughout the software lifecycle, which includes operation and maintenance phases, the software evolves as new requirements are identified and errors in design or programming are corrected. This ongoing evolution may necessitate revisiting earlier phases of the development process for modifications and software maintenance. Thus, flexibility and adaptation are crucial in responding to changes and improving software over its lifecycle

b. Iterative Waterfall Model:

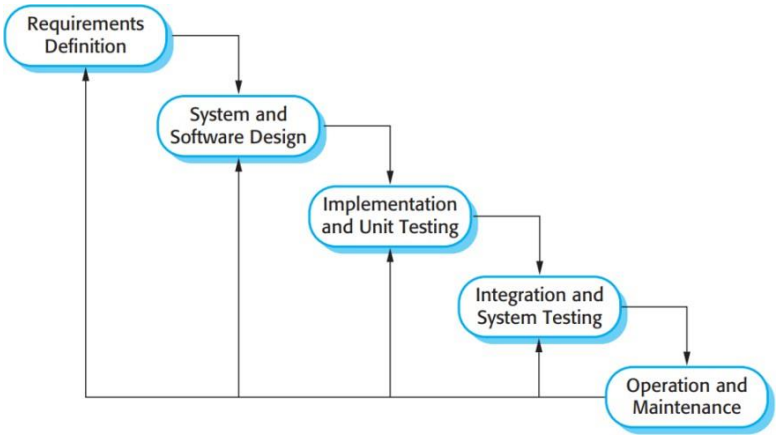

Figure2 Iterative Waterfall Model

As mentioned previously, applying the traditional waterfall approach to real-world software development projects has proven challenging and unrealistic. The conventional waterfall model has been criticized for its inflexibility and linear progression, which often does not accommodate changes and feedback effectively.

In response to these limitations, the iterative waterfall model represents an adaptation of the traditional waterfall model to better suit practical software development scenarios. The primary

enhancement in the iterative waterfall model is the introduction of feedback loops from each stage back to its preceding phase.

Figure 2 illustrates how these feedback pathways operate in the iterative waterfall model. These feedback loops allow for the identification and correction of issues discovered in later phases back to earlier phases. For example, if a design flaw is detected during the testing phase, the feedback pathway enables revisions to the design and updates across allsubsequent documents.

This iterative approach addresses the shortcomings of the traditional waterfall model by incorporating flexibility and responsiveness to changes throughout the software development lifecycle. It acknowledges that software development is iterative and requires continuous refinement and adjustment based on ongoing feedback and evolving requirements.

c. Prototype Model

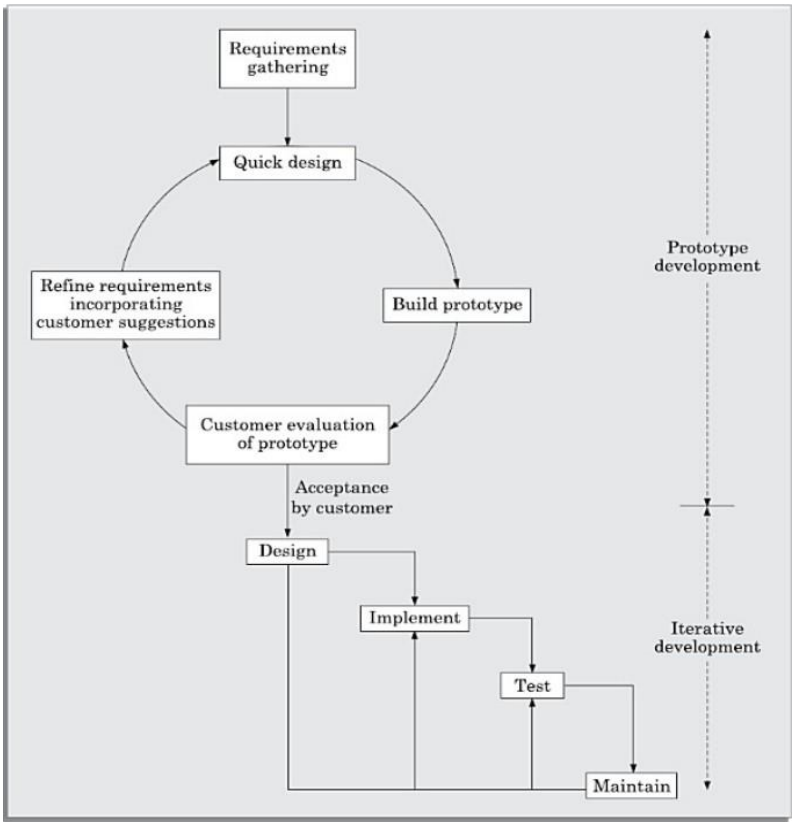

Figure3.Prototype model of the software development process

Another popular lifecycle model is the prototype model, which can be seen as an extension of the waterfall model. This model proposes creating a functional system prototype before developing the actual software. A prototype is a rudimentary implementation of the system, often with inefficiencies, low reliability, or functional limitations compared to the final product. Rapid prototyping techniques involve shortcuts to quickly build prototypes, such as using table look-ups instead of actual computations to achieve desired outputs.

The term "rapid prototyping" typically refers to the use of software tools to expedite prototype development. By constructing a prototype and presenting it for user feedback, many customer requirements can be accurately specified, while technical issues can be identified and addressed through experimentation. This approach helps minimize costs associated with additional client requests for modifications and redesigns.

A key rationale for adopting the prototype model is the acknowledgment that it's challenging to achieve perfection in the first attempt. As articulated by Brooks [1975], software development often requires iterative refinement and the willingness to discard early versions to achieve high-quality outcomes. Therefore, the prototype paradigm is particularly useful when rapid development and effective software solutions are paramount.

Advantages of the Prototyping Approach

The prototype model is most suitable for projects facing technical and requirements risks, as it helps mitigate these risks by providing a built prototype early in the development process.

However, there are some disadvantages associated with the prototype model:

1. **Increased Development Costs**: For projects that do not face significant risks, the prototype model can increase the overall cost of development unnecessarily.
2. **Risk Identification**: The prototype model works best when risks can be identified upfront, before actual development begins. It may not be effective for projects where risks only become apparent later in the development cycle.
3. **Timing of Risk Discovery**: Since prototypes are typically built at the beginning of the project, they may not address risks that emerge later in the development process. This makes the prototyping approach less suitable for projects where risks are discovered after development has started.

In summary, while the prototype model is effective for managing upfront technical and requirements risks, its suitability diminishes for projects where risks are discovered later in the development lifecycle or where risks are not substantial enough to justify the added cost of prototyping.

d. Evolutionary Model:

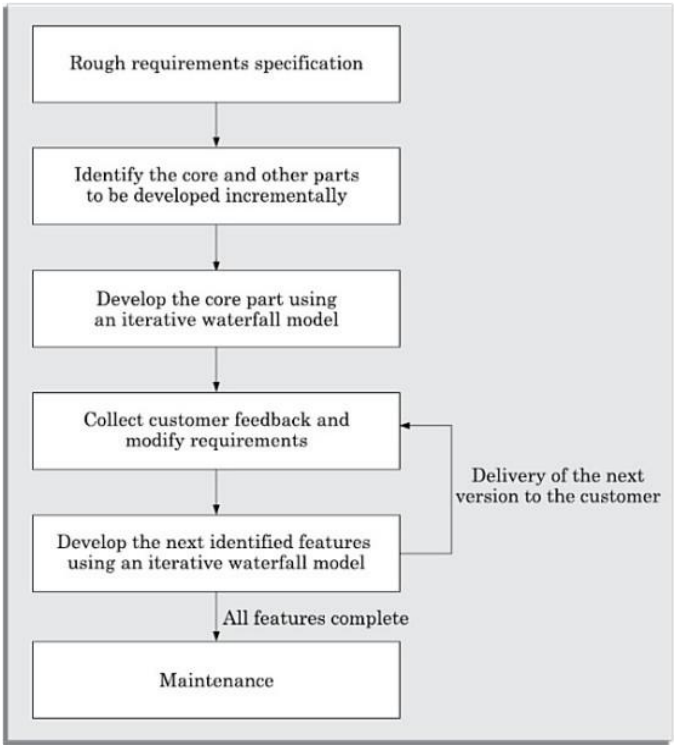

Figure: 4 Evolutionary Model

The Evolutionary model shares many similarities with the Incremental model. In this approach, software development progresses through multiple increments, where each increment implements and deploys a specific feature or idea to the client site. Over time, the software is enhanced and enriched with additional features until it reaches its final state. The name "evolutionary lifecycle model" aptly describes its core principle.

Initially, comprehensive requirements are generated, and an SRS (Software Requirements Specification) document is created as part of the incremental development paradigm. However, unlike the incremental model where requirements, plans, estimates, and solutions are mostly

fixed before development starts, the evolutionary model allows these elements to evolve iteratively throughout the cycles. This flexibility accommodates the discovery of unforeseen features and adjustments that commonly arise during the development of new products.

The evolutionary software development model is particularly suited for developing object-oriented applications, where software can be divided into small, manageable units.

Advantages

Evolutionary Model:

a. Effective Elicitation of Real Customer Requirements: In the evolutionary model, users can test software that is not fully constructed yet. This approach facilitates precise elicitation of user requirements through feedback gathered from distributing several software versions. Consequently, there are significantly fewer change requests once the software is delivered in its entirety.

b. Simplified Change Management: Due to the absence of rigid long-term plans, managing change requests becomes simpler under this paradigm. As a result, rework necessitated by change requests is typically less frequent compared to sequential models.

Disadvantages:

a. Challenge in Feature Division: It can be difficult to divide a feature into incremental pieces suitable for execution and delivery across multiple stages, especially in smaller development projects. Even for larger projects, specialists often struggle with planning incremental delivery due to the interdependencies among features.

b. Ad Hoc Design: Since only the current increment is designed at any given time, there's a risk of hasty design without sufficient attention to maintainability and efficiency. In contrast, the iterative waterfall methodology may produce superior solutions for moderately sized problems where customer needs are explicit.

V-Model:

The V-model is a software development model distinct from the waterfall approach. Like the waterfall model, its name derives from its visual appearance. However, in the V-model,

verification and validation steps are integrated throughout the development lifecycle, significantly reducing the likelihood of errors in work outputs.

Thus, it is widely acknowledged that this model is suitable for projects involving the development of highly reliable, safety-critical software. Represents the validation phases. The test case design testing the work product is finished in each development phase concurrently with the development of the work product itself; however, the actual testing is completed in thevalidationphase.

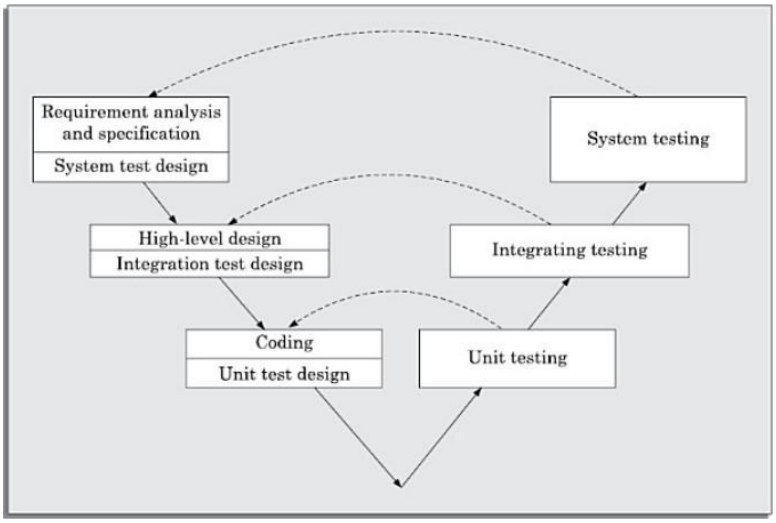

Figure5:V software life cycle Model

Advantages:

V-Model:

a. Simultaneous Testing and Development: In the V-model, many testing tasks such as test case design and test planning are carried out concurrently with development tasks. This approach ensures that a significant portion of testing operations, including test case creation and planning, is completed before the actual testing phase commences.

b. Efficiency Compared to Iterative Methods: Compared to iterative methodologies, the V-model typically results in a shorter testing period and faster overall product development.

c. Higher Test Case Quality: Test cases in the V-model are generally of higher quality because they are developed before schedule pressures escalate. Unlike the waterfall model, where testers are engaged only during the testing phase, the test team remains actively involved throughout the development cycle. This leads to more efficient utilization of human resources.

d. Early Involvement of Test Team: The V-model incorporates the test team from the project's inception. This early involvement allows them to gain a thorough understanding of the development artifacts, facilitating efficient testing. In contrast, in the waterfall model, testing activities start late in the development cycle since no testing occurs before the implementation and testing phase begins.

Disadvantages:

Since the V-model is essentially a successor to the waterfall model, many of the weaknesses inherent in the waterfall model are also present in the V-model.

e. Boehm'sSpiral Model:

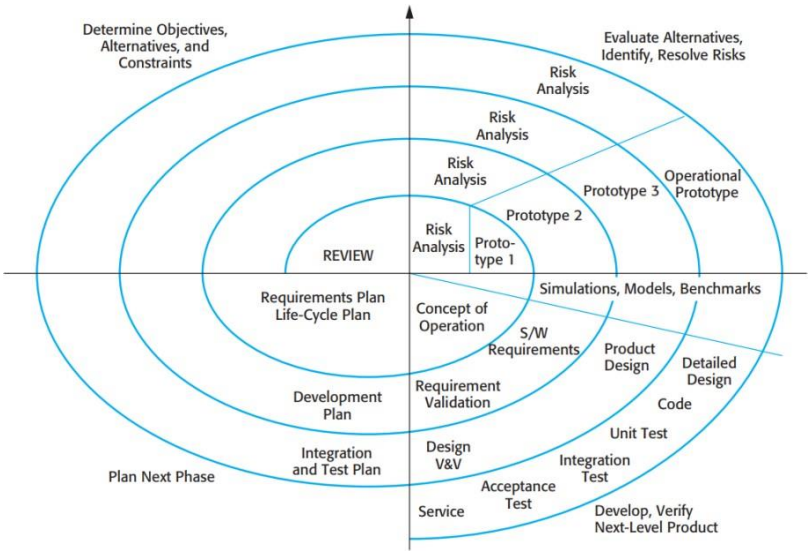

Figure 6: Spiral lifecycle Model

Boehm (1988) presented the spiral model as a framework for risk-driven software **Spiral Model:** The spiral model of software development is depicted as a spiral rather than a linear sequence of steps, emphasizing iterative progress and risk management. Each loop in the spiral represents a

stage in software development, starting from economic feasibility assessment to requirements identification, system design, and subsequent phases.

Advantages:

a. **Effective Elicitation of Customer Requirements:** The spiral model allows users to test software that is not fully constructed, enabling precise elicitation of requirements through feedback gathered from multiple software versions. This reduces the need for extensive changes after final delivery.

b. Simplified Change Management: Unlike sequential models, managing change requests is simpler in the spiral model. This results in fewer reworks needed due to change requests compared to traditional methodologies.

c. Disadvantages:

a. Challenge in Incremental Development: Dividing features into incremental pieces can be challenging, especially for smaller projects or those with tightly integrated features.

b. Ad Hoc Design: The focus on current increments may lead to rushed design decisions, potentially compromising maintainability and efficiency, unlike more structured approaches.

V-Model:

The V-model is a structured approach where verification and validation steps are mirrored throughout the development lifecycle, aiming to minimize errors and ensure reliability in software outputs.

Advantages:

a. Concurrent Testing and Development: Many testing tasks, such as test case design and planning, are completed concurrently with development tasks. This reduces overall testing time and speeds up product delivery.

b. High-Quality Test Cases: Test cases are developed early in the process, resulting in higher quality due to less schedule pressure. Unlike the waterfall model, testers are actively engaged throughout the development cycle, optimizing human resource utilization.

c. Early Involvement of Test Team: The test team is involved from the project's outset, gaining a solid understanding of development artifacts and ensuring efficient testing.

Disadvantages:

The V-model inherits many weaknesses of the waterfall model, such as its inflexibility in accommodating changes after the initial planning stages.

Agile Methodology:

Agile methodologies emphasize iterative development and flexibility in responding to changing requirements throughout the software development process. They prioritize delivering working software quickly and incorporating user feedback for continual improvement.

Conclusion:

- This chapter explores various software development life cycle (SDLC) methodologies used in the industry, emphasizing the importance of selecting an appropriate model based on project specifics.
- The adoption of a suitable SDLC model is crucial for project success, considering factors like project scope, team expertise, adaptability to change, risk management, stakeholder involvement, regulatory compliance, cost, timing, and organizational culture.
- Agile methodologies, including Extreme Programming, Adaptive Software Development, DSDM, Scrum, Crystal, and Feature-Driven Development, offer flexibility and iterative cycles, making them suitable for projects with evolving requirements.

Keywords:

- **SDLC (Software Development Life Cycle)**: Systematic approach to developing software from inception to deployment.
- **Agile Methodology**: Iterative approach to software development that adapts to changing requirements.
- **Waterfall Model**: Traditional sequential model of software development with distinct phases.
- **Stakeholders**: Individuals or groups directly affected by the product or development process.
- **Prototype**: Dummy implementation illustrating user requirements in GUI, forms, and reports, aiding in cost reduction despite additional development expenses.

2.4 Summary

The Software Development Life Cycle (SDLC) encompasses a systematic approach to software development, guiding the progression of a project from conception to delivery and maintenance. Its structured framework ensures the efficient and methodical creation of software systems while minimizing risks and errors. The SDLC typically consists of several phases, including planning, requirements analysis, design, implementation, testing, deployment, and maintenance. During the planning phase, project objectives, scope, and timelines are established, laying the groundwork for subsequent activities. Requirements analysis involves gathering and documenting user needs and system specifications. Design involves creating detailed plans for software architecture and functionality. Implementation translates these designs into actual code, followed by rigorous testing to ensure functionality and reliability. Deployment involves deploying the software to end-users, while maintenance involves ongoing updates and support. The SDLC promotes collaboration among stakeholders, facilitates clear communication, and ensures alignment with project goals and user needs. By adhering to the SDLC model, organizations can streamline development processes, improve quality assurance, and deliver successful software solutions.

2.5 Self-Assessment Questions

1. What is waterfall model?
2. Discuss various selection criteria for SDLC model.
3. Discuss Agile Software methodology.
4. Explain Spiral Model.
5. Discuss various entry and exit check point for SDLC model.

2.6 References:

- I. Sommerville: Software Engineering, 10th 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

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:

3.1 Functional Requirement

It is widely acknowledged that having a high-quality requirements document is crucial for the successful execution of any software development project. A comprehensive requirements specification not only facilitates a clear understanding of all necessary software features but also provides a framework for tasks throughout subsequent lifecycle stages.

The Significance of Documentation in Outsourced Projects

Documentation plays a vital role in software development projects undertaken for other organizations, especially in outsourced projects. It serves several key purposes:

1. **Clarity and Agreement:** A well-documented requirements document ensures that both parties (the client and the development team) have a clear understanding and agreement on the scope, features, and functionalities of the software to be developed.
2. **Alignment of Expectations:** It helps align expectations between the client and the development team, reducing the chances of misunderstandings or discrepancies during the development process.
3. **Legal and Contractual Purposes:** Documentation serves as a legal and contractual reference, outlining the responsibilities, deliverables, timelines, and other terms agreed upon by both parties.

3.1 Requirement Gathering and Analysis

3.1.1 Requirement Gathering

Requirement gathering, also known as requirements elicitation, is the process of collecting and documenting requirements from stakeholders. While it may seem straightforward, gathering requirements is often challenging due to the need to compile essential information from diverse stakeholders and dispersed sources.

In outsourced projects, requirements gathering often begins before on-site visits, utilizing existing documentation and preliminary discussions to gather as much information as possible. Analysts conduct interviews with end-users and customer representatives, along with tasks such as task analysis, scenario analysis, and form analysis.

3.1.2 Requirement Analysis

After gathering requirements, the analyst proceeds with requirement analysis to ensure a clear understanding of customer needs and to address any discrepancies or ambiguities in the gathered requirements. Given that stakeholders may have different perspectives and incomplete views of the system, the analyst identifies and resolves contradictions and gaps through further discussions and clarification with the client.

The primary objectives of requirement analysis include:

- Clarifying the application to be developed based on gathered requirements.
- Resolving ambiguities, inconsistencies, and conflicts in the requirements.

Before conducting analysis, analysts must address fundamental questions about the project:

1. **Problem Identification:** What is the core problem that needs to be addressed?
2. **Significance:** Why is solving this problem important?
3. **Inputs and Outputs:** What are the precise inputs and outputs of the system?
4. **Solution Steps:** What steps can be taken to address the problem?
5. **Complexities:** What complexities might arise in addressing this problem?
6. **Integration Requirements:** What are the data exchange formats if the system needs to communicate with external systems?

During requirement analysis, analysts typically encounter and resolve three categories of challenges:

- **Anomalies:** Various interpretations of a single requirement.
- **Consistency Issues:** Conflicts between different requirements.
- **Incompleteness:** Requirements that lack sufficient detail due to clients' inability to anticipate all necessary features.

3.2 Structure of SRS Document

Functional requirements refer to the services that a system must provide. These requirements specify how the system will behave under certain conditions and how it will respond to specific inputs.

The functional requirements of a system outline what the system is expected to accomplish. These specifications are determined by the type of software being developed, the target users of the software, and the organization's overall operational style. Typically, functional requirements are articulated in a way that system users can understand, often presented as user requirements. However, detailed functional system requirements provide a more precise description of the system's operations, inputs, outputs, exceptions, and so on.

Functional system requirements can vary from high-level needs that outline the system's capabilities to more detailed requirements that consider local work practices or integrate with existing systems within an organization. These requirements are crucial as they define the

specific behaviors and functionalities that the system must exhibit to meet the needs of its users and stakeholders.

3.2.1 Importance of SRS document

The Software Requirements Specification (SRS) document is widely regarded as one of the most critical and challenging documents in the software development lifecycle. This complexity arises primarily because the SRS document is expected to cater to a wide range of users with varying needs and expectations. It serves as a comprehensive repository of all user requirements, encompassing both structured and unstructured information.

Once all necessary data about the software project has been gathered, the analyst undertakes the daunting task of systematically organizing these requirements into the SRS document. This process involves meticulous arrangement to ensure clarity, coherence, and completeness while eliminating inconsistencies and anomalies from the specification.

The analyst's role in structuring the SRS document is crucial as it lays the foundation for all subsequent stages of software development. By methodically organizing and refining the requirements, the SRS document serves as a blueprint that guides the development team in building a software solution that aligns with the client's expectations and operational needs.

In essence, the SRS document is pivotal not only for documenting user requirements but also for establishing a common understanding among stakeholders, thereby facilitating effective communication and collaboration throughout the software development lifecycle. Its comprehensive nature ensures that all aspects of the software's functionality, performance, design constraints, and interfaces are clearly articulated, providing a roadmap for successful project execution.

The main Users of the SRS documents are as follows:

a. Users, clients, and marketing staff:
b. Software engineers:
c. Testing team
d. Document writers
e. Project manager
f. Maintenance specialists:

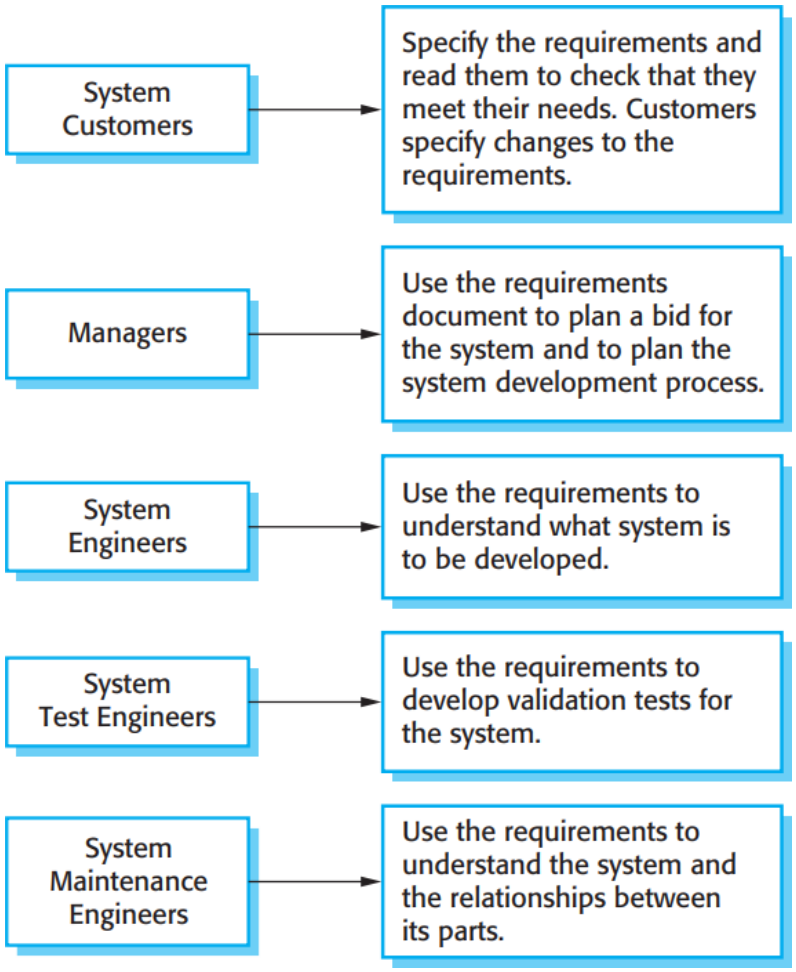

Figure1:User of SRS document

The Software Requirements Specification (SRS) document holds significant importance not only within the software development lifecycle but also in legal contexts such as resolving disputes between clients and developers. Once accepted by the customer, the SRS document serves as a contractual agreement, ensuring that the software developed aligns precisely with the specifications outlined therein.

3.3 Usage of SRS Document

a. **Estimating Software Size**: Project managers rely on the SRS document to assess the scope and size of the software project.

b. **Ensuring Comprehensive Understanding**: The process of creating the SRS document compels stakeholders to carefully consider each requirement upfront, reducing the need for later modifications, rework, and retesting.

c. **Early Detection of Issues**: By meticulously reviewing the SRS document early in the development cycle, errors, misconceptions, and inconsistencies can be identified and addressed promptly.

d. **Establishing a Contract**: The SRS document effectively creates a formal contract between the clients and the development team, specifying the deliverables and expectations.

e. **Benchmark for Compliance**: It sets a standard against which the software's compliance and functionality can be evaluated throughout its development and testing phases.

f. **Guiding Test Strategy**: Test engineers utilize the SRS document to develop the test strategy and test cases that ensure the software meets all specified requirements.

g. **Future Enhancements**: Future enhancements and modifications to the software can be planned and implemented based on the details outlined in the SRS document.

3.4 Characteristics of a Well-Written SRS Document

a. **Verifiability**: Every system requirement listed in the SRS document should be verifiable, meaning it should be possible to determine whether the software satisfies each requirement.

b. **Handling Undesirable Occurrences**: The SRS document should address how the system responds to unexpected events and exceptional circumstances that may arise.

c. **Ease of Modification**: An effectively organized SRS document is easy to modify and update as project requirements evolve over time.

d. **Traceability**: Each specified requirement should be traceable to the design components that implement it, ensuring clarity and accountability throughout the development process.

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

f. **Clarity, Consistency, and Completeness**: It should be concise yet clear, consistent across all sections, and comprehensive in covering all aspects of the software's functionality and performance requirements.

In summary, the SRS document not only serves as a blueprint for software development but also plays a crucial role in contractual agreements, compliance evaluation, and guiding the entire development lifecycle from initial planning to final implementation and beyond. Its careful preparation and adherence to best practices ensure that software projects are executed smoothly and successfully.

3.5 Design of the SRS document

The requirements in an SRS (Software Requirements Specification) document should be appropriately categorized and organized according to IEEE 830 guidelines. These guidelines outline several important customer requirement categories that should be explicitly documented:

1. **Functional Requirements**: These specify what the system should do, including its functionalities and behaviors in response to inputs under specific conditions.
2. **Non-functional Requirements**: This category encompasses various aspects that are not directly related to the system's specific behaviors but are critical for its overall performance and usability. Non-functional requirements can include:
    - **Implementation and Design Limitations**: Constraints or limitations imposed by the implementation approach or design decisions.
    - **Other Non-functional Criteria**: Additional requirements related to performance, reliability, scalability, maintainability, etc.
    - **Need for External Interfaces**: Requirements specifying how the system interacts with external systems or services.
3. **Implementation Objectives**: These describe the goals and objectives related to how the system will be developed, implemented, and deployed.

Organizing requirements into these categories helps ensure clarity and completeness in the SRS document, facilitating effective communication between stakeholders and guiding the development team in meeting the system's specified needs and objectives.

3.6 Decision Trees and Decision Tables

Decision trees and decision tables are fundamental tools used for analyzing and representing complex processing logic. Once decision-making logic is captured in the form of trees or tables, test cases can be automatically derived to validate this logic. However, it should be noted that decision trees and decision tables have applications beyond defining complex processing logic solely within an SRS (Software Requirements Specification) document. They also find applications in information theory and switching theory.

Decision Trees

A decision tree visually represents the processing logic involved in decision-making, including the actions taken based on specific conditions. The edges of a decision tree represent conditions or decisions, and the leaf nodes depict the actions or outcomes resulting from evaluating these conditions.

Decision trees are particularly useful for:

- **Visualizing Decision Logic**: They provide a clear, graphical representation of how decisions are made based on different conditions or criteria.
- **Facilitating Decision-Making**: Decision trees help in understanding the sequence of decisions and actions that need to be taken based on various scenarios or inputs.
- **Automatic Test Case Generation**: Once a decision tree is defined, test cases can be automatically generated to ensure that all possible decision paths and outcomes are tested thoroughly.

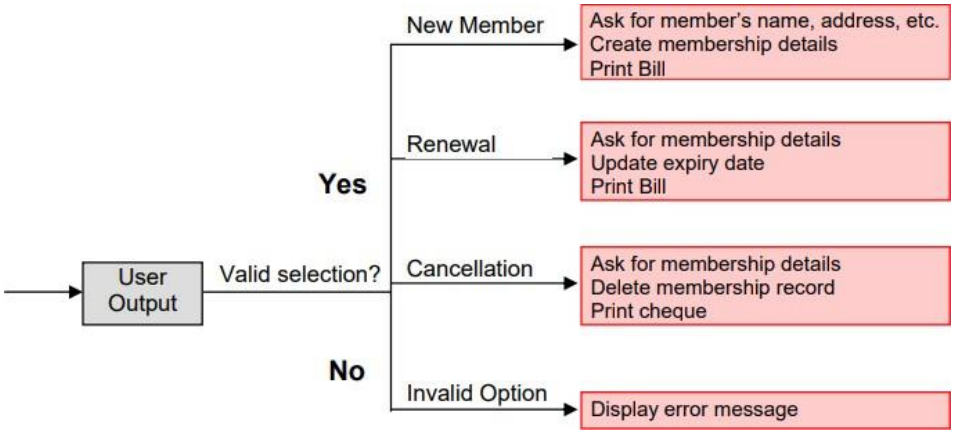

Figure2: Decision Tree for the Library management system

3.6.1 Decision Table

A decision table is structured in a tabular or matrix format that outlines the logic governing decisions and the corresponding actions to be taken based on different conditions. The factors or conditions that influence the decision are typically listed in the upper rows of the table, while the actions or steps to be executed based on those conditions are listed in the lower rows.

In essence, each column in a decision table represents a rule. A rule specifies that a particular action should be taken if a specific combination of conditions is met or true. This format allows decision-makers to visualize and analyze different scenarios comprehensively, ensuring that all possible combinations of conditions and corresponding actions are accounted for.

Decision tables are particularly useful in software development, business analysis, and other fields where complex decision-making processes need to be structured and evaluated systematically. They help in clarifying decision criteria, reducing ambiguity, and ensuring consistent decision-making across different scenarios.

A decision-making table for the Library Management system:

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

Figure3:Decision Table for the Library Management System

3.6.2 Differences between the Decision tree and Decision table

Decision Trees vs Decision Tables for Small Number of Conditions

Decision trees are generally easier to read and understand compared to decision tables when dealing with a small number of conditions. This is because decision trees provide a visual representation of decision-making logic, where conditions and actions are depicted hierarchically, making it intuitive to follow the flow of decisions.

Benefit of Decision Trees in Multilevel Decision-Making

In scenarios requiring multilevel decision-making, decision trees are more advantageous. Decision trees naturally lend themselves to representing hierarchical decision structures, where each node in the tree represents a decision based on specific conditions. This hierarchical representation is straightforward and aligns well with complex decision-making processes.

Challenges of Decision Trees with Increasing Complexity

As the number of conditions and actions increases, decision trees can become extremely complex and difficult to comprehend. Visualizing a large decision tree on a single page can be impractical, leading to challenges in maintaining clarity and understanding. In such cases, the decision table representation may be preferred as it can systematically organize conditions, actions, and their relationships in a tabular format.

Formal Requirement Specifications

Formal methods provide mathematical techniques to specify, analyze, and verify hardware and software systems. These methods enable rigorous specification of system behaviors and properties without executing the system itself. Key aspects of formal methods include:

- **Specification Language**: A formal specification language consists of syntactic and semantic domains and a satisfaction relation between them. This mathematical foundation ensures precise specification of system requirements and behaviors.
- **Axiomatic Specification**: Axiomatic specification uses first-order logic to define pre-conditions (criteria before operation invocation) and post-conditions (requirements after function execution). It provides a rigorous approach to defining and understanding system operations.
- **Algebraic Specification**: Algebraic specification defines object classes or types based on the relationships among operations defined on those types. This approach, popularized in defining abstract data types, uses algebraic structures to specify system components.
- **Executable or 4GL Process**: Executable specifications, often implemented in 4th Generation Languages (4GLs), allow for immediate execution of system specifications without developing full implementation code. While beneficial for rapid prototyping,

executable specifications may exhibit performance inefficiencies compared to optimized code written in lower-level languages (3GLs).

3.7 Summary

- **Requirement Gathering and Analysis**: This chapter focuses on the crucial phase of Software Development Life Cycle (SDLC), emphasizing the importance of producing a high-quality Software Requirements Specification (SRS) document before proceeding to design activities.
- **Formalizing Requirements**: Formal methods in requirement specification offer advantages in ensuring precise and unambiguous system specifications. However, they also pose challenges in practical application, which may be mitigated with improved tool support and usability.
- **Challenges and Approaches**: The chapter discusses challenges in formal specification techniques and explores examples such as axiomatic and algebraic methods to illustrate formal specification methodologies in software engineering.

3.8 Keywords

SRS (Software Requirements Specification)

The SRS document stands out as one of the most critical and challenging aspects of the software development life cycle. Its role in serving a diverse range of users contributes to this complexity.

Formal Requirement Specification

Formal specifications add rigor to the software development process. The act of rigorously specifying system behavior often proves more crucial than the formal definition itself. It clarifies nuances of system behavior that may remain ambiguous in informal specifications.

Functional Requirement

Functional requirements outline high-level demands where each requirement processes input data from users and generates specific outputs. These requirements may encompass multiple functions to fulfill user needs effectively.

Decision Tree

A decision tree visually represents decision-making logic and subsequent actions. Conditions are depicted as branches (edges), leading to actions specified at the leaf nodes based on condition outcomes.

Decision Table

A decision table presents complex processing logic in a tabular or matrix format. Variables or criteria are listed in the table's rows, while corresponding actions to be executed are specified in the columns. Each column (rule) outlines specific actions based on condition satisfaction.

Algebraic Specification

Algebraic specification defines object classes or types based on the relationships among operations defined on those types. Initially popularized by Guttag in the specification of abstract data types, algebraic specifications use various notations such as OBJ and Larch languages.

Axiomatic Specification

Axiomatic specifications define system operations using pre-conditions (requirements that must be met before execution) and post-conditions (requirements that must be fulfilled after execution). This approach ensures clarity in defining system behaviors and outcomes.

These specifications and models play crucial roles in software development, providing structured frameworks to articulate and validate system requirements and behaviors effectively.

3.9 Self-Assessment Questions

1. What are the various types of requirements problems that an analyst normally predicts and corrects in the requirements acquired before beginning to produce the SRS document? Please include at least one example of each.
2. What is the difference between Requirement gathering and Requirement analysis?
3. How many types of system requirements are available? What is the difference between Functional and Non-functional requirements?
4. Describe five desirable features of a good software requirements specification (SRS) document.

5. What are the objectives of the phase of requirements analysis and specification? How and by whom are the requirements analysis and specification processes carried out?

3.10 Case study

Case Study: Requirement Gathering and Requirement Analysis

Company Overview: XYZ Technologies specializes in developing customized business applications for clients across various industries. Recently, they received a project from a healthcare organization to develop a new electronic medical records system.

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

Requirement Gathering:

a) Methods Used for Requirement Gathering:

- XYZ Technologies utilized a combination of interviews, surveys, and workshops to gather requirements. This approach ensured a comprehensive understanding of user needs and system requirements.

b) Involvement of Stakeholders:

- Key stakeholders including doctors, nurses, administrators, and IT staff were actively involved in the requirement gathering process. This ensured diverse perspectives were captured and requirements were aligned with user expectations.

c) Effectiveness of Requirement Documentation:

- Requirements were effectively documented using collaborative tools and techniques such as prototyping and storyboarding. This helped visualize and validate requirements early in the project lifecycle.

Recommendations for Requirement Gathering: a. Use a combination of interviews, surveys, and workshops to ensure comprehensive requirement gathering. b. Involve key stakeholders from different departments to capture diverse perspectives. c. Utilize collaborative tools like prototyping and storyboarding for effective visualization and validation of requirements. d. Maintain clear and concise documentation of all requirements, both functional and non-functional, to serve as a reference throughout the project.

Requirement Analysis:

a) Analysis of Gathered Requirements:

- XYZ Technologies employed techniques such as requirement validation, verification, and traceability to analyze gathered requirements. This ensured clarity, completeness, and consistency in the requirements.

b) Identification of Conflicts or Ambiguities:

- During requirement analysis, conflicts, ambiguities, or gaps in the requirements were identified through reviews and walkthroughs with stakeholders.

c) Prioritization of Requirements:

- Requirements were prioritized based on their criticality, impact on system functionality, and alignment with project objectives.

Recommendations for Requirement Analysis: a. Employ requirement validation, verification, and traceability techniques to ensure clarity and completeness. b. Conduct regular requirement reviews and walkthroughs with stakeholders to identify conflicts, ambiguities, or gaps in requirements. c. Prioritize requirements based on criticality, impact, and alignment with project objectives. d. Establish a requirement traceability matrix to link requirements to design, development, and testing phases.

Questions to be considered:

a) Capturing All Requirements:

- XYZ Technologies ensured all requirements were captured by involving stakeholders from different departments and utilizing comprehensive requirement gathering techniques.

b) Challenges Faced:

- Challenges during requirement gathering and analysis included managing diverse stakeholder expectations, handling evolving requirements, and ensuring clear communication.

c) Management of Requirement Changes:

- Changes in requirements were managed through a robust change management process that evaluated, prioritized, and incorporated new requirements while managing project scope and timelines effectively.

Recommendations: a. Establish robust communication channels with stakeholders to ensure ongoing collaboration and feedback. b. Conduct regular requirement review sessions to accommodate changes and address evolving needs. c. Implement a change management process to evaluate, prioritize, and incorporate new requirements while managing project scope and timelines effectively.

Conclusion: Requirement gathering and analysis are critical stages in any software development project. XYZ Technologies should focus on involving all stakeholders, employing effective techniques, and maintaining clear documentation to ensure accurate and comprehensive requirements. By addressing challenges and implementing recommended strategies, the company can enhance the success of the electronic medical records system project.

3.11 References:

- I. Somerville: Software Engineering 10th 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 determine the software development activities.
- During the initial and detailed design phases, identify the products to be designed.
- Understand the key distinctions between analysis and design tasks.
- To determine the key elements created throughout the software design phase.
- Explain the key desirable aspects of a good software design.
- To determine the features of a design document that are required for understanding.

Structure:

Introduction

During the software design phase, the design document is created based on the customer requirements outlined in the Software Requirements Specification (SRS) document. This process, known as the design process, involves transforming the SRS document into the design document through various design activities. Creating an effective software design is challenging and typically cannot be accomplished in a single step. It necessitates multiple iterations and a structured series of processes. The design procedure can be broadly divided into two main

categories:

a. Preliminary (or high-level) design
b. and detailed design.

4.1 Types of Design Activity

High-level and detailed design tasks have distinct purposes and scopes depending on the technique employed. High-level design focuses on identifying various modules, along with defining the control linkages and interfaces between them. The output of this phase is known as the software architecture or program structure. Numerous notations are used to represent a high-level design. A common technique to depict the control structure in a high-level architecture is a structure chart, which resembles a tree. Other notations, such as the Warnier-Orr diagram (1977, 1981) or the Jackson diagram (1975), can also be used. In the detailed design phase, the data structures and algorithms for each module are developed. The result of the detailed design phase is typically called the module-specification document.

4.2 Difference between Analysis and Design

The analysis findings are generally conceptual and do not address implementation or platform-specific issues. Typically, graphical formalisms are used to document the analytical model. In the function-oriented approach discussed here, design documentation employs a structure chart, while the analysis model is represented using Data Flow Diagrams (DFDs). In contrast, both the design and analysis models in the object-oriented approach are documented using the Unified Modeling Language (UML). Constructing the analytical model using a programming language would be exceedingly challenging.

The design model evolves from the analysis model through a series of transformations. Unlike the analysis model, the design model incorporates specific decisions about system implementation. The design model must be precise enough to facilitate straightforward implementation using a programming language.

Outcome of the Design Phase (4.1.2):

During the software development phase, several components are designed to ensure the conceptual solution can be effectively implemented in a conventional programming language:
a. Various modules are structured to realize the design solution. b. Control relationships between identified modules, sometimes referred to as call or invocation relationships. c. Interaction interfaces between modules that specify the particular data items exchanged. d. Individual module data structures. e. Algorithms necessary for the implementation of each module.
These components are essential to visualize the system under development and anticipate all required features comprehensively.

4.3 Important Characteristics of the Design Document

Developing a precise description of a robust software architecture that applies universally across different domains is challenging. The notion of a "good" software design can vary depending on the specific application being developed. However, most researchers and software engineers agree on several essential characteristics that a successful software design for general applications should possess. These characteristics include:

a. Correctness: The design should accurately implement all functionalities specified in the SRS document.

b. Understandability: The design should be easy to comprehend.

c. Effectiveness: The design should be efficient in its operation.

d. Changeability: The design should be easy to modify.

4.3 Cohesion

There is a consensus among researchers and engineers that effective software design requires a clear decomposition of the problem into modules and a well-defined placement of these modules within a hierarchy. The fundamental properties of effective module decomposition are high cohesion and low coupling. Cohesion measures the functional strength of a module. A module with high cohesion is functionally independent, meaning it performs a single task or function and has minimal interaction with other modules. Functional independence implies that a cohesive module engages as little as possible with other modules. Various types of cohesion are illustrated in the figure below.

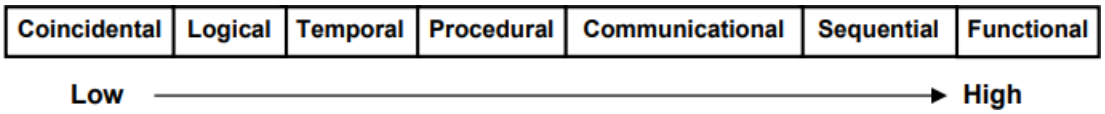

Figure1: Types of Cohesion

Types of Cohesion

Coincidental Cohesion: A module exhibits coincidental cohesion when it performs a series of tasks with minimal connections between them. The module consists of an arbitrary collection of operations, often grouped together without any deliberate planning. For example, in a transaction processing system (TPS), a module might arbitrarily include methods like get-input, print-error, and summarize-members, without any logical grouping related to the problem structure.

Logical Cohesion: A module is logically cohesive when its components perform similar functions, such as error handling, data input, or data output. An example of logical cohesion is a module containing various print functions that generate different types of output reports.

Temporal Cohesion: A module has temporal cohesion if it contains functions that must be executed simultaneously. For instance, functions responsible for initialization, start-up, and shutdown of a process exhibit temporal cohesion.

Procedural Cohesion: Procedural cohesion occurs when a module's functions are part of a specific procedure or algorithm, where steps must be performed in a particular sequence to achieve a goal. An example is a module that implements an algorithm for decoding a message.

Communicational Cohesion: Communicational cohesion is present when a module's functions operate on or update the same data structure, such as an array or a stack.

Sequential Cohesion: A module demonstrates sequential cohesion when its elements form a sequence where the output of one element is the input to the next. For instance, in a TPS, a module might include functions like get-input, validate-input, and sort-input.

Functional Cohesion: Functional cohesion exists when different aspects of a module work together to fulfill a single objective. An example is a module that contains all the functions necessary to manage employee payroll. Describing the functionality of such a module can be done succinctly in one sentence.

4.4 Coupling

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. High interdependence occurs when two modules exchange substantial amounts of data. The complexity of their interfaces determines the degree of coupling. Although there are no precise methodologies for quantitatively estimating coupling between modules, categorizing the types of coupling can help assess their degree of interconnection. Each module can be coupled in one of five ways.

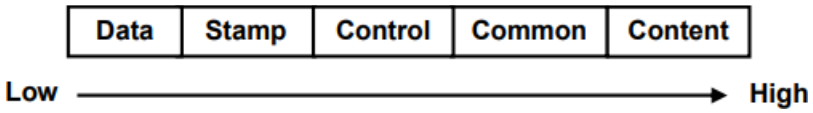

Figure2: Types of Coupling

Types of Coupling

Data coupling occurs when two modules interact through parameters, using simple data items like integers, floats, or characters. These data items should solve specific problems rather than control the flow of the program.

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

Control coupling arises when data from one module dictates the sequence of instructions in another. An example is a flag set in one module that is checked in another.

Common coupling is present when two modules share data through global variables.

Content coupling exists when two modules share code directly, such as when one module branches into another.

4.5 Layered Modular Design Approach

When designing a solution for a specific problem, we often generate a variety of design options. The first step in selecting the best one is to eliminate any faulty designs. How do we then choose the best design from the correct ones?

An ideal design solution should be straightforward and easy to understand. A simple design is not only easier to create but also easier to maintain. In contrast, a complex design can lead to significantly higher life cycle costs. Implementing, testing, debugging, and maintaining a complex design can be extremely labor-intensive. Therefore, simplicity is crucial.

A layered and modular design approach is particularly advantageous due to its simplicity and clarity. This approach divides the system into distinct layers and modules, making it more manageable and easier to understand. By organizing the system into layers, each responsible for specific aspects, and breaking those layers down into modules, the overall complexity is reduced. This facilitates easier implementation, testing, and maintenance, ensuring that the design is robust, adaptable, and cost-effective over its life cycle.

4.3.1 Modularity

A Data Coupling occurs when two modules communicate by passing simple data items as parameters, such as integers, floats, or characters. These data items are used for solving problems and do not control the flow of execution.

Stamp Coupling involves communication between modules using composite data items, like records in Pascal or structures in C, where multiple data items are bundled together.

Control Coupling arises when one module controls the behavior of another module by passing control flags or similar mechanisms. For example, setting a flag in one module that controls the execution flow in another module.

Common Coupling happens when two modules share data via global data items, potentially leading to dependencies and issues with data integrity and maintenance.

Content Coupling exists when modules share code, such as one module branching into another, which can lead to dependencies that complicate understanding and maintenance.

Layered Modular Design Approach (4.3):

In selecting the best design solution from multiple options, the first step is to eliminate any flawed designs. The criteria for choosing the best design often include simplicity and ease of comprehension. A design that is easy to understand is also easier to implement, test, debug, and maintain throughout its lifecycle. Complex designs, on the other hand, tend to incur higher costs and greater effort in these phases.

Modularity (4.3.1):

Modularity is a key aspect of effective design solutions where a problem is divided into modules with minimal connections between them. The divide-and-conquer approach becomes feasible by breaking down the problem into manageable modules. Each module can then be understood independently or with minimal interaction with other modules, thereby reducing the perceived complexity of the overall design.

Measuring the modularity of a design solution objectively can be challenging as there are no standardized metrics for this purpose. However, modularity can be assessed based on the cohesion within modules and the coupling between modules. A highly modular design exhibits strong intra-module cohesion (related elements grouped together) and low inter-module coupling (minimal dependencies between modules).

Effective issue decomposition in software design relies on achieving good cohesion within modules (elements within a module are closely related) and maintaining low coupling (dependencies between modules are minimized), which facilitates easier comprehension, maintenance, and scalability of the system.

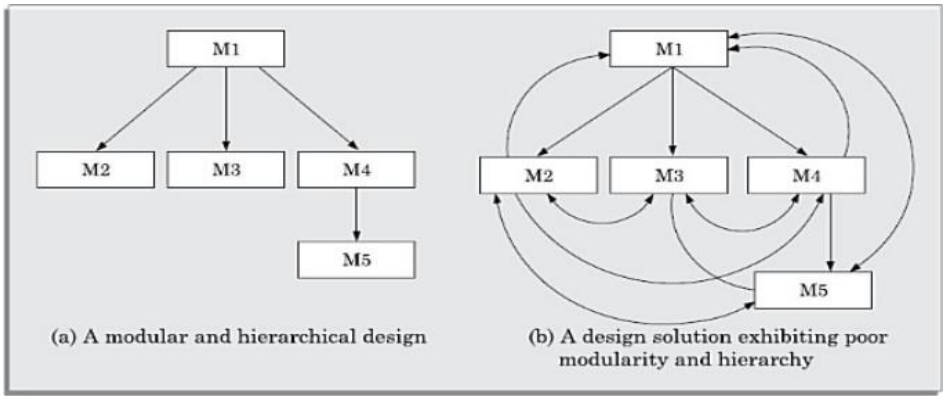

Figure3:Two design solutions for the same problem

As we can say,the1st design solution is easy to understand as compared to another one.

4.3.2 Layered Design Solution

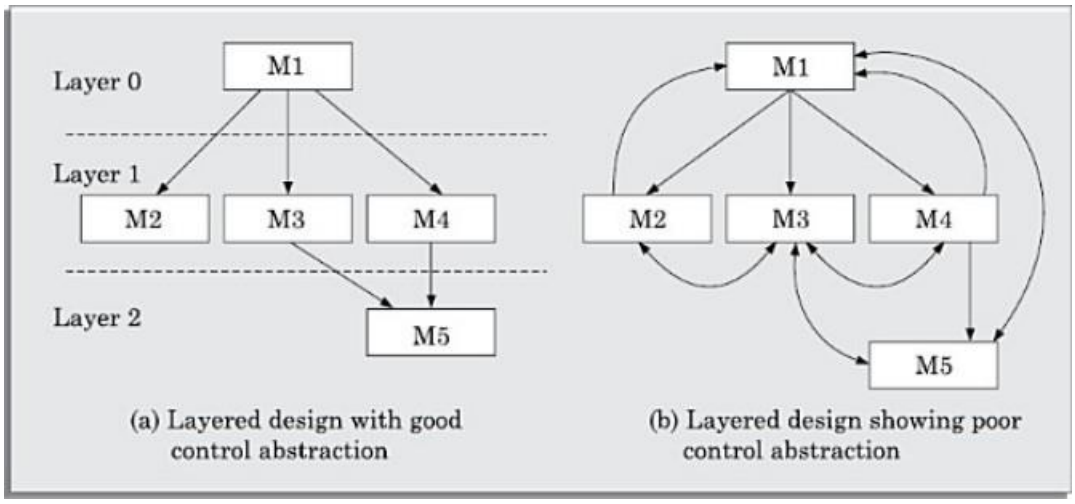

Figure4: Example of good and bad layer abstraction

A layered design is structured like a tree diagram where modules are organized into distinct layers based on their dependencies and call relationships. In this design approach, modules in each layer can only call upon modules in the layer directly beneath them to perform specific tasks. This hierarchical arrangement resembles a managerial structure where higher layer modules instruct or invoke lower layer modules to carry out their designated functions. This concept of layering provides a form of control abstraction.

The benefit of a layered design lies in its simplicity and ease of understanding. Developers typically need to comprehend only how the modules in the immediate lower layer function, without needing detailed knowledge of modules in higher layers. This division of responsibilities makes it easier to manage and maintain the design solution.

Moreover, when debugging a system with a layered design, if an error occurs in a module during execution, it is often straightforward to identify the cause. Errors are typically localized to the module itself or to modules directly below it in the layer hierarchy. This focused approach to debugging reduces the complexity of tracing and fixing issues because the scope of investigation is limited to a smaller set of modules.

In summary, a layered design enhances clarity and manageability by structuring modules into hierarchical layers based on their dependencies and interactions. It simplifies both development and maintenance processes, making it easier to understand, debug, and modify software systems.

4.6 Terminology of Layered Architecture

Types of Module Coupling

Superordinate: A module that exerts control over another module in a control hierarchy is termed superordinate.

Subordinate: A module under the control of another module is termed subordinate.

Visibility: Module B is visible to Module A if A directly calls B.

Control Abstraction: In a layered design, a module should only invoke functionalities of modules in the layer directly beneath it.

Depth and Width: The depth of a control hierarchy indicates the total number of layers, while the width indicates the span of control.

Fan-out: This refers to the number of modules that a given module directly controls. In Figure 4, module M1 has a fan-out of 3. A high fan-out often indicates that a module lacks cohesion, performing multiple distinct functions rather than one unified function. Generally, a fan-out greater than 7 is undesirable.

Fan-in: This refers to the number of modules that directly invoke a specific module. High fan-in suggests code reuse, which is typically desirable. In Figure 4, 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.

Outcome of the Design Process

For a design to be effectively implemented in a typical programming language, the following components need to be created during the design process:

a. Identification of the various modules required for the design solution.

b. Definition of the control relationships between these modules.

c. Specification of the connections between modules, also known as call connections or invocation connections.

d. Description of the interactions between the modules.

e. Identification of the specific data items exchanged between modules via their interfaces.

f. Design of the data architectures for the various modules.

g. Development of the algorithms required to implement each module.

4.7 Summary

- This chapter focuses on the Design Phase of the Software Development Life Cycle (SDLC).
- Software design typically involves two stages: high-level design and detailed design.
- High-level design determines the system's key components (modules) and their relationships.
- Detailed design identifies the data structures and algorithms required.
- Design clarity is emphasized as a crucial criterion for evaluating the quality of a design.
- Design clarity is defined as the effective use of decomposition and abstraction principles.
- Design characteristics are further classified in terms of cohesiveness, coupling, layering, control abstraction, fan-in, fan-out, and other factors.

4.8 Keywords

Cohesion: The Software Requirements Specification (SRS) document stands as one of the most critical and challenging artifacts in the software development lifecycle. Its complexity arises from the diverse range of users it serves, necessitating comprehensive coverage of all user requirements.

Coupling: Rigorous specifications add rigor to software development. The process of crafting a rigorous specification often holds more significance than the formal definition itself. A rigorous specification clarifies nuances of system behavior that may remain ambiguous in informal specifications.

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

Fan-in: Fan-in denotes the number of modules that directly invoke a specific module.

Modularity: Modular design involves breaking down a problem into individual components that achieve high cohesion (components logically related and performing a single task) and low coupling (components have minimal interdependencies).

4.9 Self-Assessment Questions

1. Write down the difference between analysis and design.
2. What is cohesion? Discuss various types of cohesion.
3. Explain layered modular design approach.
4. Discuss various type of coupling.
5. Explain various terminology of layered architecture.

4.10 References:

- I. Sommerville: Software Engineering 10th 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-5

Structure Analysis

Learning Objectives:

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

Structure:

Introduction

5.1 Function-Oriented Design and Object-Oriented Design

Function-oriented design and object-oriented design represent two fundamentally different approaches to software development. Despite their distinct methodologies, they can complement each other effectively rather than existing in isolation. Object-oriented design, while relatively newer and continually evolving, is gaining popularity for its numerous advantages in developing complex software systems. Conversely, function-oriented design enjoys a strong following and is a well-established discipline in software engineering.

Function-Oriented Design

Function-oriented design approaches software development with the following key characteristics:

1. **Task-Centric Approach:** The system is conceptualized as a collection of tasks, where each function begins with a high-level overview of the system and is progressively refined into more detailed functions. For example, the function "create-new library-member" involves tasks such as creating a membership record, assigning a membership number, and generating a membership fee bill. These tasks can further break down into more specific sub-functions.
2. **Centralized System State:** Data within the system, such as member records in a library system, are centralized and shared among various functions. This allows functions like adding new members, deleting existing ones, and editing member records to access and update member data as needed.

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)

These methodologies provide structured approaches to function-oriented design, each offering its own set of principles and techniques for developing software systems based on task decomposition and centralized data management.

5.2 Structure Analysis

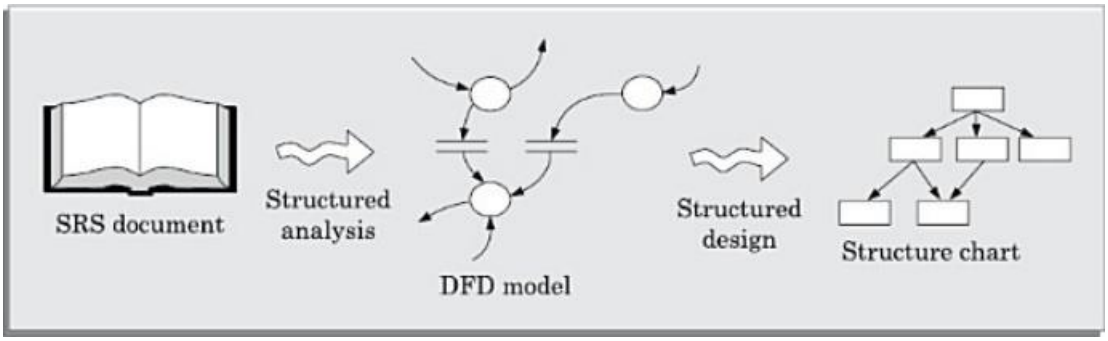

Figure5.1: Structure Analysis and Structure Design

Structured Analysis (SA) and Structured Design (SD)

The figure illustrates the schematic responsibilities of Structured Analysis (SA) and Structured Design (SD) in software development:

- **Conversion from SRS to DFD:** The Software Requirements Specification (SRS) document is transformed into a Data Flow Diagram (DFD) model during the structured analysis phase.
- **Conversion from DFD to Structure Chart:** The DFD model, representing the system's functions and data flows, is further refined into a Structure Chart during structured design. This chart maps out the module structure or high-level design of the software architecture.

  In structured analysis, each function required by the system is systematically broken down into more detailed functions using a top-down decomposition approach. This ensures that all aspects of system functionality are thoroughly analyzed and understood. The resulting DFDs use graphical representations to depict these functions and the flow of data between them.

  Following structured analysis, structured design maps the identified functions to a modular structure. This high-level design phase precedes the detailed design stage, where algorithms and data structures for individual modules are specified. The detailed design is crucial as it directly informs the implementation phase using standard programming languages.

  The structured analysis technique, pioneered by Gane and Sarson [1979] and DeMarco and Yourdon [1978], is grounded in several fundamental principles:

- **Top-Down Decomposition:** Breaking down complex functions into simpler, more manageable tasks.
- **Divide and Conquer:** Each high-level function is further divided into detailed functions to facilitate analysis and design.
- **Use of Data Flow Diagrams (DFDs):** Graphical tools like DFDs are employed to visually represent the analytic outcomes and data flows within the system.

5.3 Data Flow Diagram (DFD)

A Data Flow Diagram (DFD), often referred to as a bubble chart, provides a hierarchical graphical representation of a system's processes. Each process, depicted as a node or bubble, shows the processing activities performed by the system. These processes receive input data and produce output data, illustrating how data flows through the system.

Importance of DFD in a good software design:

Data Flow Diagrams (DFDs), also known as bubble charts, are hierarchical graphical models that depict the various processing activities or functions within a system and the data exchange between these processes. Each function is represented as a processing station that takes input data and outputs data. The system is visualized through its input data, the various processing steps, and the output data. DFDs utilize a minimal set of primitive symbols to represent these functions, making them an intuitive and straightforward tool for modeling systems.

Popularity of DFD Technique

The popularity of DFDs stems from their simplicity and ease of understanding and application. A DFD model organizes numerous sub-functions hierarchically, starting with high-level functions. Hierarchical models are inherently easier to comprehend because they begin with a simple and abstract representation of a system, gradually revealing more details through successive layers. This step-by-step approach aligns with the human cognitive process, making hierarchical models particularly intuitive.

Fundamental Principles of Structured Analysis

Structured analysis techniques, such as those developed by Gane and Sarson (1979) and DeMarco and Yourdon (1978), are based on several fundamental principles:

- **Top-Down Decomposition**: Breaking down the system from a high-level overview into detailed parts.
- **Divide and Conquer**: Each high-level function is subdivided into more detailed functions.
- **Data Flow Diagrams (DFDs)**: Using DFDs to graphically represent the analysis results.
- **Drawbacks of DFD Models**
- Despite their advantages, DFD models have several drawbacks:
- **Potential for Errors**: The purpose of a bubble is determined by its label, which may not fully convey all functionalities. For example, a bubble labeled "find-book-position" might not specify what happens if input data is missing or incorrect, or if the book is not found.
- **Lack of Control Aspects**: DFDs do not define control aspects such as the order in which inputs and outputs are processed or the sequence of bubble execution, which is crucial for real-time systems.
- **Subjectivity in Decomposition**: The decomposition process and the level of detail to which functions are broken down are subjective and depend on the analyst's judgment. This can lead to multiple, equally valid DFD representations for the same problem, with no clear way to determine which is better.
- **Unclear Decomposition Guidance**: Data flow modeling does not provide explicit guidance on how to break down a function into subfunctions, requiring reliance on the analyst's judgment.

Common Errors in DFD Design

Some common mistakes in DFD design include:

- **Multiple Bubbles in Context Diagram**: A context diagram should represent the system as a single bubble.
- **External Entities at All Levels**: External entities should only appear in the context diagram, not at other levels.

- **Imbalance in DFD Levels**: Ensuring balanced levels with an appropriate number of bubbles (typically 3 to 7 per diagram) is crucial.
- **Improper Data Store Connections**: Data stores should only connect to bubbles via data arrows, not directly to other data stores or external entities.
- **Incomplete Functionality Representation**: The DFD model must represent all system functions described in the SRS document without introducing assumed functionalities.
- **Unclear Naming**: Functions and data should have clear, understandable names, avoiding symbolic names like a, b, c, which obscure the DFD's meaning.

5.4 Structured Design

The goal of structured design is to transform a DFD representation of the structured analysis results into a structured chart. There are two primary methods to guide this transformation:

- **Transform Analysis**
- **Transaction Analysis**

Transform Analysis

Transform analysis identifies the key functional components (modules) and their high-level inputs and outputs. The process involves:

1. **Segmenting the DFD**: Dividing the DFD into input, processing, and output sections.
2. **Creating a Structure Chart**: Drawing one functional component for the central transform, as well as the afferent and efferent branches, beneath a root module.
3. **Factoring**: Adding sub-functions required by each high-level functional component, including read/write modules, error-handling modules, initialization and termination processes, and customer modules.

Transaction Analysis

Transaction analysis is useful for transaction processing programs where different paths are taken through the DFD based on input data. It involves:

1. **Identifying Transactions**: Tracing the input data to the output for each transaction.
2. **Creating Transaction Modules**: Drawing a root module and modules for each identified transaction, using tags to distinguish transaction types.

Structure Chart

A structure chart represents the software architecture, showing the system's components, their dependencies, and the parameters exchanged between them. It includes:

- **Rectangular Boxes**: Represent modules.
- **Module Invocation Arrows**: Indicate control transfer direction.
- **Data Flow Arrows**: Show data flow direction between modules.
- **Library Modules**: Depicted with double-edged rectangles.
- **Diamond Symbols**: Represent selection.
- **Repetition Loops**: Indicate repetition of control flow.

5.5 Summary

- **Structured Analysis and Design (SA/SD)**: A technique that combines elements of various design methodologies to create a good design meeting multiple criteria.
- **Functional Decomposition**: The aim of structured analysis, using DFDs to represent the results.
- **Structured Chart Representation**: Created by transforming DFDs, easily implemented using conventional programming languages.

5.6 Keywords

- **Structure Charts**: Represent software architecture, showing module dependencies and parameter exchanges.
- **Structured Analysis**: Captures the detailed structure of a system using the user's vocabulary.
- **DFD**: A powerful modeling approach for depicting the results of structured analysis and other purposes like organizational document flow.
- **Transaction Analysis**: A method for creating transaction processing programs by identifying and representing different transaction paths.
- **Transform Analysis**: Identifies key functional components and segments the DFD into input, processing, and output sections.

5.7 Self-Assessment Questions:

1. What do the terms "structured analysis" and "structured design" mean to you?
2. What are the primary goals of "structured analysis" and "structured design?"
3. Explain how the DFD model helps one understand how the system works.

4. What do the terms" transform analysis" and "transactional analysis" mean to you?
5. What are the primary drawbacks of using a data flow diagram (DFD) to do structured analysis?

5.8 Case study

Case Study: Function-Oriented Design in Software Development

Introduction: XYZ Solutions specializes in developing enterprise-level applications across various industries. Recently, the company 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 highlight 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 alignment of project requirements and goals with function-oriented design principles.
3. Establish clear guidelines and standards for implementing function-oriented design in the project.

Questions to be considered:

a) **Identification and Definition of Functions:** The functions of the transportation management system were identified through comprehensive analysis of system requirements. Core functionalities were defined based on the needs of the logistics company.

b) **Methodologies and Techniques:** Structured techniques such as structured analysis, data flow diagrams (DFDs), and entity-relationship diagrams (ERDs) were employed to achieve function-oriented design. These methodologies helped in modeling and defining the system's functions systematically.

c) **Benefits and Challenges:** Benefits:

- Enhanced clarity and understanding of system functionalities.
- Improved modularity and maintainability of the software.
- Facilitated easier management of future changes and enhancements.

Challenges:

- Initial learning curve for the development team unfamiliar with function-oriented design.
- Ensuring all stakeholders understand and support the approach.

Recommendations:

1. Conduct regular reviews and walkthroughs to validate function-oriented design and ensure alignment with project objectives.
2. Establish metrics to measure the effectiveness and efficiency of function-oriented design implementation.
3. Encourage the adoption of function-oriented design in future projects to leverage its benefits.
4. Invest in training and upskilling of the development team in function-oriented design principles and methodologies.

Benefits of Function-Oriented Design:

- Contribution to project success through enhanced system functionality and maintainability.
- Specific benefits observed included clearer system structure and easier integration of new features.
- Function-oriented design facilitated effective management of system complexities and scalability.

Conclusion: Function-oriented design proved instrumental in the successful development of the transportation management system by XYZ Solutions. By continuing to embrace and refine function-oriented design practices, the company can further enhance software quality and maintainability in future projects.

5.9 References:

- I. Sommerville: Software Engineering, 10th 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

Unified Modeling Language (UML)

Learning Objectives:

- Describe what a model is.
- Describe how model scan 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 compare end the many forms of UML diagrams

Structure:

Introduction

Object-Oriented Software Development methodology enjoys widespread popularity in both industry and academia. Since its humble beginnings in the early 1980s, object technologies have significantly evolved and advanced. Today, they are extensively utilized for developing large-scale programs due to their numerous inherent benefits.

Initially emerging in the 1990s, object technology quickly gained prominence and has since matured considerably. Given its pervasive use and popularity, a solid understanding of this technology is essential. Proficiency in object-oriented programming languages such as Java or C++ is crucial for developing high-quality Object-Oriented Software.

This methodology continues to be favored for its ability to enhance code reusability, promote modular design, and facilitate easier maintenance and scalability of software systems. As object technologies continue to evolve, their impact on software development practices remains profound and enduring.

6.1 Object-Oriented Design

The basic concepts of object-oriented are shown in the figure. We will discuss all these concepts in the upcoming sub section**.**

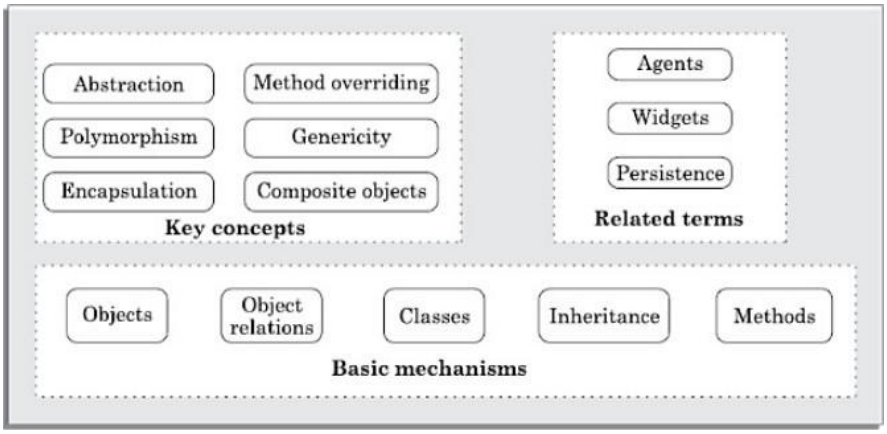

Figure6.1: Object-Oriented Design Concepts

Object

An object in programming is a representation of a real-world entity, encapsulating related attributes and methods that operate on its data. Objects ensure that the attributes are private, and operations on the data are performed through defined methods. Examples of real-world entities that can be represented as objects include books, cars, houses, people, accounts, and more.

When using object-oriented approaches, it becomes easier to conceptualize software as a collection of interacting objects. For instance, in a manually operated library system, before a book is issued, an entry must be made in the issue register and a return date stamped. Similarly,

in an object-oriented library automation system, interactions would occur between a book object and an issue register object to perform these tasks. Each object would contain related data and methods for operating on that data.

Example: Library Member Object

Consider a library member in an automated library system. The library member can be represented as an object, encapsulating various attributes and methods.

Private Data of a Library Member Object:

- **Member's name**
- **Membership number**
- **Address**
- **Phone number**
- **Email address**
- **Date of admission as a member**
- **Membership expiry date**
- **Books currently checked out**

Methods to Operate on the Data:

- **Issue Book**: Issues a book to the member.
- **FindBooksOutstanding**: Lists all books currently checked out by the member.
- **Find Books Overdue**: Identifies books that are overdue.
- **Return Book**: Processes the return of a book.
- **Find Membership Details**: Retrieves the member's details.

Class

A class serves as a blueprint for creating objects. Once a class is defined, you can instantiate multiple objects from it, each with the same properties and methods. For example, all objects created from a Library Member class will have properties like member Name, email, joining Date, and expiry Date, along with methods such as issue Book, return Book, and find Book. Essentially, a class is a user-defined data type.

Methods

In a class, methods define the operations that can be performed by an object. For example, in a Library Member class, the methods might include actions like issue Book, return Book, and find Book.

Encapsulation

Encapsulation refers to the concept of binding together the related attributes and methods within a class. This approach helps in bundling the data (attributes) and the methods that operate on the data into a single unit or class, providing a clear structure and security by hiding the internal state of the object from the outside world.

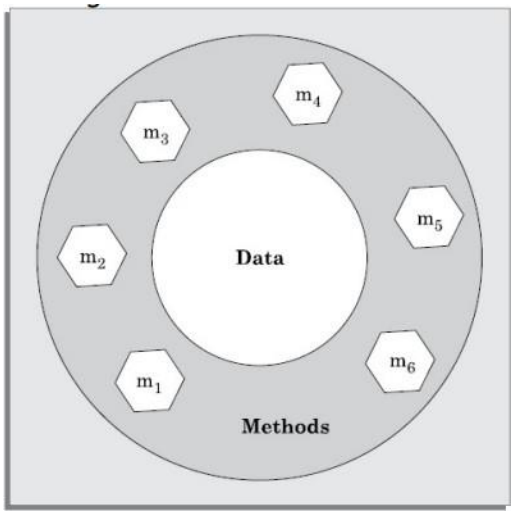

Figure6.2: Encapsulation

Inheritance

Inheritance is a fundamental concept in object-oriented programming that enables the creation of a new class based on an existing class. The class from which a new class is derived is known as the superclass, while the newly created class is called the subclass.

The key principle of inheritance is that the subclass inherits the properties (attributes and fields) and behaviors (methods) of the superclass. This means that the subclass can reuse and extend the functionalities defined in the superclass without needing to redefine them.

By inheriting from a superclass, the subclass gains access to all public and protected members of the superclass. This allows for code reuse, enhances modularity, and supports the concept of

hierarchical classification in programming. Inheritance facilitates building relationships between classes and promotes efficient software development by reducing redundancy and promoting code organization and reusability.

Overloading

The same method with a different input parameter list is known as method overloading

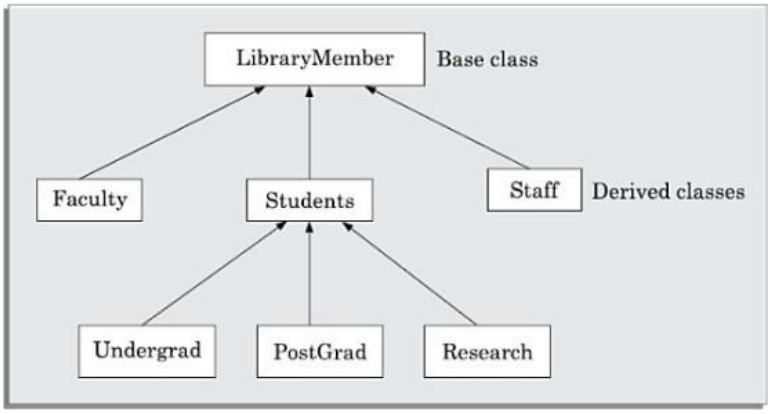

Figure6.3. Library Management System

Overriding

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

Polymorphism

Polymorphism allows a single method or function to behave differently in different contexts. This concept enables objects to be used interchangeably when they share a common interface, but implement the interface in different ways.

Abstraction

Abstraction in programming involves hiding the complex implementation details of a system and exposing only the necessary functionality. When creating a class, this principle allows us to present only the relevant information to the user, while hiding the underlying complexity.

6.2 Unified Modeling Language (UML)

UML is a visual modeling language used to describe the structure, behavior, and architecture of software systems. It uses various notations such as rectangles, lines, and ellipses to create visual representations of system components and their relationships. UML is not a system development method per se, but it is a valuable tool for visualizing and documenting object-oriented designs.

Emergence of UML

In the late 1980s and early 1990s, various object-oriented design techniques and notations were developed, leading to a proliferation of different modeling languages. This variety often caused confusion within the software development community. To address this, UML was developed to standardize the different object-oriented modeling notations that were in widespread use. Key methodologies that contributed to UML's development include:

- Object Management Technology [Rumbaugh, 1991]
- Booch's methodology [Booch, 1991]
- Object-Oriented Software Engineering [Jacobson, 1992]
- Odell's methodology [Odell, 1992]
- Shlaer and Mellor methodology [Shlaer, 1992]

UML has since become a widely accepted standard in software engineering for visualizing software architecture and design.

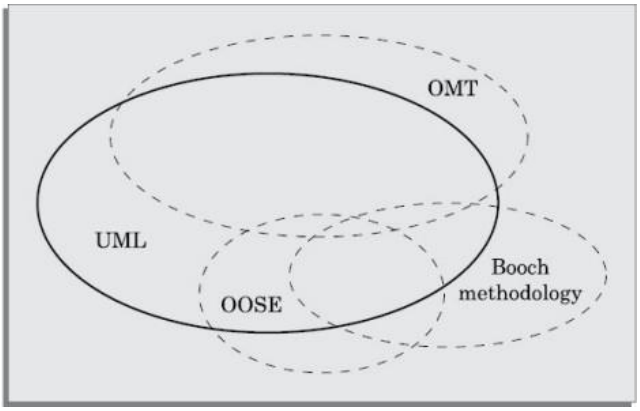

Figure 6.4: Impact of different modelling techniques on UML

UML is adopted by OMG(Object Management Group) and accepted in the wide software

industry. UML can be used in both small and large problems. Now UML is used worldwide for modelling system design.

6.3 UML Diagrams

UML (Unified Modeling Language) employs nine different types of diagrams, each designed to capture different aspects or views of a system. These diagrams help in understanding the system from various perspectives, which are crucial for comprehensive analysis and design. UML diagrams can represent the following five views of a system:

1. **User's Perspective**: This view focuses on how users will interact with the system, emphasizing user interfaces and user experience.
2. **Structural Perspective**: This view illustrates the static aspects of the system, such as the system's architecture and the relationships between its components.
3. **Behavioral Perspective**: This perspective describes the dynamic aspects of the system, including how the system behaves and how its components interact over time.
4. **Implementation Perspective**: This view shows how the system will be implemented, detailing the physical aspects of the system's components, such as modules, classes, and processes.
5. **Environmental Perspective**: This view considers the system's interaction with its external environment, including other systems, hardware, and external agents.

These different perspectives provided by UML diagrams help in capturing the complexity of a system and facilitate effective communication among stakeholders in the software development process.

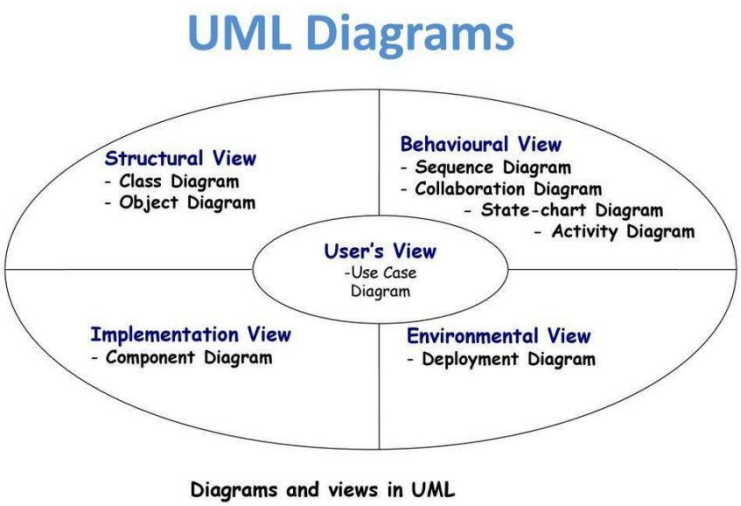

Figure 6.5: UML Diagrams and Different Views Structural View

6 The structural view in software design focuses on identifying the necessary objects (or classes) and their relationships to understand and implement a system. This view remains static over time, reflecting the system's foundational structure, which is why it's also referred to as the static model.

7 In contrast, the behavioral view captures the interactions between objects or components to depict the dynamic behavior of the system. This view is essential for understanding how the system functions over time, capturing its time-dependent aspects.

8 The implementation view details the system's primary elements and their interconnections, providing insights into how the components are integrated to achieve the system's functionality.

9 The environmental view illustrates how the system components are deployed across various hardware pieces or environments, highlighting the physical deployment aspects of the system.

10 The user view, distinct from other views, focuses on the system's capabilities and functionalities as perceived by its external users. It presents a black-box perspective where users interact with the system based on its functionalities, without needing to understand its underlying structure, dynamic behavior, or implementation details. This

user-centric view guides the development process to ensure that the system meets the functional needs and expectations of its users effectively.

6.4 Use Case Diagrams

The structural view in UML outlines the essential objects or classes needed to understand and implement a system, along with the relationships between these classes. This model, known as the static model, remains consistent over time as it reflects the system's structure.

Behavioral View

The behavioral view captures the interactions between components to understand the system's dynamic behavior, showing how elements interact over time to produce observable effects.

Implementation View

This view illustrates the primary elements of the system and their interconnections, providing insight into the physical arrangement and dependencies within the system's architecture.

Environmental View

This view depicts how the system's components are deployed across various hardware platforms, illustrating the physical deployment environment.

User View

The user view defines the system's capabilities from the user's perspective, presenting how users interact with the system without revealing the internal structure or implementation details. This view is crucial in user-centric development approaches, where the focus is on the functionality from the user's point of view.

Use Case Diagrams

Use case diagrams represent the interactions between users and the system, detailing the different ways users can engage with the system. Common use cases in systems like a Library Information System might include actions such as issuing books, querying books, returning books, creating members, and adding books. These use cases define the system's behavior from a high-level functional perspective, breaking down the system's operations into manageable transactions.

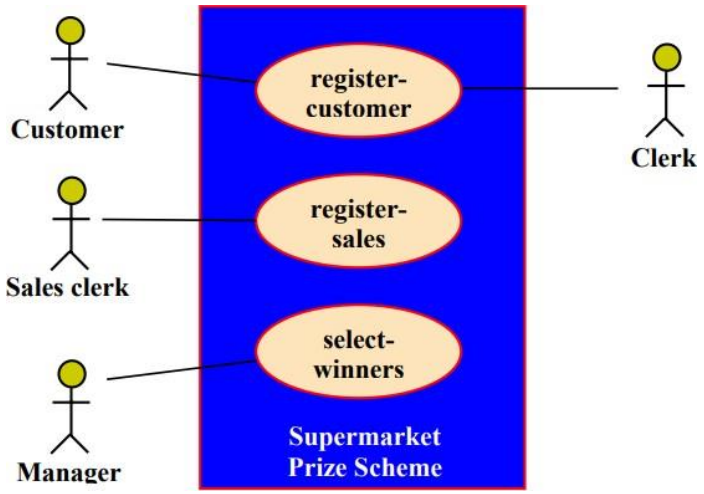

Figure 6.6: Use Case Diagram for Supermarket

The use case model for the Supermarket Prize Scheme includes three primary use cases: "register-customer," "register-sales," and "select-winners." Below is a detailed description of each use case along with their respective scenarios:

U1: Register Customer

This use case allows customers to register by providing necessary information.

Scenario 1: Standard Registration Process

1. **Customer:** Selects the "Register Customer" option.
2. **System:** Presents a popup for the customer to enter their name, address, and phone number.
3. **Customer:** Fills in the required fields with the correct data.
4. **System:** Displays a generated ID and a notification confirming the successful registration.

Scenario 2: Duplicate Registration Attempt

1. **System:** Indicates that the customer has already registered when attempting to register again.

Scenario 3: Missing Information

1. **System:** Alerts the customer that some required input data has not been submitted and prompts for the missing information.

U2: Register Sales

This use case enables clerks to record details of a customer's purchase.

Scenario 1: Recording a Sale

1. **Clerk:** Chooses the "Register Sales" option.
2. **System:** Requests the customer's ID and details of the purchase.
3. **Clerk:** Enters the necessary information.
4. **System:** Provides a notification confirming that the sale has been successfully registered.

U3: Select Winners

This use case allows the manager to generate a list of winners.

Scenario 1: Selecting Winners

1. **Manager:** Selects the "Select Winner" option.
2. **System:** Displays the list of winners, including those receiving a gold coin and surprise gifts.

These use cases describe the interactions between the users (customers, clerks, managers) and the system, outlining the steps each participant takes to complete their tasks within the supermarket's prize scheme.

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—generalisation, aggregation, association, and other types of dependencies. We will now go through the UML syntax for representing the classes and their relationships.

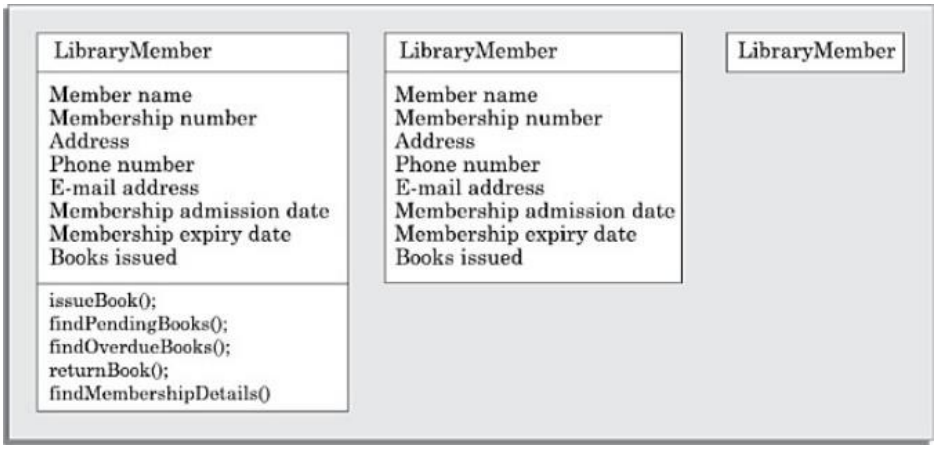

Figure 6.7: Different Representation of a Class

U1: Register Customer

This use case enables customers to register by providing essential details.

Scenario 1: Standard Registration Process

1. **Customer** selects the "Register Customer" option.
2. **System** prompts the customer to enter their name, address, and phone number.
3. **Customer** completes the fields with the required information.
4. **System** generates a unique ID and confirms the registration with a notification.

Scenario 2: Attempting to Register Again

1. **System** detects an existing registration attempt and alerts the user that they are already registered.

Scenario 3: Incomplete Information

1. **System** notifies the customer about missing required information and requests the necessary details.

U2: Register Sales

This use case allows clerks to log details of customer purchases.

Scenario 1: Logging a Sale

1. **Clerk** selects the "Register Sales" option.
2. **System** prompts for the customer's ID and details of the purchase.
3. **Clerk** inputs the required information.
4. **System** confirms the successful registration of the sale with a notification.

U3: Select Winners

This use case enables the manager to compile a list of prize winners.

Scenario 1: Compiling the Winners List

1. **Manager** selects the "Select Winner" option.
2. **System** shows the list of winners, including those who will receive gold coins and surprise gifts.

These use cases detail the interactions between different users (customers, clerks, managers) and the system, specifying the steps each user takes to fulfill their responsibilities within the supermarket's prize scheme.

Association

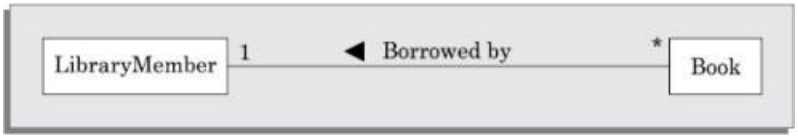

Figure 6.8: Association between 2 Classes

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 might be added on the association line to show the association's reading direction. 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, 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.

In the above diagram, an asterisk (*) denotes a wildcard that signifies one or many instances. The figure illustrates that many books can be borrowed by a library member.

Aggregation

Aggregation in software design refers to a relationship where classes are connected, and there exists a whole-part relationship between them. This means that an aggregate object not only maintains references to its parts but also takes responsibility for their creation and deletion. For example, a book registry can be seen as an aggregation of book objects, allowing books to be added or removed from the registry as needed.

In diagrams depicting aggregation, an empty diamond symbol is used at the aggregate end of the relationship. The number '1' annotated at the diamond end signifies that a single document can include numerous paragraphs, while the *'' annotated at the other end indicates that a document can have multiple paragraphs. If, for instance, the intention is to specify exactly ten paragraphs in a document, '10' would replace the ".*

This approach clarifies how objects can be structured hierarchically, such as viewing a document as composed of paragraphs, each paragraph in turn being composed of lines. It emphasizes the flexibility and hierarchical nature of aggregation in defining complex relationships between objects in software systems.

Composition

Composition is a more stringent form of aggregation where the parts are intrinsically dependent on the whole for their existence. This means that the parts cannot exist independently without the whole. In essence, the lifecycle of the whole entity and each of its elements are tightly coupled. When the whole entity is created, its parts are created as well, and when the whole entity is destroyed, all its parts are also destroyed.

An example illustrating composition is an order object in a system. In this case, the order object and its order items exhibit a composition relationship. Once an order is placed, no individual item within the order can be altered independently. If any modifications are required to the items in the order, the entire order must be canceled, and a new order containing the revised items needs to be placed. Therefore, when an order object is instantiated, all its associated order items are also instantiated. Similarly, when the order object is deleted or canceled, all its order items are also removed from the system.

In summary, composition ensures that the parts (order items) are integral to the existence of the whole (order), and their lifecycles are managed together, reflecting a strong relationship where the whole controls the existence and management of its parts.

A full diamond drawn at the composite end represents the composition relationship.

6.6 Sequence Diagram

6.7

A sequence diagram is a graphical representation used to illustrate the interactions between objects in a system. It consists of two dimensions and is read from left to right. At the top of the diagram, rectangular boxes represent the objects involved in the interaction. Each box contains the name of the object, written in the format "objectName : ClassName", where "objectName" refers to any specific instance of the class denoted by "ClassName".

Objects appearing at the top of a sequence diagram indicate that they existed before the start of the use case execution. Objects created during the execution of the use case, which participate in interactions such as method calls, are positioned at the moment of their creation on the diagram.

The vertical dashed lines extending downwards from the object boxes represent the lifelines of the objects. The presence of an object on its lifeline at any point signifies its existence at that particular time during the sequence. When an object ceases to exist, typically after completing its purpose or when the use case concludes, its lifeline is ended with a cross.

Activation symbols, depicted as rectangles placed on an object's lifeline, denote the times when the object is actively engaged in processing or participating in interactions. The presence of an activation symbol indicates that the object is in an active state for the duration that the rectangle is visible on its lifeline.

In summary, a sequence diagram provides a visual overview of how objects interact in a system, showing the sequence of messages exchanged between them over time and illustrating the lifetimes and activations of participating objects during the interaction.

6.7 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 across numerous use case executions. State chart diagrams, however, are inappropriate if we want to depict behavior that involves multiple objects working together. As we've already seen, sequence or collaboration diagrams are more suited to modeling this type of conduct.

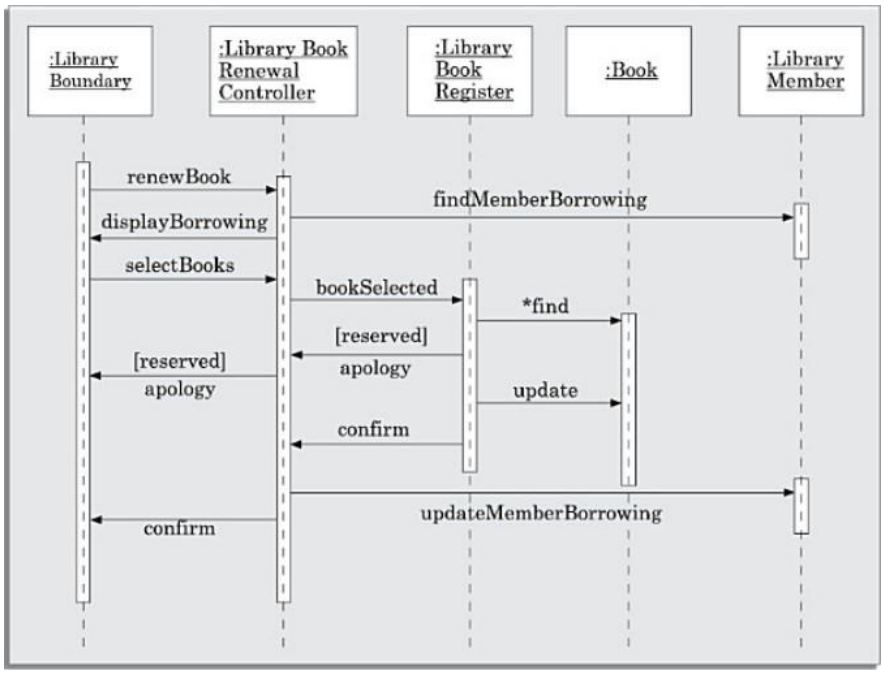

Figure 6.9: State Chart

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.

Basic elements: 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: is represented by an arrow between two states. The name of the event that generates the transition is usually placed alongside the arrow. You can also have a guard stand guard throughout the transition. A guard is a condition in Boolean logic. Only if the guard evaluates to true may the transfer occur. The syntax for the transition label is shown in three parts—[guard]event/action.

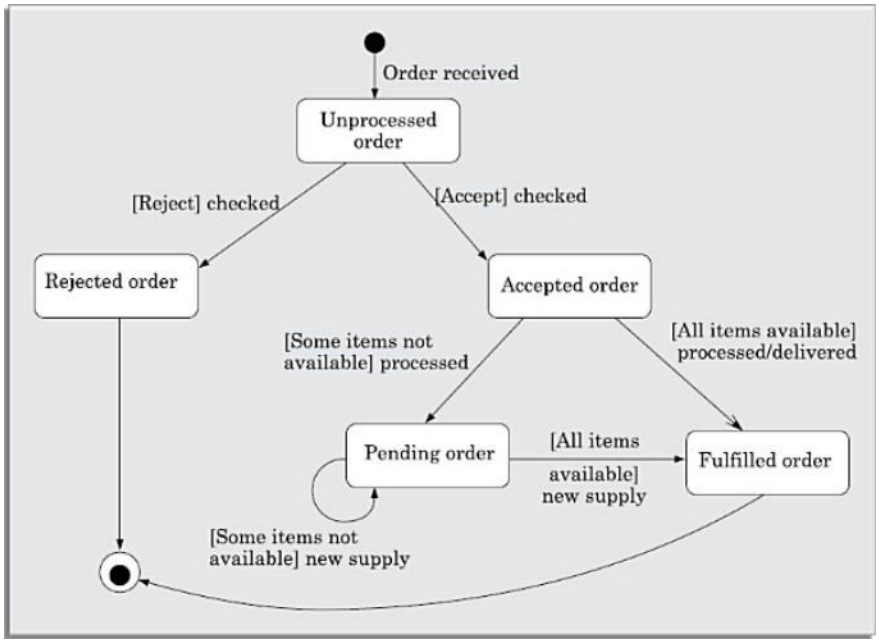

Figure 6.10: State Chart diagram for order

6.8 Summary

- We began this chapter by reviewing essential concepts related to object orientation.
- A major advantage of object orientation is its ability to significantly enhance productivity in software development teams.
- The increased productivity in object-oriented projects can largely be attributed to the reuse of predefined classes and partial reuse through inheritance, as well as the conceptual clarity provided by the object-oriented approach.

- Object modeling plays a crucial role in system analysis, design, and understanding. UML has become widely popular and is increasingly recognized as a standard tool for object modeling.
- We discussed the syntax and semantics of several common types of diagrams that can be created using UML. We'll explore an object-oriented system development process that utilizes UML for model documentation.

6.9 Keywords

- **Use Case**: A collection of "use cases" forms the use case model for any system, reflecting the various ways that users can interact with the system.
- **Sequence Diagram**: This diagram illustrates interactions between objects in a two-dimensional format. It shows the lifeline of an object as a vertical dashed line, with messages between objects represented as arrows connecting these lifelines, displayed in the order they occur from top to bottom.
- **State Chart**: A state chart diagram is used to represent how an object's state changes over time, effectively illustrating how an object's behavior evolves across different use case executions.
- **Aggregation**: Aggregation is a type of relationship where the classes involved represent a whole-part relationship. In this setup, the aggregate is responsible for delegating messages to the appropriate parts, assuming leadership in the process.
- **Association**: Associations are necessary for objects to interact with each other. An association describes the relationship between two classes, also known as an object connection or link, which can be mental or physical.

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 withdrawal 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 examples

6.11 Case Study

Introduction: In a software development project, effectively implementing various use case views is crucial for capturing and understanding the functional requirements of the system. However, several challenges hinder this process. Let's explore a case study and provide recommendations to address these issues.

Problem: A software development team is facing challenges in implementing different use case views for their project. These challenges include:

1. Incomplete and ambiguous use case descriptions: The descriptions lack necessary details and clarity, leading to misinterpretation and confusion among team members.
2. Lack of user-centric perspective: Use cases do not adequately focus on the needs and goals of the system's end-users, causing a disconnect between the system's functionality and user expectations.
3. Inconsistent and disconnected use case diagrams: Diagrams fail to effectively represent relationships between use cases and actors, lacking coherence and clarity, which complicates understanding of the overall system behavior.

Recommendations:

1. Conduct thorough requirements gathering:
    - Engage stakeholders, including end-users and domain experts, to gather comprehensive requirements.
    - Key questions to ask: What are the system's primary goals and objectives? Who are the main actors interacting with the system? What are the core functions or tasks the system should perform?
2. Apply user-centered design principles:

- Define use cases from a user-centric perspective.
- Questions to address: How will the system meet the needs and expectations of end-users? Are there specific user roles or personas to consider? What are the primary user interactions and workflows within the system?

3. Create detailed use case descriptions:
   - Document use cases with clear and detailed descriptions.
   - Address questions: What are the preconditions and postconditions of each use case? What are the primary and alternative flows of events? How should exceptional or error conditions be handled?
4. Validate and refine use case descriptions:
   - Collaborate with stakeholders and subject matter experts to validate and refine use case descriptions.
   - Questions for validation: Are the descriptions accurate and complete? Do they reflect the intended system behavior and user requirements? Are there any missing or conflicting use cases to address?
5. Improve use case diagram representations:
   - Enhance clarity and coherence of use case diagrams.
   - Questions to consider: Do the diagrams accurately represent relationships between actors and use cases? Are use cases organized and grouped logically and meaningfully? Can diagrams be easily understood by all stakeholders?
6. Utilize UML tools and templates:
   - Use UML modeling tools and predefined templates to facilitate creation and documentation of use case views.
   - These tools ensure standard notations and symbols, enhancing consistency and professionalism in diagrams.
7. Conduct regular reviews and walkthroughs:
   - Schedule regular sessions to review use case views with stakeholders and development team members.
   - Use these sessions to gather feedback, clarify ambiguities, and ensure a shared understanding of system requirements.

By implementing these recommendations, the software development team can overcome challenges in implementing different use case views. This approach will result in comprehensive and well-defined use cases that accurately represent the system's functional requirements and align with user expectations.

6.12 References:

- I. Sommerville: Software Engineering 10th 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 - 7

Coding Standards

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 begins once the design phase is completed and the design documents have been thoroughly reviewed. During this phase, each module outlined in the design documentation is developed and subjected to unit testing. Unit testing involves evaluating each module in isolation, ensuring that it functions correctly on its own. This means that after the coding of a module is finished, it is immediately tested independently from other modules.

Following the completion of coding and unit testing for all system modules, the integration and system testing phase commences. The input for the coding phase is the design document

prepared at the end of the design phase. This document includes both a high-level system design, typically presented as a module structure (e.g., a structure chart), and a detailed design. The functional architecture is usually described in terms of module requirements, which specify the data structures and algorithms for each module. Throughout the coding phase, the various modules identified in the design document are developed according to their respective module requirements.

7.1 Coding Standard

Developers are required to adhere to coding standards throughout the coding process. During code inspection, compliance with these standards is verified. Any code that does not meet the established coding standards is rejected during the review process and must be rewritten by the responsible developer. Although coding standards provide general guidelines regarding coding style, the specific implementation is left to the discretion of individual developers.

Reputable software development companies often establish their own coding standards and rules, tailored to what is most effective for their organization and the nature of their products.

The following are some examples of coding standards.

a. Rules for Limiting the Use of Global Variables: These rules define which types of data are permitted or prohibited from being declared as global.

b. Contents of Module Headers: The headers preceding the code for various modules should adhere to organization-wide standards. A specific format for organizing header information can be defined. Standard header data typically includes:

- The module's name.
- The date of development.
- The author's name.
- Modification history.
- A brief synopsis of the module.
- Functions supported by the module and their input/output parameters.
- Global variables accessed or modified by the module.

c. **Naming Conventions for Variables:**

- Global variable names should start with a capital letter.

- Local variable names should begin with a lowercase letter.
- Constant names should be written in uppercase letters.

d. **Error Return Standards and Exception Handling Mechanisms:** The method of reporting and handling errors should be consistent across the organization. For instance, functions should uniformly return a 0 or 1 when an error condition is encountered.

Coding Guidelines

a. **Maintain a Clear Coding Style:** Code should be easy to understand. Avoid overly complex or obscure coding practices, as these can obscure the meaning of the code and complicate maintenance.

b. **Be Aware of Function Side Effects:** Function calls can have side effects such as modifying parameters passed by reference, changing global variables, or performing I/O operations. These side effects should be clear to prevent misunderstandings about the code's behavior.

c. **Avoid Reusing Identifiers for Multiple Purposes:** Do not use the same identifier for different temporary entities. This practice, often justified by memory efficiency, can lead to confusion and errors.

d. **Document the Code Properly:** A general guideline is to include at least one comment line for every three lines of source code.

e. **Limit Function Length:** Each function should ideally have no more than ten source lines to maintain clarity and reduce the likelihood of bugs.

f. **Use Go to Statements Sparingly:** Excessive use of go to statements can make the program disorganized and difficult to follow.

7.2 Coding Walkthrough

A code walkthrough is an informal method of code review conducted after the module has been coded, compiled, and syntax errors have been resolved. Team members review the code individually, selecting test cases and tracing the execution of statements and functions. The primary goal is to identify algorithmic and logical issues.

Effective code walkthroughs follow these guidelines:

- The team should consist of three to seven members.
- The focus should be on identifying errors, not on solving them.

- Managers should not attend to foster teamwork and avoid making engineers feel judged.

7.3 Code Inspection

In contrast to code walkthroughs, which focus on hand-simulating the execution of code to uncover algorithmic and logical issues, the primary objective of code inspections is to identify common types of problems resulting from poor programming practices or misunderstandings. Rather than manually tracing the execution of code, code inspections involve systematically checking for specific types of errors. For example, a common mistake such as writing a procedure that alters a formal parameter while the calling procedure uses an actual constant parameter is more likely to be caught through inspection than through manual simulation.

Code inspections also verify adherence to coding standards and document common errors. Leading software development companies track the types of errors frequently made by their engineers to compile a list of common mistakes. This list can then be used during code inspections to spot potential issues. Some typical programming errors that can be identified during code inspection include:

- Uninitialized variables.
- Jumps into loops.
- Non-terminating loops.
- Incompatible assignments.
- Out-of-bounds array indices.
- Improper memory allocation and deallocation.
- Misalignment of actual and formal parameters in procedure calls.
- Incorrect logical operators or inappropriate operator precedence.
- Incorrect modification of loop variables.
- Improper comparison of floating-point variables, etc.

7.4 Documentation

In software development, creating not only executable files and source code but also various documents such as user manuals, software requirements specifications (SRS), design documents, test documents, and installation manuals is crucial. These documents are essential for several reasons:

- They enhance the accessibility and reliability of the software product, reducing maintenance effort and time.
- They help users efficiently utilize the system.
- They facilitate effective management of labor turnover by allowing new engineers to quickly acquire necessary knowledge.
- They assist project managers in tracking progress, as completed work is documented and evaluated.

Types of Software Documentation

Software documentation can be broadly categorized into:

- Internal Documentation
- External Documentation

Internal Documentation

Key types of internal documentation include:

- Comments within the source code.
- Meaningful variable names.
- Headers for packages and functions.
- Proper code indentation.
- Well-structured code (modular breakdown).
- Use of enumerated types.
- Use of constant identifiers.
- Utilization of user-defined data types.

Internal documentation enhances code comprehension by including module headers, comments, meaningful variable names, and proper code structuring. Studies have shown that meaningful variable names significantly aid code understanding, even more so than comments. For instance, comments like a = 10; /* a assigned 10 */ are not helpful. Strong software development organizations enforce robust internal documentation by establishing appropriate coding standards and guidelines.

External Documentation

External documentation includes user manuals, software requirements documents, design documents, and test documents. These documents must be created promptly and maintained to high standards to ensure compatibility with the code. Inconsistencies between documents and code can lead to confusion and hinder understanding. All documents should be up-to-date, reflecting any modifications in the code. Clear, understandable documentation tailored to its intended audience is critical. One way to measure readability is the Gunning Fog Index.

7.5 Gunning Fog Index

The Gunning Fog Index, developed by Robert Gunning in 1952, assesses the readability of a document. It estimates the years of formal education required to understand the text. For instance, a fog index of 12 suggests that a person needs a 12th-grade education to comprehend the document. The fog index is calculated using the average number of words per sentence and the number of complex words (those with more than three syllables).

For example, consider 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 for this sentence is calculated as follows:

Fog Index=0.4(Total WordsTotal Sentences+Complex WordsTotal Words×100)\text{Fog Index} = 0.4 \left(\frac{\text{Total Words}}{\text{Total Sentences}} + \frac{\text{Complex Words}}{\text{Total Words}} \times 100 \right)Fog Index=0.4(Total SentencesTotal Words +Total WordsComplex Words×100)

Using the given example: Fog Index=0.4(231+423×100)=23\text{Fog Index} = 0.4 \left(\frac{23}{1} + \frac{4}{23} \times 100 \right) = 23Fog Index=0.4(123+234×100)=23

If a user manual is intended for factory workers with an 8th-grade education, its Gunning Fog Index should not exceed 8 to ensure readability.

7.6 Summary

- In this chapter, we discussed the coding phases of the software development process.
- Most software development companies establish their own coding standards and require programmers to follow them. While these standards offer general guidelines for good

programming practices, individual engineers have the discretion to apply these principles in their work.

- Compared to testing, code review is a more effective method for reducing faults as it identifies problems, whereas testing identifies failures.

7.7 Keywords

- **Code Review:** After a module has been successfully compiled and syntax issues have been resolved, its code is reviewed. Code reviews are cost-effective methods to minimize coding errors and enhance code quality. Two main types of code reviews are code inspection and code walkthrough.
- **Code Inspection:** This process aims to identify common mistakes due to oversight or poor programming. Unlike code walkthroughs, which involve manually simulating code execution, code inspections focus on detecting specific types of errors.
- **Code Walkthrough:** An informal method of code analysis conducted after a module is coded and compiled, and syntax errors are removed.
- **Coding Standards:** Guidelines for coding practices, such as naming conventions, code layout, error return conventions, etc.

7.8 Self-Assessment Questions

1. What is the difference between a coding standard and a coding guideline? Why is it important for a software development organization to define and implement appropriate coding standards and guidelines? List five significant coding standards and recommendations you would suggest.
2. Briefly explain the difference between code inspection and code walkthrough. Compare the advantages of code inspection vs. code walkthrough.
3. Determine the Fog Index for this question. What does the Fog Index signify? How can the Fog Index help create good software documentation?
4. Differentiate between a software product's external and internal documentation. Why are they important to the organization?
5. Identify the types of errors that can be found during a code walkthrough.

7.9 Case Study

Introduction: In software engineering, adhering to coding standards is essential for ensuring code quality, readability, maintainability, and effective team collaboration. This case study explores coding standards and provides recommendations to address related challenges.

Problem: A software development team is struggling with inconsistent coding practices and a lack of adherence to coding standards. Challenges include:

1. **Inconsistent code formatting:** Developers use different formatting conventions, leading to hard-to-read and maintain code with inconsistent indentation, spacing, and naming conventions.
2. **Lack of code documentation:** Inconsistent documentation makes it difficult to understand code functionality, purpose, and usage, hindering code reuse and collaboration.
3. **Inefficient code organization:** Poor modularization and organization result in monolithic code files or classes, making navigation and maintenance difficult and reducing productivity.

Recommendations:

1. **Define and Communicate Coding Standards:** Establish guidelines for code formatting, naming conventions, documentation, and organization. Ensure all team members understand and follow these standards.
2. **Automate Code Formatting:** Use automated tools like Prettier or ESLint to enforce consistent code formatting across the project.
3. **Provide Code Review and Feedback:** Foster a culture of code reviews where team members review each other's code for adherence to coding standards, using code review tools to provide feedback and suggestions for improvement.
4. **Create Code Documentation Templates:** Develop templates for documenting code, including function descriptions, parameter details, and usage examples, making it a standard practice during development.
5. **Implement Modular Code Organization:** Encourage modular programming techniques and design patterns to enhance code organization and maintainability, breaking down complex code into smaller, reusable modules with well-defined responsibilities.

6. **Conduct Coding Standard Training Sessions:** Offer training sessions or workshops to educate developers on the importance of coding standards and their effective implementation, covering topics such as code formatting, documentation, and organization.
7. **Establish Code Quality Metrics:** Define and track metrics such as code coverage, code complexity, and adherence to coding standards. Use code analysis tools to identify improvement areas and measure progress over time.
8. **Continuously Improve Coding Standards:** Regularly review and update coding standards based on industry best practices and team feedback, encouraging developers to suggest improvements.

By implementing these recommendations, the software development team can enhance code quality, readability, and maintainability, ensuring consistent coding practices. Adherence to coding standards fosters better collaboration, reduces errors, and improves overall development efficiency.

7.10 References

- I. Summerville: Software Engineering 10th 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 & Verification

Learning Objectives:

- To understand the testing lifecycle, from development testing through customer acceptance testing; have been taught methodologies that assist you select test cases that are aimed towards finding program flaws;
- To describe the concept of black box testing.
- To describe what boundary value analysis means.
- To be familiar with test-first development, in which tests are created before code and run automatically;
- To understand the key distinctions between component, system, and release testing; and be knowledgeable about the methods and procedures used in user testing.

Structure:

Introduction

Testing a program involves providing it with a set of test inputs (or test cases) and observing its behavior to determine if it functions as expected. If the program does not perform as planned, the conditions under which failure occurs are documented for further debugging and adjustments.

The primary objective of testing is to identify all defects in the software. However, even after extensive testing, it is nearly impossible to ensure that the software is completely error-free, particularly in real-world applications with vast input data domains. Testing every possible input configuration is impractical. Despite this limitation, testing is crucial as it reveals many weaknesses in the software, reducing system flaws and increasing customer confidence in the final product.

There are two main objectives of the testing process:

1. **To demonstrate that the software meets its requirements:** For custom software, each requirement in the requirements document should have at least one corresponding test case. For generic software products, all system features and their combinations intended for the final release should be tested.
2. **To identify instances of inappropriate, undesirable, or incorrect behavior:** These issues stem from programming errors. Defect testing aims to detect and eliminate adverse system behaviors, such as crashes, unintended interactions with other systems, incorrect calculations, and data corruption.

8.1 Verification and Validation

Verification: Are we building the product right?

Validation ensures that the fully developed system meets its requirements specification, while verification checks that each stage of software development outputs correctly align with the previous phase. Validation aims to ensure the final product is error-free, while verification focuses on containing issues within each development phase.

The verification and validation processes ensure that the software adheres to its specifications and delivers the expected functionality to the stakeholders. These processes start as soon as the requirements are defined and continue throughout the development lifecycle. Verification checks that the software meets its stated functional and non-functional requirements. Validation, on the other hand, ensures that the software meets the customer's expectations and requirements, which might extend beyond the specifications.

Validation is crucial because requirements specifications may not always accurately reflect the true desires or needs of the system's users and customers. The main goal of verification and

validation is to ensure 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, the users' expectations, and the current market environment.

8.2 Fault and Failure

Fault

A fault arises from a mistake made by a developer during any stage of the development process. Various types of errors can occur in a program, such as calling the incorrect function. In the context of program testing, terms like error, fault, bug, and defect are often used interchangeably. However, it's important to note that in hardware testing, the term "fault" has a slightly different meaning compared to "error" and "bug"【IEEE90】.

Failure

A program failure indicates incorrect behavior exhibited by the program during its execution. This erroneous behavior can manifest as either an incorrect output or an inappropriate action performed by the program. Each failure results from a defect in the program. Essentially, every software failure can be traced back to some defect in the code. The number of possible failure modes for a program is vast. Here are three randomly selected instances of how an application can fail:

1. **Producing an Incorrect Output**: The program generates a result that does not match the expected output.
2. **Crashing**: The program terminates unexpectedly due to an unhandled exception or critical error.
3. **Unresponsive Behavior**: The program becomes unresponsive or freezes during execution.

These examples illustrate the broad spectrum of ways in which a program can fail, underscoring the importance of thorough testing to identify and rectify defects.

8.3 Debugging

Debugging After Detecting Failures

When a failure is detected, the next step is to identify the program statements responsible for the failure and then correct the errors. Below are some essential methods for locating errors, each with its own advantages and disadvantages. Here are some tips for effective debugging:

Debugging Approaches

Programmers commonly use the following approaches for debugging:

Brute Force Method

This is the least efficient method of debugging. It involves inserting print statements throughout the program to display intermediate values, hoping that some printed values will help identify the incorrect statement. The process becomes more systematic with the use of a symbolic debugger (or source code debugger), allowing for easy verification of variable values and setting breakpoints and watchpoints to monitor these values. A variation of this approach is single-stepping with a symbolic debugger, where the developer computes the expected result mentally after each source instruction and checks if it is correctly computed by single-stepping through the program.

Backtracking

This is another commonly used strategy. Starting from the statement where an error symptom is observed, the source code is traced backwards until the error is found. However, as the number of source lines to be traced back increases, so does the number of alternative backward paths, which can become unmanageably large for complex programs, limiting the applicability of this approach.

Cause Elimination Method

Once an error has been identified, the symptoms of the failure (e.g., a variable has an incorrect value) are logged. Based on these failure symptoms, potential causes are determined and tested to eliminate them. A related method is fault tree analysis, which is used to identify the error based on the error symptom.

Program Slicing

This method is similar to backtracking but reduces the search space by creating slices. A program slice for a specific variable and at a specific statement includes the source lines before this statement that potentially influence the value of that variable【Mund2002】. The idea is that an error in the value of a variable might be caused by the statements on which it is data-dependent.

8.4 Types of Testing

A commercial software system typically undergoes three levels of testing:

Development Testing

This involves testing the system to find flaws and defects. System designers and programmers usually conduct this testing.

Release Testing

An independent testing team tests the entire system before it is released to users. The goal is to ensure that the system meets the requirements of stakeholders.

User Testing

Users or potential users test the system in their own environments. For software, the 'user' might be an internal marketing team that decides if the software can be marketed, released, and sold. Acceptance testing is a type of user testing where the customer formally tests a system to determine whether it should be accepted by the system supplier or if additional development is required.

Development Testing

Testing during development can be done at three levels of detail:

Unit Testing

This involves testing individual program units or object classes. The functionality of objects or methods should be tested during unit testing. Unit testing begins after a module has been coded and reviewed.

Component Testing

Multiple units are combined to form composite components, and these are tested. Component testing focuses on component interfaces.

System Testing

Some or all components of a system are integrated, and the entire system is tested. System testing focuses on component interactions. This overlaps with component testing but differs in two key ways:

- Reusable components and off-the-shelf systems may be integrated with newly developed components, and the whole system is tested.
- Components developed by different team members or groups may be combined. System testing is a collaborative effort, often performed by a separate testing team without input from designers or programmers.

Unit Testing:

After a module has been coded and thoroughly reviewed, unit testing begins. Unit testing, also known as module testing, involves testing individual units or modules in isolation. To effectively test a single module, a complete environment that includes all necessary components for the module's operation is required. This environment should include:

- The processes from other modules that the module under test calls.
- Nonlocal data structures that the module accesses.
- A procedure for invoking the functions of the module under test with the appropriate arguments.

Modules necessary to create this environment, which either call or are called by the module under test, are generally not available until they have been unit tested themselves. To address this, stubs and drivers are used to simulate the required environment for a module.

Stubs and drivers play a crucial role in unit testing. A stub is a simplified version of a procedure that has the same input and output parameters as the actual procedure but with much simpler behavior. For instance, a stub procedure might produce the expected behavior using a basic table lookup method.

A driver module contains the nonlocal data structures accessed by the module under test, as well as the code to call the module's various functions with the required parameter values.

Integration Testing:

After a module has been coded and properly reviewed, unit testing commences. Unit testing, also referred to as module testing, involves testing individual units or modules in isolation. To conduct effective testing of a single module, it is essential to have a complete environment that

includes all necessary components for the module's operation. This environment should encompass:

- The processes invoked from other modules that the module under test interacts with.
- Nonlocal data structures accessed by the module.
- A mechanism to invoke the functions of the module under test with appropriate arguments.

Typically, modules required to provide this environment, either as callers or callees of the module under test, are not available until they themselves have undergone unit testing. To circumvent this dependency, stubs and drivers are used. A stub is a simplified version of a procedure that mimics the input/output parameters of the actual procedure but with simpler behavior. For instance, a stub might simulate expected behavior using a straightforward table lookup method. This setup allows testing to proceed without waiting for the completion of all dependent modules.

Stubs and drivers play a critical role in unit testing by facilitating the isolation and testing of individual modules before integration into the larger system.

1. Acceptance:

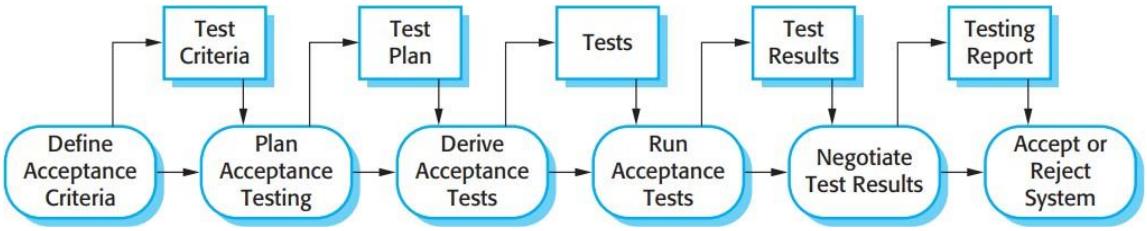

Figure 8.1: Acceptance Testing

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

Set acceptance standards: Before finalizing the system contract early in the process, it's essential to establish acceptance criteria agreed upon by both the customer and developer. However, in practice, defining criteria at this stage can be challenging due to evolving requirements throughout the project.

Acceptance testing plans: This phase determines the resources, time, and budget allocated for acceptance testing, along with a detailed testing schedule. The acceptance test plan should specify the coverage of requirements and the sequence for evaluating system features. Additionally, it should address potential risks to the testing process, such as system failures or performance issues, and propose mitigation strategies.

Construct acceptance tests: Once acceptance requirements are defined, tests are developed to verify whether the system meets these criteria. Acceptance tests should comprehensively evaluate both functional and non-functional aspects of the system. However, creating completely objective acceptance criteria can be challenging, as determining whether a test unequivocally demonstrates criterion fulfillment often involves discussion.

Conduct acceptance testing: The agreed-upon acceptance tests are executed on the system. Ideally, these tests should be conducted in the environment where the technology will be deployed, but practical constraints may necessitate setting up a suitable test environment. Automation of acceptance testing can be difficult as it may also assess how end-users interact with the technology, requiring user involvement and possibly training.

Discuss test results: It's unlikely that the system will pass all acceptance tests flawlessly. In cases where issues are identified, agreement between the software engineer and client is necessary to determine if the system is suitable for deployment. Plans for addressing identified issues should also be agreed upon during this phase.

Accept/Reject system: This phase involves a meeting between the customer and developers to decide whether the system should be approved for deployment. If the system is deemed unsuitable, further development is required to address identified issues. Once addressed, the acceptance testing phase is repeated.

8.5 Summary:

This chapter has focused on the Testing phase of the SDLC. Proper testing is crucial for identifying flaws in a system. Effective test cases are necessary to detect a wide range of errors without redundancy. Two primary testing approaches discussed are black-box testing, which focuses on functionality without knowing internal architecture, and white-box testing, which requires knowledge of the software's internal workings. Integration and system testing challenges were highlighted, emphasizing the importance of using the Software Requirements Specification

(SRS) as a basis for creating the system test suite. Functional and performance testing were identified as key categories within system testing, each targeting specific aspects of system requirements.

8.6 Keywords:

- **Testing:** Essential for identifying software flaws.
- **Development testing:** Conducted by the development team.
- **Unit testing:** Evaluates individual components.
- **Acceptance testing:** Determines if software is suitable for deployment.
- **Regression testing:** Ensures no new bugs introduced during updates.

8.7 Self-Assessment Questions

1. What do you mean when you say "integration testing"? What kinds of integration testing techniques can be applied to the integration testing of a sizable software product?
2. What does "performance testing" mean to you? Make a list of the many performancetesting types.
3. Differentiate between acceptance, alpha, and beta testing. How are the test cases for these tests created? Are the test cases for the three different sorts of testing always the same? Describe your response.
4. Which sort of testing—unit, integration, or system testing—tests the usability of a software product? Exactly how is usability tested?
5. Assume that a piece of software that has been produced has completed unit testing, integration testing, and system testing. Is it possible to state that the software is error- free? Justify your response.

8.8 Case Study

Introduction

Company XYZ, a prominent software development firm, is currently engaged in developing a large-scale e-commerce application. However, their testing phase in the SDLC has encountered

significant challenges impacting both quality and efficiency. Let's delve into these issues and propose solutions to address them effectively.

Problem:

1. **Inadequate test coverage:** Company XYZ primarily focused on functional testing while neglecting critical non-functional aspects such as performance, security, and usability testing. This oversight led to potential risks and performance bottlenecks that were not thoroughly validated.
2. **Lack of test environment management:** The company faced difficulties in setting up and managing dedicated test environments that mirrored the production environment accurately. Inconsistencies between test and production environments resulted in issues that surfaced only in the live environment.
3. **Limited test data management:** Company XYZ struggled with creating and managing diverse test data sets that encompassed real-world scenarios. The lack of varied test data hindered the detection of edge cases and potential data-related issues during testing.

Recommendations:

Comprehensive test coverage:

- **Prioritize critical non-functional requirements:** Identify and prioritize aspects like performance, security, and usability. Allocate adequate time and resources to validate these requirements thoroughly.
- **Implement risk-based testing:** Use a risk-based approach to identify high-risk areas and focus testing efforts accordingly.
- **Utilize testing techniques:** Employ techniques such as boundary value analysis, equivalence partitioning, and exploratory testing to enhance test coverage and identify edge cases effectively.

Effective test environment management:

- **Establish dedicated test environments:** Create dedicated test environments that closely replicate the production environment in terms of hardware, software, and configurations.
- **Use configuration management tools:** Implement tools for configuration management to ensure consistency across different test environments.

- **Automate environment setup:** Automate the setup and deployment of test environments to reduce manual effort and improve reproducibility.

Robust test data management:

- **Create diverse test data sets:** Develop diverse and realistic test data sets that cover a wide range of scenarios and edge cases.
- **Ensure data privacy:** Use data generation tools and anonymization techniques to protect data privacy and security during testing.
- **Implement data masking and subsetting:** Employ techniques like data masking and subsetting to efficiently manage test data while maintaining relevance and consistency.

Automation and tooling:

- **Leverage test automation:** Utilize test automation frameworks and tools to increase test coverage, improve efficiency, and reduce manual effort.
- **Adopt continuous integration:** Implement continuous integration and continuous testing practices to automate test execution and reporting.
- **Use performance testing tools:** Adopt performance testing tools to simulate real-world user loads and identify performance bottlenecks early in the development lifecycle.

Collaboration and communication:

- **Promote collaboration:** Foster collaboration among developers, testers, and stakeholders throughout the testing phase.
- **Establish clear communication:** Set up clear communication channels for defect reporting and tracking to ensure timely resolution and minimize rework.
- **Conduct regular test status meetings:** Hold regular meetings to provide updates, address concerns, and align testing efforts with project milestones effectively.

By implementing these recommendations, Company XYZ can significantly enhance the effectiveness and efficiency of its testing phase. Comprehensive test coverage, robust test environment management, effective test data management, automation, and improved collaboration will collectively contribute to higher software quality, reduced risks, and increased customer satisfaction.

8.9 References

- I. Summerville: Software Engineering 10th 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 Reverse Engineering

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 upkeep.
- 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 increasingly prioritizing software maintenance as a critical task. This trend arises from the rapid pace of hardware obsolescence, the extended lifespan of software products, and the user community's demand for software to operate on new platforms, in new contexts, or with enhanced capabilities. Software maintenance becomes necessary when a software product performs low-level tasks tied to changing hardware platforms or when it needs updates to support new interfaces brought by changes in the operating environment, such as new operating system versions.

9.1 Characteristics of Software Maintenance

In this section, we categorize various maintenance efforts into distinct classes and discuss general characteristics and specific challenges associated with maintenance initiatives. Software products inherently continue to evolve beyond their initial development through maintenance activities.

Types of Software Maintenance

Software maintenance can be broadly classified into three categories:

- **Corrective Maintenance:** This type addresses fixing flaws discovered during the software product's operational use.
- **Adaptive Maintenance:** Required when clients need the software product to function on new platforms, integrate with new hardware or software, or adapt to new operating system environments.
- **Perfective Maintenance:** Involves updating software functionalities, enhancing existing features, and incorporating new capabilities as requested by users.

By understanding and implementing these types of maintenance, software organizations can ensure that their products remain relevant, functional, and compatible with evolving technological landscapes.

9.2 Software Reverse Engineering

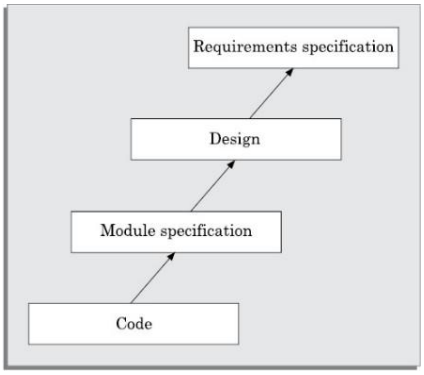

Figure 9.1: Process Model for Reverse Engineering

Software reverse engineering involves the process of extracting the design and requirements specification of a product by examining its code. The primary goals of reverse engineering are to generate necessary documentation for legacy systems and to facilitate easier maintenance by

enhancing system comprehensibility. This practice has become increasingly vital as older software products often lack structure and adequate documentation, eventually becoming legacy software through repeated maintenance cycles.

In the initial stages of reverse engineering, the focus typically revolves around aesthetic improvements to the code without altering its functionality. This includes enhancing readability, structure, and overall understanding of the codebase. Various tools, such as pretty-printer programs, can reformat the code to improve its layout and presentation. Many legacy software systems suffer from complex control structures and unclear variable names, which hinder understanding.

A critical aspect of improving code readability is using descriptive variable names throughout the codebase. Meaningful names for variables, data structures, and functions significantly aid in code documentation. Complex nested conditional statements in the code can be simplified by using straightforward conditional or case statements where appropriate.

By employing these practices, software developers can effectively reverse engineer legacy systems, improve code clarity, and streamline maintenance processes without altering the software's intended functionality.

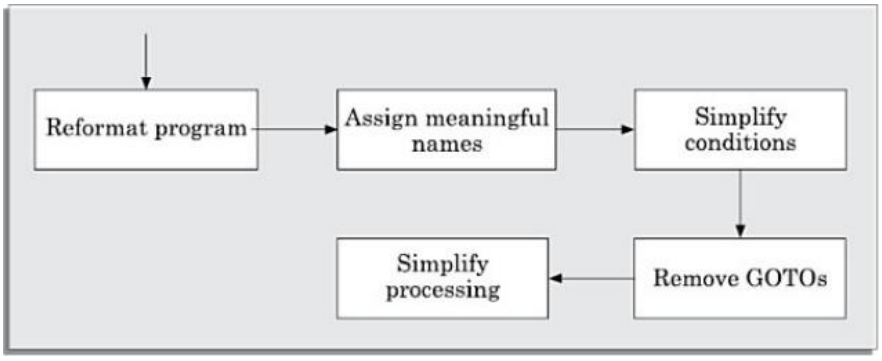

Figure 9.2: Cosmetic Changes before reverse engineering

After making cosmetic improvements to legacy software, the next step involves the process of extracting its code, design, and requirements specification. These activities are illustrated in Figure [insert figure reference]. To effectively extract the design, a comprehensive understanding of the code is essential. Technologies exist that can automatically generate data flow and control flow diagrams, aiding in this comprehension process. Additionally, extracting the structure chart, which outlines module invocation sequences and data transfers between modules, is crucial.

Once the entire codebase has been thoroughly understood and the design components extracted, the Software Requirements Specification (SRS) document can be prepared. This document serves as a detailed description of what the software system should do and how it should perform, based on the insights gained from reverse engineering.

By systematically following these steps, software teams can enhance their understanding of legacy systems, document their designs effectively, and prepare comprehensive SRS documents to guide further development or maintenance efforts.

9.3 Maintenance Process Model

There are two primary software maintenance process models that can be implemented. The first model is suitable for small rework projects where code updates are made directly and then reflected in the necessary documentation. The maintenance process is depicted graphically in Figure [insert figure reference].

In this model, the maintenance project begins with gathering the requirements for changes. These requirements are then analyzed to devise strategies for modifying the code. It is beneficial to involve at least some members of the original development team during this stage, particularly for projects dealing with unstructured and poorly documented code. This involvement helps mitigate risks and improves the understanding of the system's intricacies.

Having access to the original working system at the maintenance site significantly simplifies the workload for maintenance engineers. It allows them to gain a clear understanding of how the original system functions and facilitates comparison with their modified system. This access also streamlines the debugging process for the re-engineered system.

By adopting this approach, software maintenance teams can effectively manage rework projects, ensure clarity in code updates, and maintain documentation accuracy throughout the maintenance process.

Re-engineered system is also simplified because the program traces of both systems may be compared to locate problems.

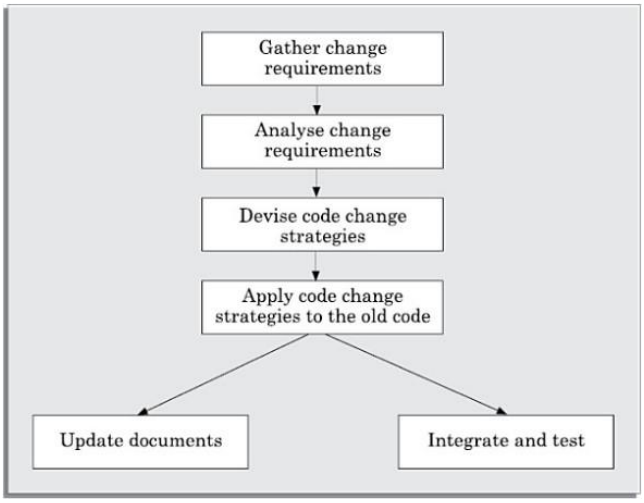

Figure 9.3: Maintenance Process Model -1

The second process model for software maintenance is appropriate for projects that require a considerable quantity of rework. The process involves a reverse engineering cycle followed by a forward engineering cycle, which collectively constitutes software reengineering. This method, illustrated in the figure, is particularly crucial for legacy products.

During the reverse engineering cycle, the old code undergoes analysis or abstraction to extract module specifications. These specifications are then further analyzed to formulate the design of the software system.

Subsequently, in the forward engineering cycle, the designed system is implemented or reconstructed based on the extracted specifications and design. This phase essentially translates the abstracted design into actual executable code or software components.

Overall, the process of software reengineering aims to modernize or improve legacy systems by first understanding their existing structure and functionality through reverse engineering, and then using this understanding to implement enhancements or updates through forward engineering. This approach helps in maintaining and evolving software systems that have been in operation over extended periods, often with outdated technologies or architectures.

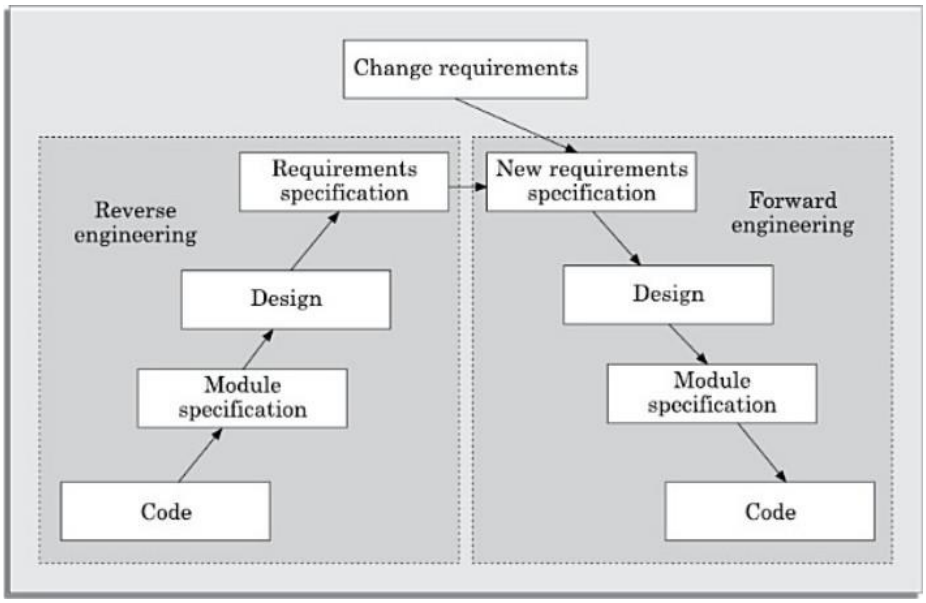

Figure 9.4: Maintenance Process model-2

The reverse-engineered items are heavily reused during the design, module specification, and coding stages. One significant advantage of this method is that it results in a more structured design than the original product, superior documentation, and, in many cases, better productivity. A more efficient design is responsible for the efficiency gains. However, this strategy is more expensive than the first.

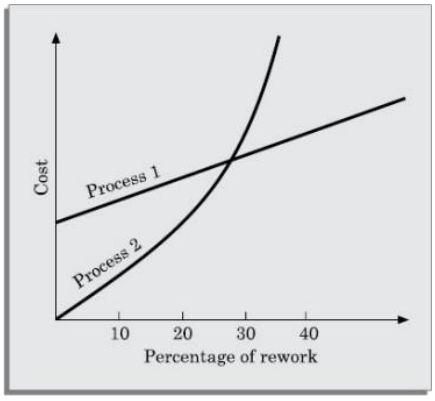

Figure 9.5: Empirical Estimation of Maintenance Cost and Rework

9.4 Self-Assessment Questions:

1. What are the characteristics of Software Maintenance?
2. What do you understand by Software Reverse Engineering?
3. Explain the various Maintenance Process Models.

9.5 Case Study

Introduction: Company ABC, a software development firm, has encountered significant challenges in managing software maintenance and controlling associated costs. This case study explores their issues, provides insightful answers to key questions, offers strategic recommendations, and concludes with essential takeaways.

Problem:

1. **Increasing maintenance costs:** Company ABC has faced a consistent rise in software maintenance expenses over time. Costs related to defect rectification, feature enhancements, and adapting to changes have become a substantial part of their budget, impacting overall profitability.
2. **Inefficient bug tracking and resolution:** The company struggles with tracking and resolving software defects efficiently. This inefficiency leads to extended resolution times, heightened customer dissatisfaction, and increased maintenance expenditures.
3. **Lack of proactive maintenance:** Company ABC relies heavily on reactive maintenance practices, addressing issues as they arise. This reactive approach has resulted in a backlog of unresolved issues and insufficient focus on preventive maintenance, causing recurring problems and additional costs.

Recommendations:

1. **Implement effective bug tracking and resolution processes:**
   - Utilize robust bug-tracking tools to systematically capture and prioritize software defects. Define clear responsibilities and establish realistic timelines for bug resolution.
   - Develop a structured process for bug triage, categorization, and resolution. Regularly communicate progress and updates to stakeholders.
   - Conduct root cause analysis to identify underlying issues contributing to defects and implement preventive measures proactively.
2. **Proactive Maintenance and Continuous Improvement:**

- Introduce a preventive maintenance strategy involving regular code reviews, refactoring, and performance optimization.
- Continuously assess software functionality and performance to preemptively identify potential issues and implement corrective measures.
- Solicit and prioritize user feedback to identify usability concerns and drive iterative improvements.

3. **Embrace automated testing and monitoring:**
   - Deploy advanced automated testing frameworks to detect and prevent regression issues effectively.
   - Adopt continuous integration and deployment practices to automate testing processes, facilitating rapid defect identification and resolution.
   - Monitor software performance metrics and usage patterns to pinpoint areas for enhancement, thereby optimizing maintenance efforts.
4. **Prioritize and manage change requests effectively:**
   - Establish an efficient change management process to handle requests systematically and expediently.
   - Evaluate change requests based on their impact, anticipated benefits, and alignment with organizational goals prior to approval.
   - Communicate the implications of change requests on maintenance costs and allocate resources judiciously for optimal outcomes.

Conclusion:

Software maintenance is pivotal in the software development lifecycle. By implementing proactive maintenance strategies, efficient bug tracking and resolution mechanisms, and leveraging automated testing and monitoring tools, companies can mitigate maintenance costs, enhance software quality, and improve customer satisfaction. Company ABC stands to benefit significantly from these recommendations, fostering a proactive approach to maintenance, minimizing recurring issues, and promoting continuous improvement. By prioritizing preventive maintenance and addressing issues promptly, organizations can optimize maintenance expenditures, ensuring sustained success and longevity for their software applications.

9.6 References:

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- Summerville: Software Engineering 10th Edition, Pearson Education, 2017 23
- 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 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

Any software development project's success depends on proper project management. In the past, lots of projects came up short due to poor project management techniques instead of a lack of skilled engineers or resources. Therefore, it's important to carefully study the most current techniques of software project management.

The field of software project management is fairly broad. Effective methods for managing software projects can be studied for a whole semester. But in this chapter, we'll limit ourselves to only a few fundamental difficulties. Let's first clarify the primary objective of software project management.

The basic objective of the management of software projects is to make it easy for a team of engineers to work together properly on a project.

Based on the definition given above, project management requires employing a variety of strategies and skills to guide a project toward success. Let's identify who should be in control of managing projects before focusing on these project management techniques. The project manager is typically a seasoned team member who serves as the group's administrative supervisor. A single team member takes on the duties of both project leadership and technical direction for small software creation projects. Large projects should be handled by a different team member than the project manager.

A single team member takes on the duties of both project management and technical management for small software development projects. A different team member (other than the project manager) takes on the role of technical leadership for major projects. The technical leader is responsible for making decisions regarding the project's tools and approaches, high-level problem solutions, appropriate algorithms, etc.

We first look into why managing projects involving software is so much more difficult than managing many other sorts of projects in this chapter.

The primary duties and pursuits of a software project manager are then described. The project planning activity is then briefly described. After that, we talk about scheduling and estimating methods. At last, we describe the risk and configuration management processes in general.

10.1 Software Project Management

Software project management is an essential discipline within software engineering. Due to constraints like organizational budgets and schedules, professional software engineering demands effective management. The primary role of a project manager is to ensure that software projects navigate these constraints while delivering high-quality products. While good management doesn't guarantee project success, poor project management often leads to late deliveries, exceeding budgets, or falling short of client expectations.

10.2 Roles and Responsibilities of Project Manager

Project managers in software development oversee the overall project management and its success. Their responsibilities range widely, from mundane tasks like team morale boosting to

critical ones such as client presentations. Key responsibilities include writing project proposals, estimating costs, scheduling, staffing, customizing software processes, monitoring project progress, managing risks, interacting with clients, and preparing managerial reports. These tasks can broadly be categorized into two main areas: project planning and project monitoring/control. Project planning organizes tasks before development, while monitoring/control activities ensure adherence to plans and adapt them as necessary during development.

Skills of Project Manager

Effective software project managers require a theoretical understanding of various project management methodologies. Moreover, they need strong qualitative judgment, decision-making skills, and effective communication abilities. Competencies in areas like cost estimation, risk management, and configuration management are crucial. Practical skills in tracking project progress, engaging clients, delivering managerial presentations, and fostering team development are typically gained through experience.

Managing software projects poses unique challenges compared to other types of projects due to their technical complexity, evolving requirements, and the need for continuous adaptation. Effective project management ensures that software projects stay on track, meet objectives, and deliver value to stakeholders.

Price: What will the software product's development cost?

Timeframe: How much time will it take to develop the product?

Effort: How much work would be involved in creating the product?

The accuracy of these three estimates significantly influences the success of subsequent planning tasks, such as scheduling and staffing.

Scheduling: Once all project parameters are estimated, schedules for manpower and other resources are developed.

Staffing: Plans for staff organization and staffing are then formulated.

Risk Management: This involves identifying risks, analyzing them, and planning measures to mitigate them.

Miscellaneous Plans: This includes developing various other plans like quality assurance and configuration management plans.

Figure (10.1) illustrates the sequence of these planning activities. Notably, size estimation is the initial activity undertaken by a project manager during project planning.

Size estimation is crucial as it forms the foundation for all subsequent predictions and project plans. Based on size estimation:

- Effort required to complete the project and the project's timeline are estimated.
- Project cost is determined based on the estimated effort, guiding negotiations with the customer regarding pricing.
- Further planning tasks, such as personnel allocation and scheduling, are conducted based on the estimated effort and timeline.

Size estimation, therefore, serves as a cornerstone in project management, influencing decision-making and planning throughout the project lifecycle.

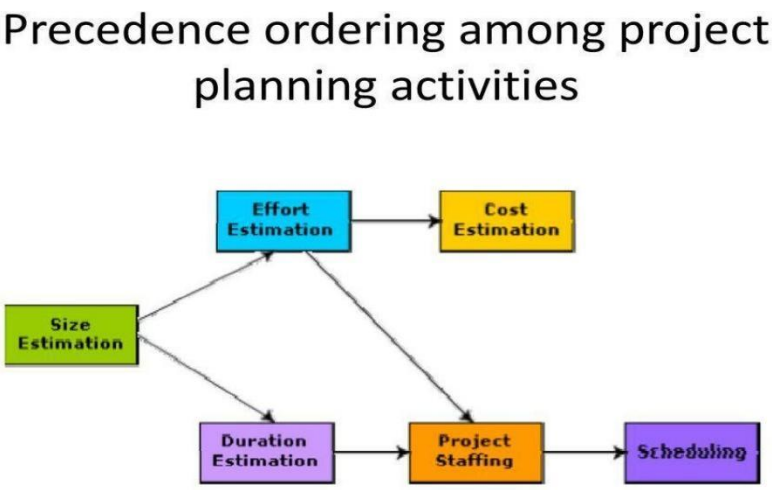

Figure 10.1: Project Planning Activities

10.3 Project Planning

Precise planning at the outset of a significant project is often exceptionally challenging, especially considering that such projects can span several years. Consequently, project parameters, scope, and staffing frequently undergo substantial changes throughout the project

lifecycle, leading to deviations from initial plans. To mitigate this challenge, project managers adopt a staggered planning approach known as "sliding," which allows for iterative refinement of project plans as more information becomes available and project parameters evolve.

Initially, project managers face incomplete knowledge about project specifics, gradually enhancing their understanding as the project progresses through different development phases. This progression reveals complexities in project activities, resolves some anticipated risks, and identifies new risks. Periodic re-estimation of project parameters ensures that subsequent activities are planned more accurately, progressively enhancing confidence levels.

The SPMP Document of Project Planning Once project planning concludes, project managers compile their strategies into a Software Project Management Plan (SPMP) document. This comprehensive document covers various critical topics:

1. **Introduction:**
    - Objectives
    - Major Functions
    - Performance Issues
    - Management and Technical Constraints
2. **Project Estimates:**
    - Historical Data Used
    - Estimation Techniques Employed
    - Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule:**
    - Work Breakdown Structure
    - Task Network Representation
    - Gantt Chart Representation
    - PERT Chart Representation
4. **Project Resources:**
    - People
    - Hardware and Software
    - Special Resources
5. **Staff Organization:**
    - Team Structure

- Management Reporting

6. **Risk Management Plan:**
   - Risk Analysis
   - Risk Identification
   - Risk Estimation
   - Risk Abatement Procedures
7. **Project Tracking and Control Plan:**
   - Metrics to be Tracked
   - Tracking Plan
   - Control Plan
8. **Miscellaneous Plans:**
   - Process Tailoring
   - Quality Assurance Plan
   - Configuration Management Plan
   - Validation and Verification
   - System Testing Plan
   - Delivery, Installation, and Maintenance Plan
   -

Metrics for Project Size Estimation Accurate estimation of project size is pivotal for determining effort, completion time, and overall project cost. "Project size" refers not merely to the bytes of source code or executable code, but rather to the complexity of the problem and the effort required to develop a solution. Two primary metrics used for size estimation are Function Points (FP) and Lines of Code (LOC).

Lines of Code (LOC) LOC is a straightforward metric for gauging project size based on the count of source instructions in the final program, excluding comments and headers. However, estimating LOC at the project's outset can be challenging and relies on systematic estimation methods. Despite its simplicity, LOC has several limitations:

- **Focus on Coding Activity:** LOC only measures coding effort and does not account for the effort involved in other lifecycle tasks like specification, design, and testing.
- **Dependence on Coding Style:** LOC count can vary based on individual coding styles, potentially leading to different size measurements for functionally similar programs.

- **Poor Correlation with Code Quality:** Larger LOC does not necessarily indicate higher efficiency or better quality code.
- **Impact on Code Reuse:** LOC metric may discourage code reuse, penalizing developers who utilize libraries or higher-level programming constructs.
- **Neglect of Logical and Structural Complexities:** LOC does not capture complexities in logic and structure, crucial factors influencing development effort.
- **Difficulty in Early Estimation:** Predicting final LOC from initial project specifications is challenging, limiting its utility in early project planning stages.

In conclusion, while LOC provides a numerical measure of project size, its limitations necessitate careful consideration when using it for project estimation and management, particularly in modern software development environments characterized by complexity and rapid change.

Function Point

Albrecht (1983) introduced the function point metric as an alternative to the lines of code (LOC) metric, addressing several shortcomings of LOC. Function points measure the size of a software product based on its functional requirements, providing a more abstract and consistent measure compared to LOC, which can vary significantly based on coding style and language used. Unlike LOC, function points can be estimated early in the project lifecycle, solely based on the problem definition, without needing to wait until the software is fully built.

10.4 Project Estimation Techniques

Estimating project characteristics such as size, effort, time, and cost is crucial for effective project planning. These estimates not only determine project costs for clients but also aid in resource allocation and scheduling. Estimation techniques fall into three main categories:

1. Empirical Techniques: Empirical estimation relies on past experience and expert judgement to estimate project parameters. Expert judgement involves experienced individuals providing educated approximations of project size and effort based on their analysis of similar projects. However, it is prone to biases and may overlook certain project aspects. Group estimation, such

as Delphi cost estimation, mitigates individual biases by aggregating estimates from a group of experts anonymously and iteratively.

2. Heuristic Techniques: Heuristic approaches use mathematical models to describe relationships among project parameters. Single-variable models, like the basic COCOMO model, estimate project features based on a single independent variable, such as project size. Multivariable models, such as intermediate COCOMO, consider multiple independent variables to provide more accurate estimates.

3. Analytical Estimation Techniques: Analytical estimation techniques, like Halstead's software science, start with fundamental assumptions about the project and derive estimates based on these assumptions. Unlike empirical and heuristic methods, analytical techniques are grounded in scientific principles and can provide insights into complex aspects like software maintenance efforts.

10.5 Scheduling, Staffing Estimation, Organization, and Team Structure

1. Scheduling: Scheduling involves identifying project tasks, estimating their duration, determining task dependencies, and creating timelines using tools like Gantt charts and PERT charts. Project managers use scheduling to allocate resources and monitor project progress against timelines, making adjustments as necessary to prevent schedule slippage.

2. Staffing Level Estimation: Staffing estimation determines the human resources required for software development based on identified work requirements. Models like the Rayleigh curve, proposed by Norden and adapted by Putnam for software projects, help estimate staffing needs based on project size and duration. Putnam's model relates lines of code delivered to effort and time required, considering factors like technology constraints and development environment quality (Ck).

3. Organization and Team Structure: Software development organizations can adopt functional, project-based, or matrix-based structures. Each structure has distinct advantages: functional structures group employees by their expertise (e.g., programming, testing), project-

based structures organize teams around specific projects, and matrix structures combine functional expertise with project responsibilities to optimize resource allocation and skill utilization.

In summary, while metrics like function points offer advantages over LOC in terms of early estimation and consistency, project estimation techniques and organizational strategies play pivotal roles in planning and executing successful software projects, mitigating risks and optimizing resource utilization. These approaches collectively contribute to effective project management and delivery in dynamic software development environments.

10.6 Functional Format vs Project Format

A software development organization can be structured in two primary ways: functionally and project-based (or project-oriented). Each structure offers distinct advantages and is chosen based on the nature of the projects and organizational goals.

1. Functional Structure: In a functional organization structure, development personnel are grouped based on their functional expertise, such as programming, testing, design, etc. Each functional group operates independently, focusing on their specific area of expertise. Engineers within each functional group report to a functional manager who oversees their work and career development. This structure facilitates specialization and skill development within each functional area, as team members collaborate closely on similar tasks and challenges.

2. Project-Based Structure: In contrast, a project-based organization structure organizes teams around specific projects or products. Each project team consists of members from different functional groups who collaborate to achieve project objectives. Project teams are usually led by a project manager who coordinates activities, manages resources, and ensures the project meets its goals within scope, time, and budget constraints. This structure promotes flexibility and responsiveness as teams are assembled and disbanded based on project needs.

3. Integration of Structures: Some organizations adopt a hybrid or matrix structure, which combines elements of both functional and project-based approaches. In a matrix structure, employees maintain their functional roles while also working on project teams as needed. This

allows for a balance between specialization within functional areas and cross-functional collaboration required for project execution.

Advantages:

- **Functional Structure:** Promotes deep expertise and specialization within functional areas. Allows for career growth and skill development within specific domains.
- **Project-Based Structure:** Enhances collaboration across functional boundaries. Facilitates efficient project execution and responsiveness to project-specific needs.

Challenges:

- **Functional Structure:** May lead to silos and difficulty in communication across functional groups. Resource allocation across projects may be challenging.
- **Project-Based Structure:** Potential for conflicts between project managers and functional managers over resource priorities. Project teams may face challenges in accessing specialized expertise.

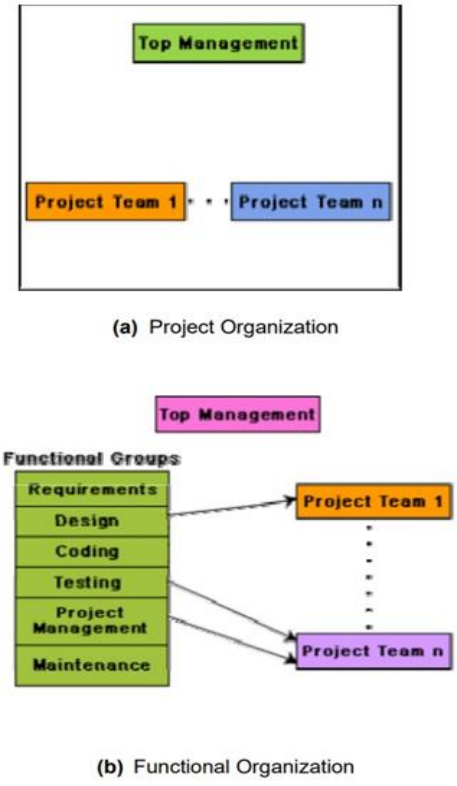

Figure 10.2: Functional and Project Organization

Different teams of programmers work on different phases of a project in a functional style. For instance, one team may handle the requirements specification.

Another person will do the design, and so on. As the project progresses, the partially completed product is passed from one team to the next. As a result, the functional structure necessitates extensive communication throughout the various teams because the work of one team must be fully understood by the successive teams working on the project. This necessitates the creation of high-quality documentation following each activity.

In the project format, a group of engineers are assigned to the project at the beginning and continue with it till the project has been finished. As a result, the same staff takes charge of every aspect of the life cycle. The functional model necessitates more team communication than the project format because one team must comprehend the work of prior 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 10.3: Matrix Organisation structure

A matrix organization aims to blend the strengths of both functional and project-based structures by integrating functional specialists into project teams as needed. This approach allows for greater flexibility in resource allocation and promotes cross-functional collaboration. However, the dynamics within a matrix organization can vary significantly based on the authority distribution between functional managers and project managers.

Types of Matrix Organizations: In a matrix organization, the balance of power between functional managers and project managers determines its categorization as strong or weak:

- **Strong Matrix:** Functional managers have significant authority and control over assigning their specialists to projects. Project managers must accept the resources allocated by functional managers. This setup prioritizes functional goals and expertise.
- **Weak Matrix:** Project managers retain more authority, including budget control and the ability to reject or request specific resources from functional groups. They may also have the flexibility to hire external resources if needed, focusing more on project-specific objectives.

Challenges in Matrix Organizations:

1. **Conflict Over Staffing:** Tensions can arise between functional managers, who prioritize their department's needs, and project managers, who require specific skills for their projects. Resolving these conflicts is crucial for maintaining project timelines and team morale.
2. **Resource Allocation Issues:** Specialists in a matrix organization often face shifting priorities as they are reassigned to different projects based on evolving needs. This can lead to inefficiencies and delays, especially when urgent issues arise in multiple projects simultaneously.

b. Team Structure:

In addition to the matrix structure, project teams within organizations can adopt various formal team structures:

- **Chief Programmer Team:** Led by a senior engineer or principal programmer, this structure provides clear technical leadership. Tasks are divided among team members, who report to the lead programmer for guidance and verification. This structure is effective for well-defined problems but may stifle creativity and morale due to centralized decision-making and supervision.

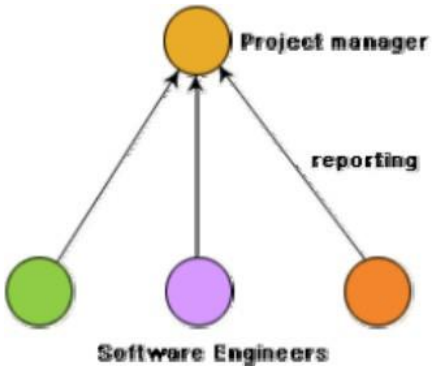

Figure 10.4: Chief Team Structure

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 single person has the necessary intellectual capacity to complete the assignment, 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.

- **Democratic Team:**

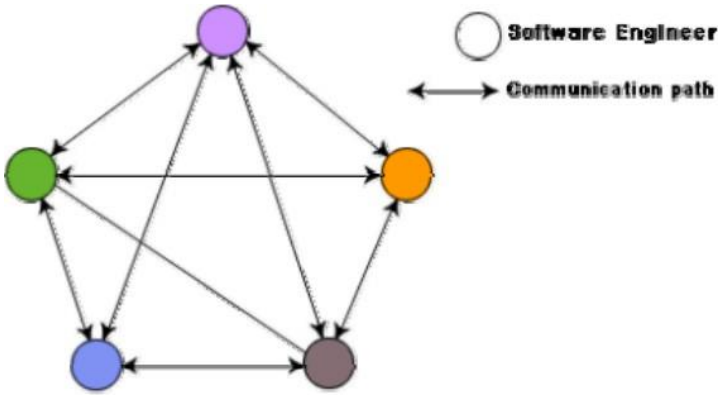

Figure 10.5: Democratic Team Structure

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 organisation is appropriate. Purely democratic organisations 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.

- **Mixed Control Team Organization:**

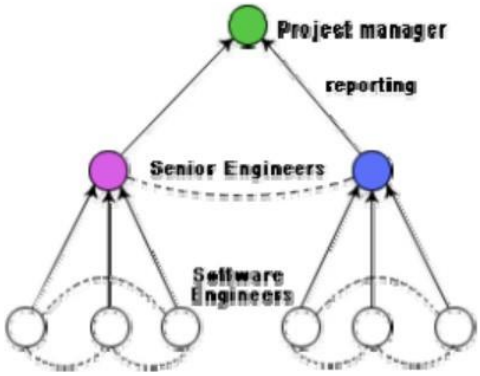

Figure 10.6: Mixed Team Structure

Introduction: Company XYZ, a software development firm, recently encountered significant challenges during a complex software development project. These issues primarily stemmed from unclear project goals and scope, inefficient resource allocation and utilization, and poor communication and collaboration among team members and stakeholders.

Problem Analysis:

1. **Unclear Project Goals and Scope:** Company XYZ struggled with defining and maintaining clear project goals and scope. This ambiguity led to frequent changes in

scope, known as scope creep, which in turn caused project delays and difficulties in managing priorities and expectations.

2. **Inefficient Resource Allocation and Utilization:** The company faced challenges in effectively allocating and utilizing resources such as human resources, time, and budget. Poor resource planning and tracking exacerbated delays, inefficiencies, and increased project costs.
3. **Poor Communication and Collaboration:** Communication gaps and ineffective collaboration among team members, stakeholders, and project managers hindered project progress. This resulted in misunderstandings, delays in decision-making, and a lack of shared understanding among project participants.

Recommendations:

1. **Clear Definition of Project Goals and Scope:**
   - Conduct a comprehensive requirements analysis to clearly identify and document project goals, objectives, and scope.
   - Collaborate closely with stakeholders to ensure mutual agreement on project deliverables, milestones, and acceptance criteria.
   - Implement a robust change management process to handle scope changes effectively, considering their impact on schedule, resources, and budget.
2. **Efficient Resource Allocation and Utilization:**
   - Conduct thorough resource planning to identify required skill sets, roles, and responsibilities for the project team.
   - Utilize project management tools or resource management software to monitor resource availability, utilization, and allocation.
   - Regularly assess resource workloads and adjust assignments to maintain balanced task distribution and prevent bottlenecks.
3. **Enhance Communication and Collaboration:**
   - Establish clear communication channels and protocols to facilitate timely and effective communication among all project stakeholders.
   - Conduct regular project status meetings to provide updates, discuss challenges, and align expectations among team members.

- Foster an open and transparent communication culture that encourages team members to voice concerns, share ideas, and provide feedback.

4. **Implementation of Project Management Methodologies:**
   - Adopt suitable project management methodologies, such as Agile or Waterfall, based on the project's nature and requirements.
   - Employ project management tools like Jira, Trello, or Microsoft Project to plan, track, and monitor project tasks, milestones, and progress effectively.
   - Break down the project into manageable work packages, set realistic deadlines, and regularly monitor and report on project progress.
5. **Regular Project Monitoring and Risk Management:**
   - Establish a robust project monitoring and reporting mechanism to track project progress, identify bottlenecks, and address risks proactively.
   - Conduct periodic project reviews to evaluate performance, identify areas for improvement, and adjust project plans as necessary.
   - Develop and implement a comprehensive risk management plan to identify, assess, and mitigate potential project risks throughout the project lifecycle.

10.7 Self-Assessment Questions:

1. What are the primary objectives of software project management, and why is it crucial for successful software development?
2. List at least three key responsibilities of a project manager in a software development project.
3. What are the essential steps involved in planning a software project?
4. Describe two common techniques used for estimating the cost and time required for a software project.

10.8 References

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- I. Summerville: Software Engineering 10th Edition, Pearson Education, 2017 23
- 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- 11

Software Reliability

Learning Objectives:

- Identify the primary reasons why software reliability is difficult to quantify.
- Identify the reliability measures that can be used to quantify software product reliability.
- Recognise the many types of software product failures.
- Describe a software product's reliability growth models. Determine the key quality characteristics of a software application.
- Identify the key characteristics of ISO 9001 certification.
- Explain the SEI CMM model's core process areas for a software organisation.

Structure:

Introduction

In many product categories, especially safety-critical and embedded software solutions, customers not only desire highly reliable products but also demand quantifiable assurances of their reliability before making purchasing decisions. However, as detailed in this chapter, accurately assessing the reliability of software products presents significant challenges. One of

the primary challenges is the subjectivity inherent in reliability assessments, where different user groups may arrive at different conclusions about a product's reliability. Additionally, fluctuating reliability values due to bug fixes further complicate precise measurement of software reliability. This chapter delves into these complexities and explores various methodologies used to quantify software product reliability.

11.1 Software Reliability

Software reliability refers to the dependability or trustworthiness of a software product over a specific period. A software product with numerous bugs is considered unreliable. Reducing the number of defects generally improves reliability, but there isn't always a direct correlation between observed system reliability and the number of latent defects within the system. For instance, eliminating bugs from less frequently used parts of the software may have minimal impact on overall reliability. This is because the reliability gain from fixing an error depends not just on the number of defects but also on where and how frequently those defects affect the software's execution.

11.2 Reliability Metrics

Different software products have varying reliability requirements, which should be clearly defined in the Software Requirements Specification (SRS) document. Several metrics can quantitatively assess software reliability:

- **ROCOF (Rate of Occurrence of Failure)**: Measures the frequency of failures occurring during system operation over a specified period.
- **MTTF (Mean Time To Failure)**: Calculates the average time between failures over a significant number of failure instances. It considers only the runtime and excludes periods like system downtime for error rectification.
- **MTTR (Mean Time To Repair)**: Represents the average time required to locate and fix errors that lead to failures.
- **MTBF (Mean Time Between Failures)**: Combines MTTF and MTTR to estimate the expected time between successive failures.
- **POFOD (Probability of Failure on Demand)**: Estimates the likelihood of a system failing when a service request is made.

- **Availability**: Measures the probability that a system will be operational and available for use during a specific period, factoring in both failure occurrences and repair times.

11.3 Reliability Growth Modelling

Reliability growth models are mathematical representations that illustrate how software reliability improves as defects are identified and corrected during the development and testing phases. These models help predict when a desired level of reliability will be achieved, aiding in decision-making regarding when to conclude testing to achieve a specified reliability threshold.

The simplest reliability growth model is a step function model in which reliability is considered to rise by a constant increment each time an issue is found and rectified. The figure depicts such a model. However, because it is well known that different types of errors contribute differently to reliability growth, this simplistic model of reliability, which implicitly assumes that all errors contribute equally to reliability growth, is highly impractical.

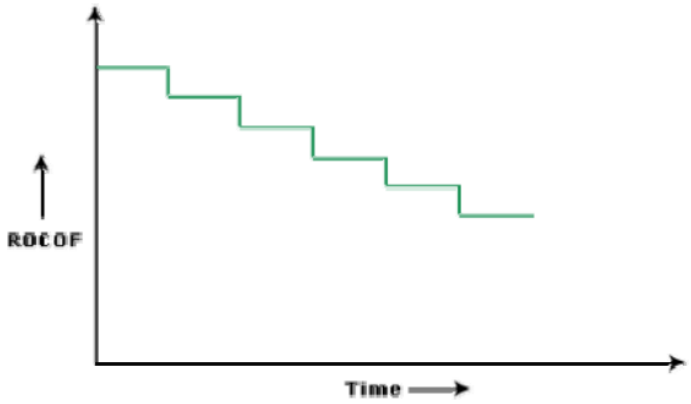

Figure:11.1 Setup Function Model of reliability growth

The Model of Littlewood and Verrall

This model allows for negative reliability growth to reflect the reality that when a repair is performed, extra defects may be introduced. It also accounts for the fact that, when errors are addressed, the average improvement in dependability per repair falls (see Fig. It considers an error's contribution to improved reliability to be an independent random variable with a Gamma distribution. This distribution represents the fact that error fixes that contribute significantly to

reliability growth are removed first. As the test progresses, the rate of return decreases.

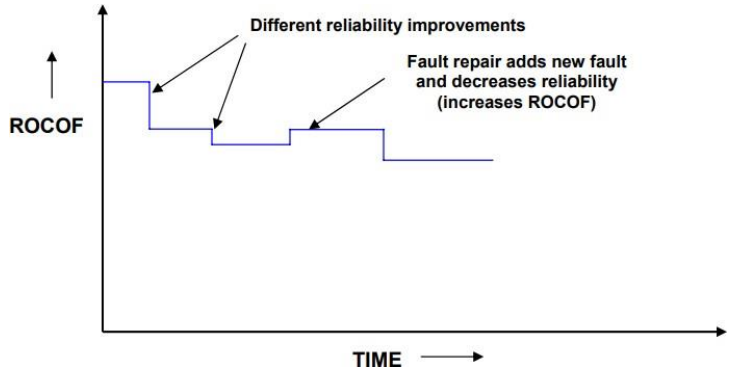

Figure 11.2: Random Step Function Model of Reliability Growth.

11.4 Software Quality

High-quality products are typically defined by their ability to fulfill their intended purpose effectively. For software products, this concept, known as "fitness for purpose," is traditionally defined by meeting the requirements outlined in the Software Requirements Specification (SRS) document. However, this definition alone does not adequately capture all aspects of software quality. For instance, a software product might function correctly according to its specifications but have a user interface that is difficult to use or maintain. Likewise, a product may meet all user expectations but have poorly written and unintelligible code, impacting its overall quality.

Modern definitions of software quality encompass a broader range of factors beyond "fitness for purpose." These include:

- **Portability**: The ability of a software product to operate effectively across different operating systems and hardware platforms.
- **Usability**: How easily and effectively users can interact with the software, catering to both novice and expert users.
- **Reusability**: The extent to which different parts of the software can be reused in other projects or scenarios.
- **Correctness**: Ensuring that the software product meets all specified requirements without errors.
- **Maintainability**: How easily defects can be corrected, new features added, or adjustments made to the software over its lifecycle.

11.5 ISO 9001

ISO 9001 is a set of standards under the ISO 9000 series, which establishes criteria for a quality management system (QMS). It is applicable to organizations involved in designing, developing, producing, and servicing products. For software development organizations, adhering to ISO 9001 ensures structured processes and continuous improvement in quality management.

ISO 9001 Requirements Summary

ISO 9001 outlines various requirements for software development organizations:

- **Management Responsibility**: Establishing and implementing an effective quality policy, defining roles and responsibilities impacting quality, and appointing a management representative to oversee the quality system impartially.
- **Quality System**: Maintaining and documenting a quality system to ensure consistency and improvement.
- **Contract Review**: Reviewing contracts to ensure understanding and capability to fulfill commitments.
- **Design Control**: Regulating the design process, including coding control, validation of design inputs and outputs, managing design changes, and utilizing configuration management systems.
- **Document Control**: Establishing protocols for document approval, distribution, and revision control, often requiring configuration management tools.
- **Purchasing**: Evaluating purchased materials, including software, for compliance with requirements.
- **Product Identification**: Ensuring products are identifiable throughout the development process, akin to configuration management in software.
- **Process Control**: Implementing process controls and documenting quality requirements.
- **Inspection and Testing**: Conducting comprehensive testing (unit, integration, system), maintaining test records, and ensuring product conformity.
- **Nonconforming Product Control**: Managing and controlling nonconforming software to prevent its unintended use.
- **Corrective Action**: Addressing errors, analyzing root causes, and enhancing processes to prevent recurrence.

- **Handling**: Managing the storage, packing, and delivery of software products.
- **Quality Audits**: Performing audits to verify the effectiveness of the quality system.
- **Training**: Identifying and providing necessary training for personnel involved in quality management.

11.6 SEI CMM and Six Sigma

SEI Capability Maturity Model (SEI CMM)

The SEI CMM is a model that assesses and improves the maturity of an organization's software processes. It categorizes organizations into five maturity levels:

- **Level 1 (Initial)**: Ad hoc processes with no defined practices, resulting in unpredictable outcomes.
- **Level 2 (Repeatable)**: Basic project management practices established for repeatable project success.
- **Level 3 (Defined)**: Defined processes for both management and development activities, focusing on standardization and documentation.
- **Level 4 (Managed)**: Emphasis on metrics to measure process and product quality, aiming for quantitative quality objectives.
- **Level 5 (Optimized)**: Continuous process improvement through quantitative feedback and innovation, implementing best practices across the organization.

Six Sigma

Six Sigma is a data-driven methodology aimed at improving process quality by minimizing defects and variations. It focuses on achieving near-perfect processes by reducing defects to less than 3.4 per million opportunities. Six Sigma employs two primary methodologies:

- **DMAIC (Define, Measure, Analyze, Improve, Control)**: Used for incremental improvement of existing processes that do not meet specifications.
- **DMADV (Define, Measure, Analyze, Design, Verify)**: Employed for developing new processes or products when existing ones do not suffice.

Six Sigma is widely applicable across industries, aiming to optimize processes for improved quality, reduced costs, and enhanced efficiency.

These frameworks—ISO 9001, SEI CMM, and Six Sigma—provide structured approaches to achieving and maintaining high software quality, each focusing on specific aspects of quality management and process improvement.

11.7 Keywords

Reliability

Reliability in software refers to the dependability and trustworthiness of a software product to perform correctly over time. It is essential to quantify reliability using metrics specified in the Software Requirements Specification (SRS) document. These metrics must be observer-independent to accurately measure and ensure the required level of reliability across different software products and categories.

Reliability Metrics

Reliability metrics vary across different types of software products and are crucial for determining the software's overall dependability. Despite the importance of reliability metrics, accurately quantifying software reliability remains a significant challenge in practice.

Software Quality

Software quality traditionally refers to the extent to which a software product meets its intended purpose. While fitness for purpose is sufficient for physical products like automobiles or machines, it alone does not adequately define software quality. For software products, quality encompasses factors such as usability, portability, reusability, correctness, and maintainability, in addition to meeting the SRS requirements.

Six Sigma

Six Sigma is a methodology pioneered by companies like Motorola and General Electric (GE) to enhance process efficiency and reduce defects. It aims to optimize processes across various business functions, including production, human resources, customer service, and more. Six Sigma uses data-driven approaches like DMAIC (Define, Measure, Analyze, Improve, Control) and DMADV (Define, Measure, Analyze, Design, Verify) to achieve near-perfect performance and quality standards.

ISO 9000

The ISO 9000 series, established by the International Organization for Standardization (ISO), provides guidelines for implementing quality management systems across industries. ISO 9000

certifications serve as benchmarks for assessing an organization's ability to consistently meet customer and regulatory requirements. These standards are crucial in contract negotiations and signify an organization's commitment to quality management and process improvement.

These keywords highlight fundamental concepts in software reliability, quality management systems (ISO 9000), process improvement methodologies (Six Sigma), and the challenges associated with quantifying software reliability through appropriate metrics.

11.8 Self-Assessment Questions

1. Determine who is responsible for quality assurance efforts in a software development organisation. Explain the primary tasks they conduct to fulfil their responsibility.
2. Whose responsibility is it in a software development organisation to ensure that the products are of high quality? Explain the primary tasks they conduct to fulfil their responsibility.
3. Create six measures to assess programme reliability. Do you think these measures are completely adequate for measuring a system's reliability?Justify your response.
4. Compare the properties of the Jelinski and Moranda Model and Littlewoodand Verall's Model in the context of reliability growth modelling.
5. Discuss the advantages and disadvantages of ISO 9001 certification and theSEI CMM-based quality evaluation.

11.9 Case Study

Introduction: Company XYZ, a software development firm, has encountered significant challenges regarding reliability and quality management in their software products. This case study delves into their specific issues, proposes solutions, provides recommendations, and concludes with key insights.

Problem: High Defect Rate and Customer Complaints

Company XYZ has been plagued by a high incidence of software defects and frequent customer complaints regarding reliability and quality issues. These issues have adversely affected customer satisfaction and eroded trust in their products.

1. Inconsistent Quality Assurance Processes

The company suffers from inconsistent quality assurance (QA) processes across different development teams. This lack of standardization results in varying testing methodologies, defect tracking practices, and overall quality management approaches. Consequently, software quality levels fluctuate widely across different products and projects.

2. Limited Reliability Testing

Company XYZ has not allocated adequate resources and effort to reliability testing. This deficiency has led to potential stability and performance issues, causing unexpected system failures, downtime, and increased support and maintenance costs.

Recommendations:

1. Implement Comprehensive Quality Assurance Processes
    - Establish standardized QA processes encompassing test planning, test case development, and execution methodologies.
    - Introduce quality checkpoints throughout the software development lifecycle to ensure adherence to defined standards.
    - Deploy a robust defect tracking and management system to effectively monitor and resolve software defects.
2. Focus on Test Automation
    - Invest in test automation frameworks and tools to enhance test coverage and efficiency.
    - Automate repetitive and critical test cases to achieve consistent and reliable test results.
    - Regularly review and update automated test suites to accommodate evolving requirements and functionalities.
3. Prioritize Reliability Testing
    - Allocate dedicated resources and time for comprehensive reliability testing activities.
    - Conduct stress testing, load testing, and performance testing to uncover potential bottlenecks and stability issues.
    - Monitor system behavior under various conditions and simulate real-world usage scenarios to ensure dependable performance.
4. Continuously Improve Through Feedback and Data Analysis
    - Encourage feedback from customers, support teams, and end-users to identify quality issues and areas for enhancement.

- Analyze data from bug reports, customer complaints, and user feedback to identify recurring issues and prioritize their resolution.
- Conduct regular retrospectives to evaluate QA practices, pinpoint improvement opportunities, and implement necessary refinements.

Conclusion:

Reliability and quality management are pivotal for software development companies to uphold customer satisfaction, reduce support expenses, and sustain competitiveness. By adopting comprehensive QA processes, emphasizing test automation, prioritizing reliability testing, and continuously improving based on feedback and data insights, organizations can elevate the reliability and quality of their software products. Company XYZ stands to benefit significantly from these recommendations, fostering consistency, efficiency, and customer-centricity in their software development practices. Prioritizing reliability and quality ensures the creation of robust and trustworthy software solutions, ultimately enhancing customer satisfaction and securing long-term success.

11.10 Reference

- Rajib Mall: Fundamentals of Software Engineering, 4th Edition, Prentice Hall of India, 2014
- Summerville: Software Engineering 10th Edition, Pearson Education, 2017 23
- 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 - 12

Computer Aided Software Engineering (CASE)

Learning Objectives:

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

Structure:

Introduction

In this chapter, we delve into Computer-Aided Software Engineering (CASE) and its role in enhancing software development and maintenance processes. CASE tools have garnered significant attention in the software industry due to their potential to reduce costs and effort associated with software projects.

12.1 Scope of CASE

CASE tools encompass a broad spectrum of software automation instruments designed to streamline various activities within the software development lifecycle. These tools automate tasks across phases such as project management, configuration management, specification, structured analysis, design, coding, and testing. The primary objectives of employing CASE tools include boosting productivity and facilitating the cost-effective production of high-quality software products.

12.2 Benefits of CASE

The adoption of CASE environments and standalone CASE tools offers several compelling advantages:

- **Cost Savings:** Studies indicate that implementing CASE systems can reduce development effort by 30 to 40%. This cost reduction is attributed to automation, which minimizes manual effort and accelerates development timelines.
- **Improved Quality:** CASE tools contribute to enhanced software quality by facilitating iterative development processes and reducing the likelihood of human error. They streamline workflows and ensure consistency across different phases of software engineering.
- **Documentation Efficiency:** CASE tools centralize critical information about software projects, reducing redundancy and the risk of conflicting documentation. This centralized approach improves documentation accuracy and accessibility.
- **Reduced Tedium:** Automation provided by CASE tools eliminates mundane tasks, allowing software engineers to focus on more strategic and creative aspects of development. For example, tasks like verifying Data Flow Diagrams (DFDs) can be efficiently managed with automated tools.
- **Enhanced Maintenance:** CASE environments support systematic data collection throughout the software development lifecycle. This data aids in effective traceability and consistency checks, significantly reducing costs associated with software maintenance.

- **Organizational Impact:** Implementing a CASE environment fosters organizational structure and discipline. It standardizes development practices, promotes collaboration, and improves overall efficiency.

CASE Environment

While individual CASE tools offer specific benefits, their full potential is realized when integrated into a cohesive CASE environment or framework. A CASE environment typically includes a central repository, often referred to as a data dictionary, where all project-related data items are defined and stored. This central repository enables seamless data exchange among different CASE tools operating across various stages of the software development lifecycle. Integrating tools through a shared repository enhances data consistency, improves collaboration, and supports comprehensive project management.

Conclusion

CASE tools and environments play a crucial role in modern software development by automating routine tasks, improving productivity, and ensuring higher software quality. For companies like Company XYZ facing challenges with software reliability and quality management, adopting CASE solutions can lead to substantial improvements in efficiency, cost-effectiveness, and customer satisfaction. By leveraging CASE tools effectively, organizations can streamline their development processes, mitigate risks, and achieve sustainable growth in a competitive market.

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.

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

Computer-Aided Software Engineering (CASE) tools provide significant support throughout various stages of the software development life cycle (SDLC). These tools are instrumental in enforcing development methodologies and ensuring consistency across different phases. Let's explore the specific contributions of CASE tools in different phases:

a. Prototyping Support

Prototyping is crucial for understanding complex software requirements, conceptualizing ideas, and validating concepts. A prototyping CASE tool should fulfill the following prerequisites and features:

- Define user interactions and system control flow.
- Manage data storage, retrieval, and processing logic.
- Interface with a data dictionary for consistency and integration.
- Enable graphical user interface (GUI) development through a graphics editor.
- Support integration with external modules in high-level programming languages like C.
- Allow user-defined operation sequencing and management of prototype runtime behavior.

b. Structured Analysis and Design

For structured analysis and design phases, CASE tools facilitate diagramming and ensure methodological support:

- Support various diagramming approaches for structured analysis and design methodologies.
- Enable creation of complex diagrams with hierarchical levels.
- Facilitate seamless navigation between different levels of design and analysis.
- Conduct consistency and completeness checks across the analysis hierarchy.
- Temporarily disable consistency testing for complex operations to optimize performance.

c. Code Generation

During the code generation phase, CASE tools aid in transforming design data into executable code:

- Generate module skeletons or templates in popular programming languages.
- Automatically create records, structures, and class definitions from the data dictionary.

- Generate database tables for relational database management systems (RDBMS).
- Provide user interface code for X Window and MS Windows-based applications based on prototype definitions.

d. Test Case Generator

CASE tools assist in generating comprehensive test cases to ensure software quality:

- Support testing based on both design specifications and requirements.
- Produce ASCII-formatted test set reports that can be directly integrated into test plan documents.

12.4 Second-generation CASE tools

Second-generation CASE tools offer advanced capabilities to support customized methodologies and integration:

- Adaptability to modified methodologies through CASE administration.
- Intelligent diagramming capabilities for automated and aesthetically pleasing diagrams.
- Integration with implementation environments to bridge design and development phases.
- Standardization of data dictionary usage across multiple development tools.
- Customization support for creating new object types and connections.
- Integration with rule engines to enforce methodological consistency checks.

12.5 CASE Environment Architecture

The figure 12.1 depicts the architectural layout of a typical modern CASE environment. A modern CASE environment must include a user interface, tool set, object management system (OMS), and repository.

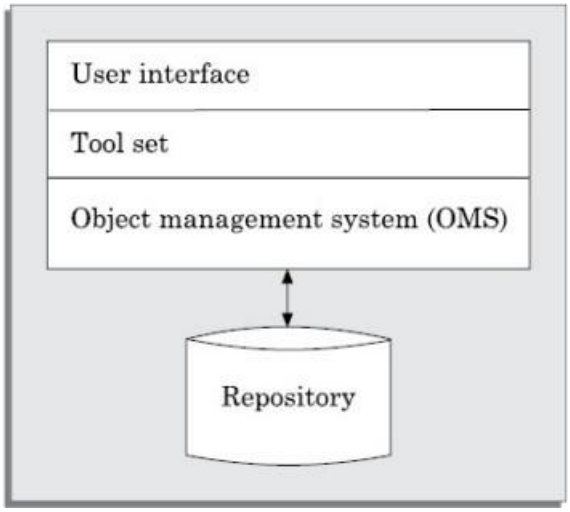

Figure12.1: Architecture of Modern CASE environment

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 utilise the various tools.

12.6 Summary

CASE tools are indispensable in modern software engineering, automating tasks across the SDLC, ensuring methodological adherence, and improving overall productivity and quality. Companies like Company ABC can benefit significantly by adopting CASE tools, streamlining processes, reducing costs, and accelerating software delivery. By leveraging these tools effectively, organizations can enhance collaboration, promote code reusability, and maintain high standards of software development, thereby gaining a competitive edge in the industry.

12.7 Self-Assessment Questions:

1. What are the primary functions and applications of Computer-Aided Software Engineering (CASE) tools in software development?
2. How do CASE tools enhance productivity and quality in the software development process? Provide specific examples of benefits.
3. At which stages of the software life cycle are CASE tools most beneficial, and how do they contribute to each stage?

4. What are the key features that distinguish second-generation CASE tools from the first generation, and how do these features improve software development?

12.8 References

1. I. Summerville: Software Engineering 10th Edition, 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, McGraw Hill Education, 2009