

Software Engineering

COURSE CODE: B21CA08DC
Bachelor of Computer Applications
Discipline Core Course
Self Learning Material



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

Software Engineering
Course Code: B21CA08DC
Semester - IV

Discipline Core Course
Undergraduate Programme
Bachelor of Computer Applications
Self Learning Material
(With Model Question Paper Sets)



SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala



SOFTWARE ENGINEERING

Course Code: B21CA08DC

Semester- IV

Discipline Core Course

Bachelor of Computer Applications

Academic Committee

Dr. Aji S.
Sreekanth M. S.
P. M. Ameera Mol
Dr. Vishnukumar S.
Shamly K.
Joseph Deril K. S.
Dr. Jeeva Jose
Dr. Bindu N.
Dr. Priya R.
Dr. Ajitha R. S.
Dr. Anil Kumar
N. Jayaraj

Development of the Content

Greeshma P.P.
Shamin S.
Anjitha A.V.
Suramya Swamidas P.C.
Dr. Kanitha Divakar
Aswathy V.S.
Sreerekha V.K.

Review and Edit

Dr. Viji Balakrishnan

Linguistics

Sujith Mohan

Scrutiny

Shamin S., Subi Priya Laxmi S.B.N.,
Greeshma P.P., Sreerekha V.K.,
Anjitha A.V., Aswathy V.S.,
Dr. Kanitha Divakar.

Design Control

Azeem Babu T.A.

Cover Design

Jobin J.

Co-ordination

Director, MDDC :
Dr. I.G. Shibi
Asst. Director, MDDC :
Dr. Sajeekumar G.
Coordinator, Development:
Dr. Anfal M.
Coordinator, Distribution:
Dr. Sanitha K.K.



Scan this QR Code for reading the SLM
on a digital device.

Edition
July 2025

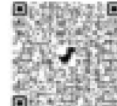
Copyright
© Sreenarayanaguru Open University

ISBN 978-81-989642-0-5



All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.m



Visit and Subscribe our Social Media Platforms

Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayana-guru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of “access and quality define equity,” laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from compromising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical framework of the widely recognized “blended format.” Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The University aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline’s evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web development.

We assure you that the university student support services will closely stay with you for the redressal of your grievances during your student-ship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.



Regards,
Dr. Jagathy Raj V.P.

01-07-2025

Contents

Block 01	Software Lifecycle Models	1
Unit 1	Evolution	2
Unit 2	Software Development Life Cycle	11
Unit 3	Traditional process Models	19
Unit 4	Comparison of different life cycle models	30
Block 02	Software Project Management	41
Unit 1	Strategic Management of Software Projects	42
Unit 2	Project Size Estimation Techniques	54
Unit 3	Cpm, Pert and Gantt Chart	75
Unit 4	Software Requirements Elicitation and SRS	99
Block 03	Software Design	117
Unit 1	Software Design	118
Unit 2	Software Design Strategies	132
Unit 3	Design Approaches	141
Unit 4	Software Analysis and Design Tools	149
Block 04	Coding and Testing	161
Unit 1	Coding standards and Code Review	162
Unit 2	Software Reliability and Quality Management	175
Unit 3	Testing	190
Unit 4	Deployment	213
Block 05	Object - Oriented Design Concepts	224
Unit 1	Unified Modeling Language	225
Unit 2	Static and dynamic diagrams	236
Unit 3	Design Walkthroughs	254
Unit 4	Other UML Diagrams	278
Block 06	DevOps	308
Unit 1	Introduction to DevOps	309
Unit 2	DevOps Life Cycle	320
Unit 3	Agile Manifesto and Scrum Framework	330
Unit 4	Continuous Integration and Deployment	341
Model Question Paper Sets		350



Software Lifecycle Models



Evolution

Learning Outcomes

At the end of this unit, the learner will be able to:

- ♦ identify the stages in the evolution of software development.
- ♦ define the difference between a software program and a software product.
- ♦ list the key features of a software program.
- ♦ define the terms software program & software product.

Prerequisites

Most students today have used popular applications like WhatsApp, Microsoft Excel, or Google Chrome on their phones or computers. You may have also written a simple Python or Java program during your school or college lab sessions. These experiences, though different, are part of the same field—software development.

You already know that some software tools are small and created for specific needs, like a program to calculate student marks. Others are large, with millions of users and regular updates, like Microsoft Word or Instagram. But have you ever thought about how such software evolved over time or what makes one a program and the other a product?

In this unit, we will study how software has grown from basic machine-level instructions to intelligent systems that understand human language. We will also learn the difference between a software program (used by a few people) and a software product (used by many around the world). For example, while a student-made mark calculator is a program, a full-featured app like Zoom is a software product.

By comparing features like user base, development process, testing, and documentation, you will learn to differentiate between software programs and products. This understanding forms the base for your future studies in software engineering and real-world application development.

Keywords

software evolution, software program, software product, software comparison, development process

Discussion

1.1.1 Evolution of Software

The evolution of software refers to the gradual improvement in how software is developed, used, and maintained over time. From early machine-level instructions to modern intelligent systems powered by artificial intelligence, software has undergone significant changes. Each phase of evolution reflects advancements in hardware, user needs, and development methodologies. Below are the major stages in the evolution of software:

1.1.1.1 Machine and Assembly Language Era (1940s–1950s)

In the earliest days of computing, software was written in machine language, consisting of binary code (0s and 1s) that could be directly executed by computer hardware. Programming was extremely difficult and time-consuming. To make things slightly easier, assembly language was introduced. It used mnemonics or symbolic instructions, which were then converted to machine code using an assembler. Software at this stage was tightly coupled with the hardware and was mainly developed for scientific or military use.

Example: Early computers like ENIAC were programmed for tasks such as ballistic calculations using machine code.

1.1.1.2 High-Level Programming and Operating Systems (1960s–1970s)

This period saw the development of high-level programming languages like FORTRAN, COBOL, ALGOL, and C. These languages allowed programmers to write instructions in a more human-readable format, making software development more efficient. During the same time, operating systems like UNIX were introduced to manage hardware, run multiple programs, and simplify interactions between users and computers. Software became more portable and reusable across different hardware platforms.

Example: FORTRAN was used for engineering and scientific applications, while COBOL was widely used in business data processing systems.

1.1.1.3 GUI-Based and Commercial Software Era (1980s–1990s)

With the arrival of personal computers (PCs), software began to reach individual users and not just organizations. Developers started creating software with graphical user interfaces (GUIs), allowing users to interact through icons and windows instead



of typing commands. The software industry began to grow rapidly, and software was packaged and sold as a commercial product.

Example: Microsoft Windows offered a GUI environment that made computing accessible to the public. Applications like Microsoft Word and Excel became standard tools for office work.

1.1.1.4 Web and Internet Software Era (Late 1990s–2000s)

The emergence of the Internet transformed software into an online service. Developers started building web-based applications that could run inside a browser without needing local installation. Software was now accessible globally, and users could interact with it in real time. New programming languages and tools such as HTML, JavaScript, and PHP became standard for web development.

Example: Gmail allowed users to access email from any device with the Internet. Amazon revolutionized online shopping with its e-commerce platform.

1.1.1.5 Mobile and Cloud-Based Software Era (2010s–Present)

The rise of smartphones and cloud computing changed the way people use and interact with software. Applications (apps) became available on mobile app stores, and data could be stored and accessed on the cloud. Software could now sync across multiple devices, provide location-based services, and offer real-time communication. Cloud platforms allowed developers to build and scale software without investing in physical infrastructure.

Example: WhatsApp and Instagram became essential mobile apps for communication and social media. Google Drive and Dropbox allowed users to store and share files in the cloud.

1.1.1.6 Intelligent and AI-Driven Software Era (Present and Future)

Today, software is becoming increasingly intelligent through the integration of artificial intelligence (AI) and machine learning (ML). Applications can now learn from data, make predictions, and automate decision-making. Software is evolving from being a tool to being a smart assistant that can personalize services and interact naturally with users. This era is also seeing developments in robotics, natural language processing, and autonomous systems.

Example: ChatGPT can engage in human-like conversations and assist in various tasks. Amazon Alexa and Google Assistant respond to voice commands and control smart devices.

1.1.2 Software Development Projects

Software development projects involve the process of creating software applications that fulfill specific user needs. These projects can range from simple one-time scripts to complex, large-scale systems that are used by thousands of people. Every software

development project includes stages like requirement analysis, design, implementation, testing, deployment, and maintenance. Depending on the purpose and scope, software development projects can result in either a program or a product.

Types of Software Development Projects

There are mainly two types of software outcomes based on the development approach and purpose:

- ◆ Software Program
- ◆ Software Product

1.1.2.1 Software Program

A software program is a small, self-contained set of instructions written to solve a specific task or problem. It is usually created by an individual or a small team for personal use, academic projects, internal company use, or automation of simple tasks. Software programs are not intended for public release and usually do not follow complex development processes.

Key Features of a Software Program:

- ◆ Developed for a specific and limited purpose.
- ◆ Used by a known group of users (often the developer themselves).
- ◆ Written using any programming language like C++, Java, Python, etc.
- ◆ May not include documentation, testing, or user support.
- ◆ Often lacks scalability and reusability.

Example:

A college student creates a Java program that calculates the total marks and grade of students. This program works only for a specific input format and is used only by the student or teacher who developed it.

1.1.2.2 What is a Software Product?

A software product is a fully developed, tested, and packaged software application designed to meet the needs of a wide range of users or customers. It is often sold or distributed for commercial or public use. The development of a software product follows a structured software development life cycle (SDLC) and involves multiple teams such as developers, testers, designers, and support staff.

Key Features of a Software Product

- ◆ Built for commercial or general use.
- ◆ Used by a large and unknown set of users.
- ◆ Follows a standard development process including requirement analysis,



design, implementation, testing, deployment, and maintenance.

- ◆ Requires user documentation, installation instructions, and regular updates.
- ◆ High quality standards, including performance, security, and user interface.

Example:

Microsoft Excel is a software product used worldwide for spreadsheet calculations. It has many features, supports different platforms, and is updated regularly to improve performance and add new functions.

1.1.3 Comparison between Program and Product

Table 1.1.1 Comparison between Program and Product

Feature	Software Program	Software Product
Purpose	Made to solve a specific task or small problem	Made to solve the needs of many users
Users	Known users like the developer or a small team	Unknown or large number of users (public/customers)
Development Team	Usually developed by one person or a small team	Developed by a big team with different roles
Development Process	Simple, may not follow a full development process	Follows complete Software Development Life Cycle
Testing	May have little or no testing	Tested at many levels to ensure quality
Documentation	May not have proper documentation	Has user manuals and technical documents
Support and Updates	Rarely updated or maintained	Regular updates, bug fixes, and user support
Use Duration	Short-term or one-time use	Long-term use with multiple versions
Examples	Python program to calculate marks	Microsoft Excel, Google Chrome, WhatsApp

Recap

Evolution of Software

- ◆ Machine and Assembly Language Era (1940s–1950s)
 - Software written in binary or assembly language; hardware-dependent
 - Example: ENIAC used for ballistic calculations
- ◆ High-Level Programming and Operating Systems (1960s–1970s)
 - Introduction of FORTRAN, COBOL, UNIX OS
 - Software became portable and reusable
 - Example: COBOL in business applications
- ◆ GUI-Based and Commercial Software Era (1980s–1990s)
 - Personal computers and graphical user interfaces
 - Commercial software industry grew
 - Example: MS Word, Excel, Windows
- ◆ Web and Internet Software Era (Late 1990s–2000s)
 - Web-based applications accessible globally
 - Use of HTML, JavaScript, PHP
 - Example: Gmail, Amazon
- ◆ Mobile and Cloud-Based Software Era (2010s–Present)
 - Mobile apps, cloud services, real-time sync
 - Example: WhatsApp, Google Drive, Instagram
- ◆ Intelligent and AI-Driven Software Era (Present and Future)
 - AI and ML integrated into software
 - Personalized, predictive, and voice-controlled systems
 - Example: ChatGPT, Alexa, Google Assistant

Software Development Projects

- ◆ Software projects create applications to meet user needs

- ♦ Follow stages like analysis, design, coding, testing, deployment, and maintenance
- ♦ Outcomes: Can be a Software Program or Software Product

Software Program

- ♦ Small, single-purpose applications
- ♦ Created by individuals or small teams
- ♦ Usually for personal, academic, or internal use
- ♦ No formal process or documentation
- ♦ Example: Java program to calculate student marks

Software Product

- ♦ Developed for commercial or public use
- ♦ Structured SDLC followed
- ♦ Built by large teams
- ♦ Includes testing, support, documentation
- ♦ Example: Microsoft Excel, Google Chrome

Objective Type Questions

1. What was the first generation of software written in?
2. Which language uses mnemonics and symbolic instructions?
3. Name the high-level language commonly used in scientific applications during the 1960s.
4. What operating system introduced multitasking in the 1970s?
5. Which user interface type became popular in the 1980s?
6. Name one commercial software developed by Microsoft in the 1990s.
7. Which decade saw the rise of web-based applications?
8. Which programming language is widely used for web development?
9. What type of computing allows access to software via the Internet?
10. Which mobile app is known for instant messaging and voice calls?

11. What technology powers intelligent software today?
12. Name a voice-based virtual assistant developed by Amazon.
13. What is the outcome of a small software created for personal use called?
14. What is the structured, fully developed software meant for mass users called?
15. What development process is followed in software product creation?

Answers to Objective Type Questions

1. Machine
2. Assembly
3. FORTRAN
4. UNIX
5. GUI
6. Excel
7. 1990s
8. JavaScript
9. Cloud
10. WhatsApp
11. AI
12. Alexa
13. Program
14. Product
15. SDLC

Assignments

1. Explain the different stages in the evolution of software with suitable examples.
2. Describe the importance of high-level programming languages in the evolution of software.
3. What role did the Internet play in changing the nature of software development? Give examples.
4. Differentiate between machine language and assembly language with examples.
5. Define a software program. Mention its key features and give a suitable example.
6. What is a software product? How is it different from a software program?
7. List and explain any five key differences between a software program and a software product.

Suggested Reading

1. Sommerville, Ian. "Software engineering (ed.)." *America: Pearson Education Inc* (2011).
2. Van Vliet, Hans, Hans Van Vliet, and J. C. Van Vliet. *Software engineering: principles and practice*. Vol. 13. Hoboken, NJ: John Wiley & Sons, 2008
3. Pressman, Roger S. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
4. Lawrence, Pfleeger Shari. *Software engineering: theory and practice*. Pearson Education India, 2006.

Reference

1. <https://archive.nptel.ac.in/courses/106/105/106105182/>



Software Development Life Cycle

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ Define the term Software Development Life Cycle (SDLC).
- ◆ List the different phases of SDLC in order.
- ◆ Identify the purpose of each phase in the SDLC.
- ◆ Name the document created during the requirement gathering phase.
- ◆ Recall various types of software testing used in the SDLC process.

Prerequisites

Think about building a new house. The process does not begin with construction right away. First, you sit with your family to understand what kind of house you all want, how many rooms, what style, and what features are needed. Then, you consult with an architect who creates a design or blueprint. After finalizing the design, the construction team starts building the house step by step. Once built, everything is checked to ensure it works properly like plumbing, electricity, and safety measures. After you move in, regular maintenance is needed, such as fixing leaks or repainting walls.

In the same way, software is not created in a single step. It begins with understanding the user's needs, then designing a plan, followed by actual coding, testing to fix issues, and finally launching it for real users. Even after deployment, the software needs regular updates and improvements. This structured process is known as the Software Development Life Cycle (SDLC).

By learning about SDLC, you will understand how software products like mobile apps, websites, or business applications are carefully developed through clearly defined stages, just like building a strong and functional house.

Keywords

Software Development, System Design, Testing, Implementation, Maintenance

Discussion

1.2.1 Software Development Life Cycle (SDLC)

The Software Development Life Cycle (SDLC) is a structured and systematic process followed for developing software applications. It defines a series of phases that provide a framework for planning, creating, testing, and maintaining software systems. The main goal of SDLC is to produce high-quality software that meets or exceeds customer expectations, is completed within time and cost estimates, and functions effectively in the real world. The typical phases of SDLC include requirement gathering and analysis, system design, implementation (coding), testing, deployment, and maintenance. By following these steps in a disciplined manner, developers can manage projects more efficiently, reduce the chances of errors, and ensure continuous improvement of the software throughout its lifecycle.

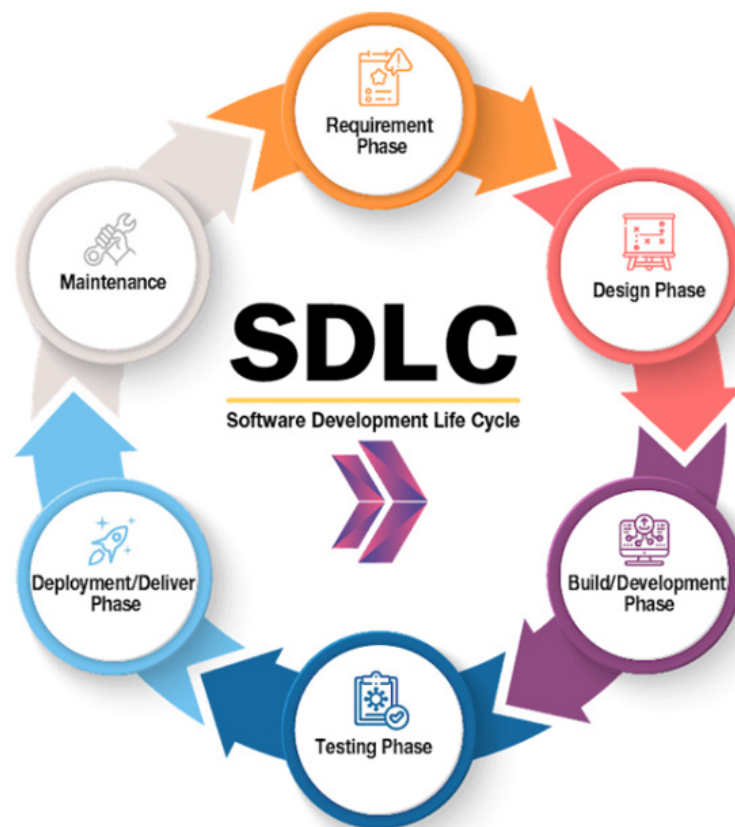


Figure 1.2.1 Software Development Life Cycle (SDLC)

1.2.2 Stages of Software Development Life Cycle (SDLC)

The SDLC is divided into several stages. Each stage has its own tasks and goals. Let's go through each one in detail:

1.2.2.1 Requirement Gathering and Analysis

The first phase of the Software Development Life Cycle is Requirement Gathering and Analysis. In this phase, the software development team meets with the client or stakeholders to understand what the software should do. The main goal is to collect all necessary information about the features, functions, and expectations of the users. For example, if the software being developed is an online shopping application, the client may ask for features like user registration, product search, shopping cart, and secure payment options. These requirements are documented clearly in a Software Requirement Specification (SRS) document, which serves as a reference for the next phases.

Key Features

- ◆ Interaction with stakeholders to collect information.
- ◆ Identification of business and user needs.
- ◆ Creation of Software Requirement Specification (SRS).
- ◆ Feasibility analysis (technical, financial, and time-based).

1.2.2.2 System Design

After gathering and analyzing the requirements, the next phase is System Design. In this stage, the structure and design of the software are planned. This includes both high-level design (overall system architecture, databases, and technologies to be used) and low-level design (detailed design of individual modules and their interactions). The goal is to create a blueprint of the system that guides developers during coding. For instance, in the shopping app example, the design will define how the user interface looks, how data flows between the product catalog and shopping cart, and how the payment module integrates with third-party payment gateways.

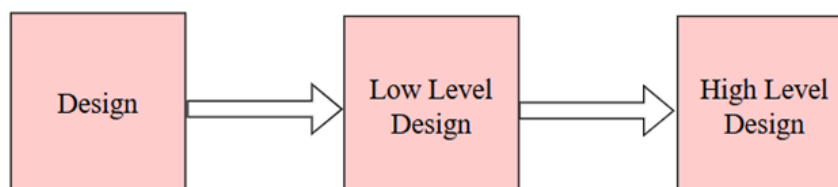


Figure 1.2.2 System design

Key Features:

- High-level design (architecture, tech stack).
- Low-level design (module details, database schema).
- User Interface (UI) and User Experience (UX) design.
- Security and performance planning.

1.2.2.3 Implementation (Coding)

The Implementation or Coding phase is where the actual construction of the software takes place. Developers write the program code using the design documents prepared in the previous phase. This work is often divided among team members based on modules. For example, one developer may code the login system, another may work on the product display, while another handles the checkout process. The coding must follow coding standards and best practices to ensure readability, maintainability, and performance. At the end of this phase, all the modules are combined to form the complete software system.

Key Features:

- ◆ Actual development of code.
- ◆ Use of programming languages and tools.
- ◆ Division of work into modules.
- ◆ Version control and collaboration using tools like Git.

1.2.2.4 Testing

Once the software is built, it enters the Testing phase. The goal of this stage is to identify and fix any bugs or errors in the software. Testing ensures that the software behaves as expected and meets the client's requirements. Various types of testing are performed, such as unit testing (testing individual parts of the code), integration testing (testing how modules work together), and system testing (testing the entire software). For example, testers may check if the shopping app correctly adds items to the cart, applies discounts, and processes payments securely. Only after successful testing can the software proceed to the deployment phase.

Key Features:

- Identification and fixing of bugs and errors.
- Various testing types: unit, integration, system, user acceptance.
- Use of automated and manual testing tools.
- Quality assurance to ensure reliability.

1.2.2.5 Deployment

The Deployment phase is when the software is released to the actual users. It may first be deployed to a limited group of users (known as beta testing) to gather feedback and identify any issues in the real-world environment. Once any final changes are made, the software is deployed on a larger scale. For example, the completed shopping app is uploaded to the Google Play Store or Apple App Store so that users can download and use it. Proper deployment ensures that the software is accessible, stable, and ready for use.

Key Features:

- ◆ Release of the software to real users.
- ◆ Use of cloud platforms or app stores.
- ◆ Initial release (beta) followed by full release.
- ◆ Monitoring performance post-deployment.

1.2.2.6 Maintenance

The final phase of SDLC is Maintenance. After the software has been deployed, it requires continuous monitoring and updates to fix any issues that users report. Maintenance also includes making changes to improve performance, add new features, or adapt to new hardware or operating systems. For instance, if users of the shopping app request a new payment method or report a bug in the cart system, the development team will work on updates to resolve those issues. Regular maintenance ensures that the software remains functional, secure, and up-to-date throughout its life.

Key Features:

- ◆ Fixing bugs reported by users.
- ◆ Adding new features or updating existing ones.
- ◆ Ensuring compatibility with new devices or OS updates.
- ◆ Continuous support and improvement.

Recap

- ◆ Definition of SDLC: A structured process for planning, creating, testing, and maintaining software.
- ◆ Purpose of SDLC: To build high-quality software that meets user expectations, within time and cost constraints.
- ◆ Phases of SDLC:
 - Requirement Gathering and Analysis

- Understand user needs and expectations.
- Create Software Requirement Specification (SRS).
- Perform feasibility analysis.
- ◆ System Design
 - Plan system architecture and module-level designs.
 - Design databases, user interfaces, and data flow.
 - Focus on security, performance, and user experience.
- ◆ Implementation (Coding)
 - Developers write code based on the design.
 - Work is divided into modules.
 - Use tools and practices like version control.
- ◆ Testing
 - Identify and fix bugs through unit, integration, and system testing.
 - Use manual and automated tools.
 - Ensure quality and reliability.
- ◆ Deployment
 - Release software to users (beta and full release).
 - Host on platforms like cloud or app stores.
 - Monitor performance after launch.
- ◆ Maintenance
 - Fix user-reported issues.
 - Add new features or enhancements.
 - Ensure software stays up-to-date and compatible.

Objective Type Questions

1. What is the full form of SDLC?
2. Which document is created during the requirement gathering phase?
3. Which SDLC phase involves writing the actual code?
4. What is the main goal of the testing phase?
5. Which SDLC phase ensures long-term support and updates for software?
6. Name the phase where system architecture is designed.
7. What type of testing checks how individual modules work together?
8. Which phase includes activities like user authentication and performance planning?
9. Which SDLC phase includes beta testing?
10. What is the primary outcome of the coding phase?
11. Which SDLC phase gathers input from clients or stakeholders?
12. Which type of design includes database schema and module interactions?
13. Name any two types of testing performed during the testing phase.

Answers to Objective Type Questions

1. Software Development Life Cycle
2. Software Requirement Specification (SRS)
3. Implementation (Coding)
4. To identify and fix bugs or errors
5. Maintenance
6. System Design
7. Integration Testing
8. System Design
9. Deployment

10. Fully functional software modules
11. Requirement Gathering and Analysis
12. Low-level design
13. Unit Testing and System Testing

Assignments

1. Explain the concept of the Software Development Life Cycle (SDLC) in your own words. Why is it important in software engineering?
2. Describe the activities involved in the Requirement Gathering and Analysis phase. Give an example from a real-life software application.
3. Write a short note on the System Design phase. What are high-level and low-level designs?
4. Discuss the importance of the Testing phase in SDLC. What types of testing are usually done?
5. List and explain the different phases of SDLC with suitable examples for each.

Suggested Reading

1. Sommerville, Ian. "Software engineering (ed.)." *America: Pearson Education Inc* (2011).
2. Van Vliet, Hans, Hans Van Vliet, and J. C. Van Vliet. *Software engineering: principles and practice*. Vol. 13. Hoboken, NJ: John Wiley & Sons, 2008
3. Pressman, Roger S. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
4. Lawrence, Pfleeger Shari. *Software engineering: theory and practice*. Pearson Education India, 2006.

Reference

1. <https://archive.nptel.ac.in/courses/106/105/106105182/>



Traditional process Models

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ Define the term Waterfall model.
- ◆ List the main phases of the Incremental model.
- ◆ Recall the purpose of risk analysis in the Spiral model.
- ◆ Identify the sequence of steps in the Waterfall model.
- ◆ Explain Spiral Model

Prerequisites

Before we begin learning about traditional software process models like the Waterfall, Incremental, and Spiral models, think about how you normally approach completing a school project or planning a birthday party. First, you gather ideas, then you make a plan, collect materials, carry out your tasks one by one, and finally, you review everything to see how it turned out. This step-by-step method ensures that you don't miss anything important.

For example, if you are planning a birthday party, you first list the requirements (like guest list, food, decorations), then design the layout, buy and prepare the items, set everything up, and after the party, you clean up and get feedback from friends. This is similar to the Waterfall Model in software development, where each step is completed before moving to the next.

Keywords

Waterfall model, Incremental model, Spiral model, Software development, Process models

Discussion

1.3.1 Traditional Process Models

In software engineering, traditional process models describe the step-by-step methods followed to develop software. These models provide a structured approach to software development and have been widely used in many projects over the years. Each model has a specific way of organizing tasks and handling the development process. Some of the most well-known traditional models are the Waterfall Model, Incremental Model, and Spiral Model. Let us understand each of them in detail.

1.3.1.1 Waterfall Model

The Waterfall Model is the earliest and simplest software development model. It follows a linear and sequential approach, where each phase must be completed before the next phase begins. This model is called the “Waterfall” Model because the development process flows downward through the phases like water flowing down a series of steps or a waterfall. Once a phase is finished, the process moves to the next one without going back, similar to how water cannot flow upward.

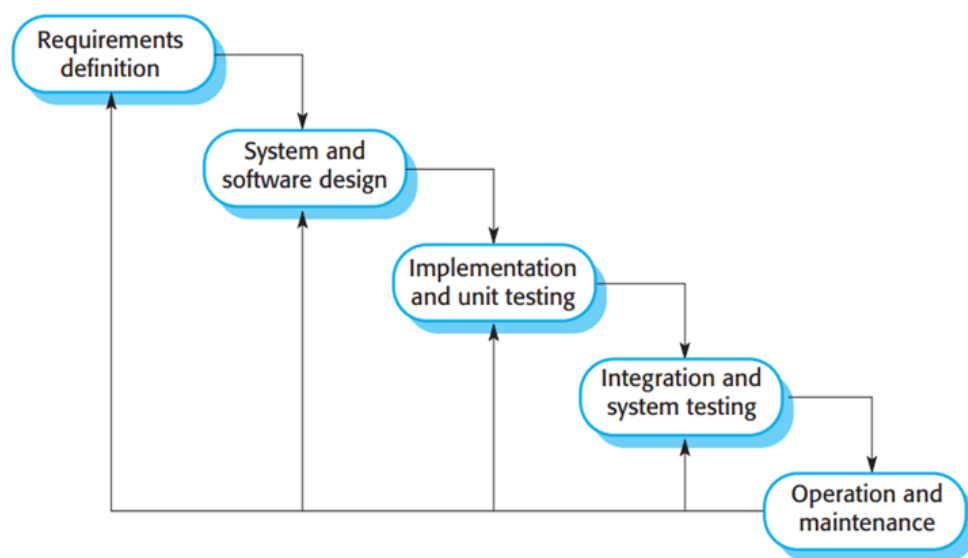


Fig. 1.3.1 Waterfall model

a. Requirements Definition

This is the first and most critical phase of the Waterfall Model. During this phase, all the requirements of the software system are gathered from the client or stakeholders. The goal is to understand what the software should do. This includes both functional requirements (features, behavior) and non-functional requirements (performance, security, usability).

Key Activities

- ◆ Conducting meetings with stakeholders and users
- ◆ Documenting user needs and expectations
- ◆ Creating a Software Requirements Specification (SRS) document
- ◆ Reviewing and approving the requirements

b. System and Software Design

After gathering requirements, the next phase is to plan how the system will be built. This includes architectural design (overall structure) and detailed design (internal components). Both the hardware and software architecture are considered, including how different modules will interact.

Key Activities

- ◆ Designing the system architecture
- ◆ Selecting technology stack (e.g., programming languages, frameworks)
- ◆ Creating data flow diagrams and database schemas
- ◆ Designing user interfaces (UI/UX)
- ◆ Preparing design documents for implementation

c. Implementation and Unit Testing

In this phase, the actual coding (programming) is done based on the design documents. The system is broken down into modules, each of which is developed separately. After coding, unit testing is performed to verify that individual components function correctly.

Key Activities:

- ◆ Writing code for each module
- ◆ Following coding standards and best practices
- ◆ Performing unit tests (testing small components)
- ◆ Debugging and fixing issues at the module level

d. Integration and System Testing

Once individual modules are tested, they are integrated into a complete system. This phase ensures that all components of the software function as intended. System testing is carried out to verify that the system meets the original requirements. It may include different types of testing like functional, performance, security, and user acceptance testing (UAT).

Key Activities

- ◆ Integrating modules into one system
- ◆ Testing overall functionality
- ◆ Identifying and fixing bugs in integrated environment
- ◆ Verifying performance and load capacity
- ◆ Conducting user acceptance tests

e. Operation and Maintenance

This is the final phase where the software is deployed to the actual users and maintained over time. Even after deployment, the software may need updates, error correction, or enhancements based on user feedback. Maintenance ensures that the software continues to perform well as technology and user needs evolve.

Key Activities

- ◆ Deploying the software to live environment
- ◆ Monitoring system performance
- ◆ Fixing issues reported by users
- ◆ Updating software for compatibility or new features
- ◆ Providing long-term technical support

1.3.1.2 Incremental Model

The Incremental Model is a software development approach where the system is built and delivered in small sections called increments, rather than developing the entire software at once. Each increment adds new features to the existing system until the full product is completed. This model combines the features of the Waterfall model with iterative development, making it more flexible and manageable, especially for medium to large-scale projects.

The process begins with the basic requirements, and the core functionalities are developed first. Once this part is complete, the software is released to the user, and feedback is gathered. Based on the feedback and remaining requirements, new features are planned and added in subsequent increments. This continues until the final product is delivered.

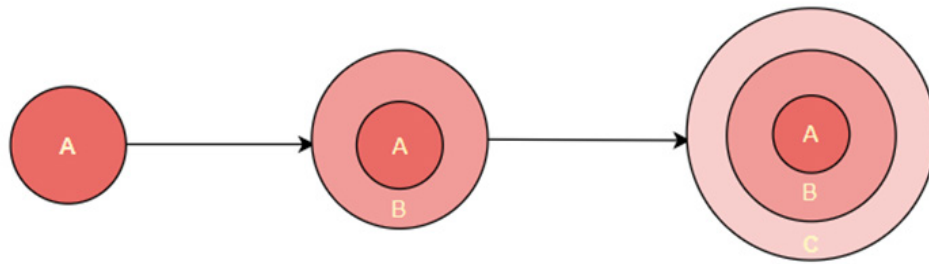


Fig. 1.3.2 Incremental Model

Phases of the Incremental Model

1. Requirement Analysis

In this phase, the development team interacts with the client to understand the overall system requirements. Unlike the Waterfall model where all requirements are fixed at the beginning, the Incremental Model allows for flexibility. Here, the requirements are analyzed and grouped into smaller sets (increments). Each group of requirements corresponds to a specific feature or functionality that can be developed and delivered independently.

2. System Design

Once the initial set of requirements is finalized, the team proceeds with designing the system architecture. This includes both high-level design (like database design, technology stack, and overall structure) and low-level design (like the logic of specific modules). Importantly, the design must be scalable so that new increments can be added later without breaking the system.

3. Implementation of the First Increment

In this phase, developers start coding the first increment of the system. This is usually the core functionality that forms the foundation of the software. Developers use programming tools and languages as per the design specifications. The first increment is kept small but functional, ensuring it can be tested and deployed on its own.

4. Testing of the Increment

After implementation, the increment undergoes thorough testing. This includes unit testing (for individual components), integration testing (for combined modules), and system testing (for the complete increment). The aim is to ensure that the developed features are bug-free and work as expected.

5. Delivery and Feedback

Once testing is complete, the increment is delivered to the end users. This may be in the form of a prototype or a functional module. Feedback is collected to assess whether

the system meets user expectations and if any changes are needed. This feedback guides the development of the next increment.

6. Planning and Development of Next Increments

Using the feedback from users and the remaining requirement sets, the team plans the next increment. The next set of features is designed, implemented, and tested, following the same process as the first increment. This cycle continues until all increments are developed and the complete system is ready.

1.3.1.3 Spiral Model

The Spiral Model is a software development process model that combines the features of both iterative development and the Waterfall model. It is especially useful for large, complex, and high-risk projects, where requirements may change or evolve during the development process.

The Spiral Model is called so because it is represented as a spiral diagram, where the process loops through repeated cycles (or spirals), with each loop representing a development phase. It allows developers to build a system incrementally, with repeated refinement and risk evaluation at every stage.

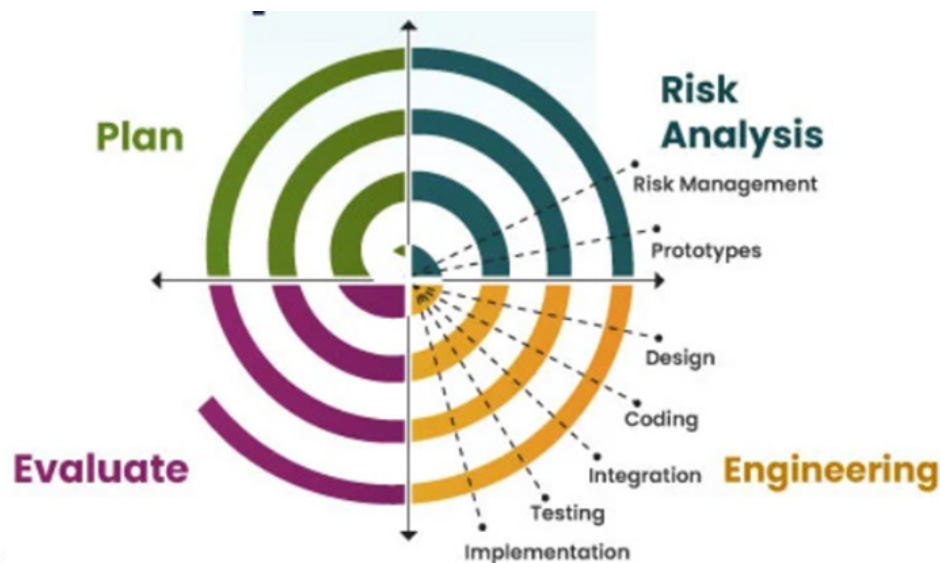


Fig. 1.3.3 Spiral Model

Phases of the Spiral Model

1. Planning Phase

In this phase, the project objectives, alternatives, and constraints are identified. The team gathers requirements from stakeholders and plans the initial stages of the software. This includes deciding the resources needed, timelines, and cost estimation. For example, if a company wants to build a mobile banking app, they'll first list key features like login, balance checking, money transfer, etc., and plan how to approach the development in phases.

Key Activities

- ◆ Identify business needs and technical requirements
- ◆ Define objectives and constraints
- ◆ Initial cost and schedule estimation
- ◆ Selection of development strategy

2. Risk Analysis Phase

This is a unique feature of the spiral model. Every cycle of the spiral includes identifying and analyzing risks that might affect the project. These could be technical risks, time risks, or cost-related risks. Based on this, risk mitigation strategies are planned. For instance, if integrating a payment gateway is a risky part of the app, the team might prototype that feature early to see how well it integrates.

Key Activities :

- ◆ Identify potential risks (budget overruns, technical failures, etc.)
- ◆ Perform risk assessment
- ◆ Plan strategies to reduce or avoid risks
- ◆ Develop and test prototypes if needed

3. Engineering Phase (Development and Testing)

This phase involves actual development and testing of the software. Based on the requirements and risk analysis, a version (or part) of the software is built. It could be a prototype or a full functional module. Testing is done to ensure the software works correctly. For the mobile banking app, the first version might just allow users to log in and view account balances.

Key Activities:

- ◆ Design the system architecture or module
- ◆ Write the code for that phase
- ◆ Perform unit and integration testing
- ◆ Document the design and test results

4. Evaluation Phase (Customer Evaluation and Review)

After each spiral iteration, the developed part of the software is shown to the client or stakeholders. Their feedback is gathered and used to refine requirements for the next iteration. This ensures that the software is always aligned with what the users want. For example, if users find the login process difficult, it will be improved in the next cycle.

Key Activities:

- ◆ Present developed module to stakeholders
- ◆ Gather feedback and suggestions
- ◆ Analyze feedback for refining the next cycle
- ◆ Update plans and documents as needed

5. Next Spiral Phase

Once all the above steps are completed, the next spiral loop begins. Each loop results in a more complete version of the software. The process is repeated until the software is fully developed and all requirements are met.

Recap

- ◆ Traditional Process Models
 - Step-by-step structured methods in software development.
 - Common traditional models: Waterfall Model, Incremental Model, Spiral Model.
- ◆ Waterfall Model
 - Linear and sequential model – complete one phase before the next.
- ◆ Phases:
 - Requirements Definition: Gather and document user requirements.
 - System and Software Design: Architectural and detailed design.
 - Implementation and Unit Testing: Code development and testing of individual modules.
 - Integration and System Testing: Combine modules and perform system-level testing.
 - Operation and Maintenance: Deploy, monitor, and update the system.
- ◆ Incremental Model
 - Development in small parts (increments).
 - Combines Waterfall + Iterative features.

- ◆ Phases:
 - Requirement Analysis: Gather and group requirements.
 - System Design: Plan scalable architecture and modules.
 - Implementation of First Increment: Code and test core functionality.
 - Testing of the Increment: Perform unit, integration, and system testing.
 - Delivery and Feedback: Deliver increment and gather feedback.
 - Next Increments: Plan and build remaining features in cycles.
- ◆ Spiral Model
 - Combines Waterfall + Iterative + Risk Analysis.
 - Suitable for complex, evolving, and high-risk projects.
- ◆ Phases (repeated in spirals):
 - Planning Phase: Define objectives, constraints, and timelines.
 - Risk Analysis Phase: Identify and plan for potential risks; prototype risky components.
 - Engineering Phase: Develop and test a version of the software.
 - Evaluation Phase: Stakeholder review and feedback.
 - Next Spiral Phase: Start next loop with updated plans.

Objective Type Questions

1. What is the first phase of the Waterfall model?
2. Which model follows a linear sequential approach?
3. Which phase in the Waterfall model involves coding and unit testing?
4. Which model is best suited for projects with well-understood requirements?
5. Name the model that develops software in small, workable parts.

6. Which model includes risk analysis as a major activity in every iteration?
7. What is the main goal of the Risk Analysis phase in the Spiral model?
8. In which model is a prototype often created to handle uncertainty?
9. Which model allows partial implementation of the system with user feedback?
10. What is the final phase in the Waterfall model?
11. Which model is most appropriate for large and high-risk software projects?
12. In the Incremental model, what is each completed section called?
13. What is the main limitation of the Waterfall model?

Answers to Objective Type Questions

1. Requirements definition
2. Waterfall model
3. Implementation and Unit Testing
4. Waterfall model
5. Incremental model
6. Spiral model
7. To identify and reduce potential project risks
8. Spiral model
9. Incremental model
10. Operation and Maintenance
11. Spiral model
12. Increment
13. It does not support changes once the process starts

Assignments

1. Explain the steps of the Waterfall model in your own words. Give one example for each phase.
2. Compare and contrast the Waterfall model and Incremental model. Highlight at least three key differences.
3. Draw and label the Spiral model. Explain how risks are handled in this model.
4. Discuss how the Incremental model helps in incorporating user feedback. Use a real-life example to support your answer.
5. What are the advantages and limitations of the Waterfall model? Explain briefly.
6. Write a short note on the importance of risk analysis in the Spiral model.
7. Imagine you are developing a mobile app for online food ordering. Which software development model would you choose and why?

Suggested Reading

1. Sommerville, Ian. "Software engineering (ed.)." *America: Pearson Education Inc* (2011).
2. Van Vliet, Hans, Hans Van Vliet, and J. C. Van Vliet. *Software engineering: principles and practice*. Vol. 13. Hoboken, NJ: John Wiley & Sons, 2008
3. Pressman, Roger S. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
4. Lawrence, Pfleeger Shari. *Software engineering: theory and practice*. Pearson Education India, 2006

Reference

1. <https://archive.nptel.ac.in/courses/106/105/106105182/>



Comparison of different life cycle models

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define the term Software Development Life Cycle.
- ◆ list the different types of SDLC models.
- ◆ identify the phases in the Waterfall Model.
- ◆ name any two advantages of the Agile Model.
- ◆ compare different SDLC models

Prerequisites

You might have already used different kinds of software applications like a mobile banking app, a food delivery app, or even your college's online portal. Each of these software applications was developed through a proper plan or step-by-step process. For example, a food delivery app might start by identifying user needs (like searching for restaurants), then move on to designing screens, writing the code, testing the app for errors, and finally releasing it to users.

These steps are not done randomly. Software engineers follow specific models called Software Development Life Cycle (SDLC) models, to guide how software is built. Different models are used for different situations. For instance:

- ◆ If the app requirements are fixed and clear from the start, the Waterfall model may be used.
- ◆ If the app needs to be updated often based on user feedback, the Agile model might be a better choice.
- ◆ If the app is large and risky (like an e-commerce platform), the Spiral model helps by focusing on risk analysis.

In this unit, you will learn how to compare these models based on their process style, flexibility, user involvement, and speed. Understanding this comparison will help you decide which model is best for different types of software projects.

Keywords

SDLC models, Waterfall model, Agile model, Spiral model, software development

Discussion

In the field of software engineering, it is essential to follow a systematic and structured approach to develop software applications. The Software Development Life Cycle (SDLC) provides such a framework by defining various stages through which software passes from initial idea to final delivery and maintenance. Over time, several SDLC models have been introduced to address different project needs, team structures, and risk factors.

This unit presents a detailed comparison of different software development life cycle models like waterfall, V-model, incremental, spiral, agile, DevOps and prototype models to help learners understand their principles, workflows, advantages, and limitations. Understanding these models is important for selecting the right development strategy for different software projects.

1.4.1 Waterfall Model

The Waterfall model is a linear and sequential approach where each phase of the development process is completed before moving to the next. It follows a strict order: Requirements gathering, System Design, Implementation, Testing, Deployment, and Maintenance. Each phase is fully completed before the next phase starts, and there's no going back to a previous phase once it's done. This model works best for projects where the requirements are well-understood and unlikely to change.

Advantages:

- ◆ **Simplicity:** The model is easy to understand and implement, making it ideal for small projects.
- ◆ **Clear Steps:** Each phase has specific deliverables, making it easy to track progress.
- ◆ **Simple to Track:** The project manager can easily measure progress due to the structured nature of the process.
- ◆ **Good for Small Projects:** Works well when the project scope is clear from



the outset and unlikely to change.

Disadvantages:

- ◆ **Inflexibility:** Changes are difficult to implement once the development process is underway. It's not suitable for projects with evolving requirements.
- ◆ **Late Testing:** Testing is done after the development phase, so errors are detected late in the process.
- ◆ **No Overlap:** Since each phase is done sequentially, the process can be slow.
- ◆ **Needs All Details at the Start:** The model assumes that all requirements must be fully defined upfront, which is not always possible in the case of complex projects.

1.4.2 V-Model (Verification and Validation)

The V-Model is an extension of the Waterfall model, emphasizing the importance of verification and validation. Each development phase has a corresponding testing phase directly associated with it, forming a "V" shape. The process starts with requirements, followed by design, implementation, and testing. The key difference is that as the development progresses, testing is planned simultaneously to ensure that each phase's output meets the desired specifications.

Advantages:

- ◆ **Early Testing:** Testing is planned early in the process, ensuring that errors are detected as soon as possible.
- ◆ **Clear and Structured:** Like Waterfall, the V-Model provides a structured approach to software development, making it easy to track progress.
- ◆ **Improved Product Quality:** Since testing and development are intertwined, it improves product quality.
- ◆ **Predictable:** It's easy to determine the outcome and timeline due to the structured phases.

Disadvantages:

- ◆ **Not Flexible:** It still lacks flexibility and is not ideal for projects with changing requirements.
- ◆ **Assumes Clear Requirements:** Like Waterfall, the V-Model assumes that all requirements are defined upfront.
- ◆ **Expensive Testing:** Since testing is an integral part of the development, the overall cost can increase, especially for large projects.
- ◆ **Not Ideal for Large Projects:** The complexity of handling multiple phases may become challenging in large-scale projects.

1.4.3 Incremental Model

The Incremental model breaks the project into smaller, manageable portions or "increments." Each increment adds new functionality to the system, and each increment is developed, tested, and integrated separately. The system is developed progressively, allowing for early releases and incremental improvements. This model works well when the project's requirements are well understood but can evolve over time.

Advantages:

- ◆ **Flexibility:** New requirements can be incorporated in later increments, making it suitable for projects where the full requirements aren't clear from the beginning.
- ◆ **Early Delivery:** A functional product is delivered early with each increment, allowing for faster deployment and user feedback.
- ◆ **Risk Management:** By developing increments, risks can be better managed and addressed at each stage.
- ◆ **Easier to Manage:** Since each increment is relatively small, it's easier to track progress and identify problems early on.

Disadvantages:

- ◆ **Integration Issues:** Integrating different increments can be complex and time-consuming, especially as the system grows.
- ◆ **Potential for Incomplete Features:** As each increment is delivered progressively, some parts of the system may remain incomplete or not fully functional until later.
- ◆ **Requires Consistent Feedback:** Continuous user feedback is necessary to ensure that each increment meets user needs and expectations.
- ◆ **Possible Overlap:** There can be overlap between increments, making it difficult to finalize features.

1.4.4 Spiral Model

The Spiral model is a risk-driven approach that combines iterative development with risk management. The project is developed in repeated cycles or "spirals," each of which involves four key steps: planning, risk analysis, engineering (design and development), and evaluation. After each cycle, feedback is gathered, and the product is refined. The model allows for flexibility and change at every stage and emphasizes continual risk assessment.

Advantages:

- ◆ **Risk Management:** Each spiral involves risk analysis, which allows teams to identify and address potential problems early in the process.
- ◆ **Flexibility:** New requirements and design changes can be incorporated into



subsequent spirals, making the model highly flexible.

- ◆ **Frequent Feedback:** The iterative nature ensures that the project is constantly refined based on feedback from stakeholders.
- ◆ **Suitable for Large Projects:** It's ideal for large, complex, or high-risk projects where requirements are uncertain.

Disadvantages:

- ◆ **Complexity:** The model can be difficult to manage due to its iterative nature and the need for constant risk analysis.
- ◆ **Costly:** Frequent evaluations and the complexity of managing spirals make it more expensive compared to simpler models.
- ◆ **Requires Expertise:** The risk management process requires skilled professionals who can properly assess and mitigate potential

1.4.5 Agile Model

The Agile model focuses on iterative development, collaboration, and flexibility. It breaks the project into small, manageable units called "sprints" (typically 1-4 weeks), with each sprint delivering a small piece of the product. The Agile methodology promotes frequent communication between developers and stakeholders, and the ability to adapt to changing requirements throughout the development process.

Advantages:

- ◆ **Highly Flexible:** Agile is highly adaptable and can respond to changes in requirements, making it perfect for projects where requirements are expected to evolve.
- ◆ **Early and Continuous Delivery:** Each sprint results in a working piece of software, allowing for frequent releases.
- ◆ **Customer Collaboration:** Stakeholders are involved throughout the development process, ensuring the product meets their needs.
- ◆ **Faster Development:** Agile emphasizes quick cycles and rapid delivery, leading to faster overall development.

Disadvantages:

- ◆ **Less Predictable:** Since requirements evolve during the process, the timeline and cost estimates can be unpredictable.
- ◆ **Requires High Customer Involvement:** Continuous feedback from customers and stakeholders is crucial, which may not always be feasible.
- ◆ **Hard to Scale:** Large teams or projects can struggle with Agile due to coordination and communication challenges across multiple sprints.

1.4.6 DevOps Model

DevOps is a continuous, integrated approach where development (Dev) and operations (Ops) teams work together throughout the entire software lifecycle. DevOps emphasizes automation, continuous integration, continuous delivery, and continuous testing. This approach allows for the rapid development and deployment of software and ensures that changes are continuously monitored and refined.

Advantages:

- ◆ Continuous Delivery: Software is deployed frequently, leading to faster product iterations and updates.
- ◆ Improved Collaboration: Development and operations teams work together, breaking down silos and improving communication.
- ◆ Faster Resolution of Issues: Continuous monitoring allows teams to detect and fix problems quickly.
- ◆ Higher Quality: Frequent testing and integration result in more stable and reliable software.

Disadvantages:

- ◆ Requires Automation: DevOps relies heavily on automation tools, which can be costly and require significant setup.
- ◆ Resource-Intensive: The infrastructure needed for continuous delivery and integration can be resource-intensive.
- ◆ Not Suitable for All Projects: Projects that don't require frequent updates or monitoring may not benefit from the DevOps approach.

1.4.7 Prototype Model

The Prototype model emphasizes creating early, working versions (prototypes) of the system to gather user feedback. Prototypes are refined iteratively based on feedback until the final system is developed. This model is particularly useful when requirements are unclear or when user interaction is critical to defining system features.

Advantages:

- ◆ Early Feedback: Prototypes allow users to interact with a tangible version of the system early on, helping to clarify requirements.
- ◆ User-Centric: The model ensures that the final system meets user needs by involving them throughout the development process.
- ◆ Useful for Unclear Requirements: It works well for projects where the requirements are not well-defined at the outset.

Disadvantages:



- ◆ **Unrealistic Expectations:** Users may expect the prototype to be the final product, which can lead to disappointment if the prototype is not fully functional.
- ◆ **Time-Consuming:** Building and refining multiple prototypes can be time-consuming and expensive.
- ◆ **Incomplete Design:** Prototypes often focus on user interface and basic functionality, leaving out important system aspects like security or performance.

1.4.8 Comparison of different models

Table 1.4.1 Comparison of different SDLC models

Model	Process Style	Flexibility	User Involvement	Delivery Speed	Best Suited For
Waterfall	Linear and Sequential	Low	Low	Slow	Small, clear projects
V-Model	Linear + Testing	Low	Low	Slow	Projects needing strong testing
Incremental	Step by step build	Moderate	Moderate	Medium	Projects with evolving needs
Spiral	Iterative + Risk Focus	High	High	Medium	Large, risky projects
Agile	Iterative and Flexible	Very High	Very High	Fast (in sprints)	Dynamic, changing requirements
DevOps	Continuous Delivery	Very High	High	Very Fast	Automation-focused environments
Prototype	Build Prototype First	High	High	Medium	Unclear user needs

Recap

- ◆ Waterfall Model
 - Linear, sequential process
 - Phases: Requirements → Design → Implementation → Testing → Deployment → Maintenance
 - Advantages: Simple, clear milestones, easy to manage
 - Disadvantages: Inflexible, late testing, assumes clear requirements
- ◆ V-Model
 - Extension of Waterfall
 - Each development phase has a corresponding testing phase
 - Advantages: Strong focus on testing, structured, predictable
 - Disadvantages: Rigid, expensive testing, not for large projects
- ◆ Incremental Model
 - Develops in increments or parts
 - Each increment = new functionality
 - Advantages: Flexible, early delivery, better risk management
 - Disadvantages: Integration issues, needs continuous feedback
- ◆ Spiral Model
 - Iterative + risk-focused
 - Steps: Planning → Risk Analysis → Engineering → Evaluation
 - Advantages: Risk management, flexibility, suitable for large projects
 - Disadvantages: Complex, costly, needs expertise
- ◆ Agile Model
 - Iterative, sprint-based (1–4 weeks)
 - Emphasizes collaboration and adaptability

- Advantages: Highly flexible, early delivery, user-focused
- Disadvantages: Scope creep, less predictable, hard to scale
- ◆ DevOps Model
 - Dev + Ops collaboration
 - Continuous integration, testing, delivery
 - Advantages: Faster delivery, better collaboration, improved quality
 - Disadvantages: Needs automation, cultural shift, resource-intensive
- ◆ Prototype Model
 - Early, working system prototypes
 - Refined based on user feedback
 - Advantages: Early feedback, user-centric, good for unclear requirements
 - Disadvantages: Unrealistic expectations, time-consuming, incomplete design

Objective Type Questions

1. Which software development model follows a linear and sequential approach?
2. What is the main focus of the V-Model in software development?
3. Which model allows for development in small, manageable parts called increments?
4. In which model is risk analysis a core component of every development cycle?
5. What do Agile sprints typically produce at the end of each cycle?
6. Which model combines development and operations teams for continuous integration and delivery?
7. What type of model is ideal for gathering early user feedback through interactive versions?
8. Which model assumes that all requirements are clearly defined from the start?

9. Which model is best suited for dynamic projects with frequently changing requirements?
10. Which model is known for involving users throughout the development through early system versions?
11. What is a major disadvantage of the V-Model?
12. Which model allows flexibility and is risk-driven, suitable for large and complex projects?
13. Which model relies heavily on automation and continuous monitoring?
14. What is the key benefit of using the Incremental Model?

Answers to Objective Type Questions

1. Waterfall Model
2. Verification and Validation
3. Incremental Model
4. Spiral Model
5. Working software/product increment
6. DevOps Model
7. Prototype Model
8. Waterfall Model
9. Agile Model
10. Prototype Model
11. Lack of flexibility
12. Spiral Model
13. DevOps Model
14. Early delivery of working software

Assignments

1. Explain the key stages of the Waterfall Model with a simple example.
2. What are the advantages and disadvantages of the Incremental Model?

3. Describe how the Spiral Model incorporates risk analysis in each phase.
4. Write a short note on Agile methodology and its importance in modern software development.
5. Compare the Waterfall Model and the V-Model with respect to testing activities.
6. Explain the steps involved in the Prototype Model and how it helps in requirement gathering.
7. What is DevOps? Discuss its key principles and benefits

Suggested Reading

1. Sommerville, Ian. "Software engineering (ed.)." *America: Pearson Education Inc* (2011).
2. Van Vliet, Hans, Hans Van Vliet, and J. C. Van Vliet. *Software engineering: principles and practice*. Vol. 13. Hoboken, NJ: John Wiley & Sons, 2008
3. Pressman, Roger S. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005.
4. Lawrence, Pfleeger Shari. *Software engineering: theory and practice*. Pearson Education India, 2006.

Reference

1. <https://archive.nptel.ac.in/courses/106/105/106105182/>

Software Project Management





Strategic Management of Software Projects

Learning Outcomes

- ♦ define the basic terms used in software project management.
- ♦ list the different types of management involved in SPM.
- ♦ describe how good communication helps in managing software teams.
- ♦ identify common problems in time and budget management.
- ♦ summarize why software maintenance is important after the project ends.

Prerequisites

Before moving to the concepts of Software Project Management (SPM), it's important to understand why SPM is needed in the first place. Managing a software project is not just about writing code, it involves organizing people, time, money, tools, and processes to ensure the project succeeds. Without proper management, software projects can easily go off track, face delays, or even fail entirely.

Let's take a simple real-world example. Imagine a team is building a mobile app that helps students find study groups at their university. If there is no clear plan, the designers might start working on the interface while the developers haven't finished setting up the backend. If the team doesn't communicate well, some features might be repeated or missed entirely. Or if the client suddenly wants to add new features, the team might not be prepared to adjust the timeline or budget. In all these cases, the lack of proper project management can lead to confusion, missed deadlines, extra costs, and a final product that doesn't meet the users' needs.

Software Project Management provides a structured way to avoid these problems. It helps in planning the project tasks clearly, assigning roles based on team members' strengths, estimating time and cost, and tracking progress effectively. A good project manager ensures that everyone knows what to do and when to do it, keeps the team focused, and helps solve problems before they grow too big. Just like an event organizer ensures everything runs smoothly during a school function, from food

to music to guest seating, a software project manager makes sure all parts of the software development process are working together in harmony.

By understanding the need for SPM, it becomes clear why it is a vital skill for anyone working in software development. It is not just about managing work but it's about ensuring the project is delivered successfully, on time, within budget, and with quality.

Keywords

Project Planning, Risk Management, Requirement Management, Change Management, Software Configuration Management, Time Management

Discussion

2.1.1 Introduction to Software Project Management (SPM)

Software Project Management, often called SPM, is the process of carefully planning and guiding software projects from start to finish. Think of it like organizing a big school event where you need to decide what tasks to do, who will do them, and when everything should be completed. In SPM, this means planning the software work, carrying out the tasks, watching the progress closely, and making sure everything stays on track. For example, if a team is creating a mobile app, the project manager will plan the features, assign roles to developers, check the progress regularly, and solve any problems that come up to finish the app on time.

This article will explain important topics in Software Project Management that help you understand how to manage software projects effectively. Learning these topics will help anyone who wants work and coordinate as part of a project team.

2.1.2 Types of Management in SPM (Software Project Management)

2.1.2.1 Conflict Management

Conflict Management is the process of recognizing and handling disagreements in a team so that they do not harm the project. In project management, conflicts often arise due to differences in opinions, work styles, or unclear roles. If not addressed, they can lead to delays, poor communication, and low team morale. However, well-managed conflict can actually lead to innovation and better decisions.

For example, suppose two team members disagree on the choice of database technology. A good project manager listens to both sides, involves the technical lead if needed, and helps them agree on a solution based on project goals and timelines. This

approach uses communication, negotiation, and decision-making to the core project management skills, to turn potential problems into productive discussions. It also improves collaboration and project outcomes.

2.1.2.2 Risk Management

Risk Management is the process of identifying possible problems before they happen, analyzing how likely they are, and planning how to avoid or reduce their impact. This is a key area in project planning and monitoring. Project managers use risk registers, impact analysis, and contingency plans to prepare for the unexpected.

Consider the case of a software project, a risk might be that the client changes the project scope halfway through. The project manager should plan for this by including time buffers in the schedule and being flexible with resource allocation. They also hold regular meetings to reassess risks and update the plan. By doing this, the team can avoid panic and stay on track even when issues arise.

2.1.2.3 Requirement Management

Requirement Management involves understanding what stakeholders want, documenting those needs clearly, and keeping track of any changes during the project lifecycle. This is important because unclear or changing requirements are one of the top causes of project failure. Project managers use tools like requirement traceability matrices and version control to manage this process.

When a software company is starting a mobile app project, during requirements collection, the client might want a simple registration form. Later, they request biometric login. A good project manager will assess how this change affects the budget and timeline, get approval from stakeholders, and update the documentation. Clear communication and version control ensure everyone stays informed and aligned. This shows the importance of scope management and change control.

2.1.2.4 Change Management

Change Management in project management refers to the structured approach of shifting or transforming a project, team, or organization from its current state to a desired future state. This includes managing changes in technology, processes, or team roles. It's not just about managing technical change, but also about helping people adapt. This aligns closely with stakeholder management and organizational behavior.

Suppose, if a software company decides to move from a traditional development model to Agile, change management ensures that developers are trained, clients are informed of the new process, and tools are updated. The project manager creates a change management plan that includes timelines, training schedules, and communication strategies. By managing resistance and ensuring a smooth transition, change management helps the project succeed in a dynamic environment.

2.1.2.5 Software Configuration Management

Software Configuration Management (SCM) is the discipline of managing and controlling changes in the software throughout the project. It includes version control,

tracking updates, and maintaining software integrity. This is critical for quality management, especially in larger teams or multi-phase projects.

In a team using Git, developers work on different modules. SCM ensures that when changes are merged, there are no conflicts or loss of data. It helps maintain baseline versions of the software, so if something goes wrong, the team can revert to a stable version. Configuration audits and status reporting are also part of this process. SCM supports project control and ensures consistent delivery of software builds.

2.1.2.6 Release Management

Release Management is the process of planning, scheduling, and controlling the build, testing, and deployment of software releases. It ensures that each release meets quality standards and is delivered without disrupting existing services. This is closely related to integration management and deployment planning.

For example, before launching a new feature in an e-commerce app, the team goes through staging, user acceptance testing (UAT), and a roll-out plan. The project manager coordinates with developers, testers, and support staff to make sure the release is smooth. Proper release management avoids downtime, ensures customer satisfaction, and keeps project milestones on schedule. It reflects the need for detailed release schedules, checklists, and rollback plans.

2.1.3 Key Factors in Software Project Management

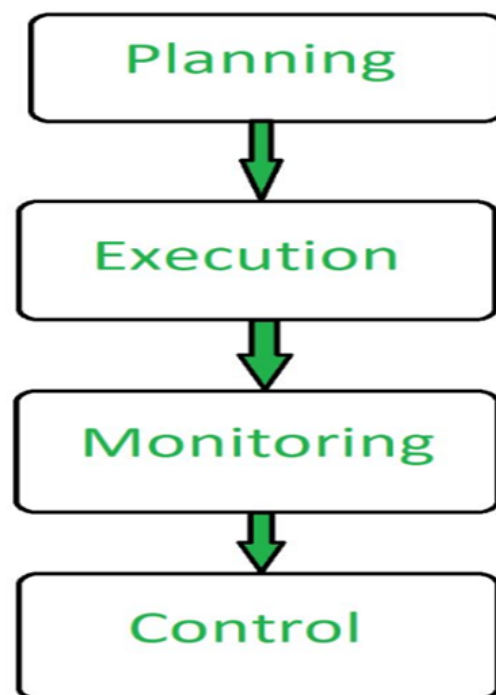


Fig. 2.1.1 Key Factors in Software Project Management

2.1.3.1 Planning

Planning is the very first and important step in managing a software project. The Software Project Manager (SPM) creates a detailed plan that shows how the whole

project will be done from start to finish. This plan includes what work needs to be done (the scope), what tools and people are needed (resources), when each task should be finished (timelines), and how the team will work together (techniques and strategy). The plan also explains how team members will communicate, how the software will be tested to find mistakes, and how it will be cared for after it is finished (maintenance). For example, if a team is building a simple app to help students track homework, the manager might decide how many programmers are needed, what programming language to use, when the app should be ready, and how they will test the app before giving it to students. The Software Project Manager's role is very helpful in making sure the plan is clear and realistic so the project runs smoothly.

2.1.3.2 Leading

A software project manager's job is to bring different people together to work as a team. This team might include engineers who build the software, strategists who plan the project, programmers who write the code, designers who create how it looks, and data scientists who analyze information. Leading this group means the manager must be really good at talking clearly and understanding others. They also need strong skills to guide and support the team so everyone can work well together. For example, if a programmer is stuck with a problem, the manager should listen carefully and help find a solution. To lead successfully, the project manager must follow important basic rules that are part of the project management process. These rules help keep the team organized, focused, and moving forward in the right direction. Without sticking to these principles, leading a team can become very difficult.

2.1.3.3 Execution

Execution is the stage where the actual work on the software project happens. Here, the person responsible for managing the project, often called the project manager, plays a very important role. If this person understands Software Project Management (SPM) or Agile methods well, they make sure that every step of the project is done properly and on time. For example, if a team is building a simple app, the manager will check if the design is ready before the programmers start coding. They will keep track of how much work is finished, watch how the team members are working together, and create reports to show the progress to everyone involved. This helps in solving problems quickly if something is not going as planned, like if a coder is stuck or if a feature is delayed. Overall, execution ensures the project moves forward smoothly and reaches its goals step by step.

2.1.3.4 Time Management

Managing your time well is very important to finish tasks on schedule. This can be especially hard when working on software projects because the plan often changes as the project goes on. For example, if you are building an app and halfway through, the client wants to add new features, the original timeline may no longer fit. Because of these kinds of unexpected changes, software project managers need to be very good at handling risks and preparing for emergencies. This means they should always be ready to deal with problems or delays that might come up. Being able to reduce risks and manage any sudden issues is a key idea in how software project management works.

Just like a captain steering a ship through rough weather, a project manager keeps the project on track even when things don't go as planned.

2.1.3.5 Budget

Software Project Managers, just like regular project managers, have the important job of creating a budget for their projects. This means they have to plan how much money will be needed for different parts of the project, like buying software tools, paying team members, or getting equipment. Once the budget is set, they must try their best to stick to it, making sure they don't spend more than planned. For example, if a manager budgets \$1,000 for software licenses but finds a cheaper option for \$800, they might use the extra \$200 for another part of the project. Managing the budget well helps avoid running out of money halfway through the project, which could cause delays or force the team to stop working. In short, learning how to handle the project's money carefully is a key skill that helps keep the project on track and successful.

2.1.3.6 Maintenance

In software project management, maintenance means taking care of the software even after it has been built. One very important part of this is testing the software regularly throughout the project. By testing often, the team can find problems or bugs early before they cause bigger issues. For example, imagine building a mobile app. If the app crashes when a user tries to open it, testing helps catch this early so the developers can fix it quickly.

The software project manager plays a key role here. They make sure that the software is checked carefully many times and any problems are fixed right away. They also make sure the software matches what the client really needs, like adding a new feature the client asks for or changing something that isn't working well. This helps the project stay on the right path and finish successfully.

So, by focusing on ongoing testing and improvements, software project management helps deliver a better, more reliable product that works well for the client.

2.1.4 Drawbacks of Software Project Management

If a software project manager does not have enough experience or proper knowledge, many problems can happen during the project. One big problem is the loss of resources, which means wasting money, time, or equipment. For example, if a manager does not plan the work well, the team might spend too much time on tasks that are not important, causing delays and extra costs. Another common difficulty is keeping the project on schedule. Sometimes, the project may take longer than expected because tasks are not organized properly or unexpected issues arise. Imagine a team trying to build a new app, but the manager does not set clear deadlines, so the app gets delayed by weeks.

Data protection is also a major concern. If the project manager doesn't enforce strong security rules, sensitive information like user details or company secrets might be accidentally shared or stolen. This can cause big problems, including legal issues or loss of customer trust. Lastly, software projects involve many people such as developers, engineers, and stakeholders who must work together. If the project manager cannot

manage these relationships well, conflicts may happen. For example, developers might disagree with what the stakeholders want, leading to arguments and slowing down progress.

In summary, without the right skills, a project manager can face many challenges like wasting resources, trouble with timing, risks to data security, conflicts among team members, etc.

2.1.4.1 High Cost

When you use software project management methods, you often have to spend a lot of money. For example, you might need to buy special tools or software to help manage the project, like apps for tracking tasks or communication platforms. These tools can be expensive to purchase and set up. Also, your team will need time to learn how to use these new tools properly, which means you might have to spend money on training sessions. Sometimes, you may need to hire experts who know a lot about certain parts of the project to help the team, and that can add to the cost. On top of this, people involved in the project, like customers or managers, might ask to add new features that were not planned at the start. All these things together can make the project cost much more than expected and take longer to finish. For example, if you start building a simple app but later need to add extra functions because users request them, it will increase your expenses and delay the project.

2.1.4.2 Increased Complexity

Managing a software project involves many steps and can be quite complicated. Sometimes, experts try to make things too detailed or difficult, which can confuse the team and slow down the work. For example, if a project manager uses very technical language or insists on strict rules, it can make the workplace stressful and hard to understand. When a project is big and has many parts, it becomes even harder to finish on time. This is especially true if there isn't a special team working only on that project. If team members have other regular jobs to do besides the project, they might not be able to focus well, which makes the project more complicated. For example, if a software developer is also responsible for daily maintenance tasks, they may not have enough time to fully concentrate on the new project. All these factors add up and make managing software projects more challenging.

2.1.4.3 Communication Overhead

When you hire new people to join your software project management team, these new recruits bring fresh communication styles with them. This means there will be a continuous flow of messages, emails, meetings, and discussions. However, not all of this communication will always fit well with the way your company usually works or communicates. For example, if your company prefers quick, informal chats but a new team member likes long, detailed emails, this can cause confusion or slow things down.

Because of this, it is generally better to keep your team as small as possible. A smaller team means fewer people to talk to and fewer chances for messages to get mixed up or lost. For example, in a small group of five people, everyone can easily stay in touch and know what's going on. But when the team grows to 20 or more, the number of

communication paths increases a lot, making it harder to coordinate.

If a project requires a big team, it becomes very important to have software project managers who are good at communicating with many different kinds of people. These managers need to make sure that everyone understands their tasks and that information flows smoothly. Imagine a project manager acting like a bridge, connecting people who speak different “languages” of work style, experience, or background, so the whole team can work well together despite its size.

2.1.4.4 Issue related to Originality

Sometimes, software project managers do not allow much room for creativity. For example, a team leader might focus too much on following strict rules and deadlines, leaving little chance for team members to come up with new ideas. Imagine a group of programmers who are told they must finish writing code by a certain date and follow a fixed set of instructions without trying anything different. This can stop them from thinking of better or smarter ways to solve problems. Creativity is important because it can lead to improvements that make the software work faster or easier for users. In managing software projects, it is important to know when to let people be creative and when to follow a clear plan. If there is no project manager, the team might finish the work faster by just coding quickly. But having a skilled project manager can actually help the team find new solutions and reach their goals more successfully and completely. For example, a good manager might give the team some time to experiment with new tools, which could make the final software better. So, balancing creativity with management is key to success in software projects.

Recap

- ◆ Software Project Management (SPM) is the process of planning, organizing, and tracking software projects from start to finish.
- ◆ A software project manager is like the leader of a big event, making sure tasks are done on time and the team works well together.
- ◆ Conflict Management involves solving team disagreements early so they don't harm the project; it helps improve communication and teamwork.
- ◆ A good example of conflict management is when a manager helps two developers agree on a database by focusing on project needs.
- ◆ Risk Management means identifying possible problems in advance and creating backup plans to avoid or reduce their impact.
- ◆ Requirement Management involves collecting, documenting, and updating what the client wants throughout the project.
- ◆ An example is when a client first asks for a basic login and later requests biometric login; the manager updates the plan and budget accordingly.

- ◆ Change Management helps teams and organizations smoothly adapt to new processes, tools, or technologies during a project.
- ◆ For example, when switching from traditional development to Agile, a project manager ensures the team is trained and ready.
- ◆ Software Configuration Management (SCM) keeps track of software versions and changes, ensuring quality and consistency in the project.
- ◆ Tools like Git help manage SCM by allowing teams to merge updates safely and go back to previous versions if needed.
- ◆ Release Management ensures software is tested and delivered properly without causing service disruptions.
- ◆ A real-world case is testing a new feature in an app and then carefully rolling it out to users with backup plans in place.
- ◆ Planning is the first and most important part of SPM, outlining what will be done, who will do it, and when it will be completed.
- ◆ In the planning stage, the manager decides on tools, people, timelines, and testing strategies, like for a student homework app.
- ◆ Leading means guiding a diverse team of developers, designers, and others by using good communication and teamwork.
- ◆ Execution is the stage where the actual work happens, and the manager ensures tasks are completed correctly and on time.
- ◆ Time Management is essential because software projects often change; managers must handle delays and adjust timelines.
- ◆ SPM has drawbacks like high cost, complexity, communication problems, and limited creativity if not managed properly.

Objective Type Questions

1. What is the process of guiding a software project from start to finish called?
2. Which type of management deals with solving team disagreements effectively?
3. Which management area focuses on identifying and reducing future problems in a project?
4. What is the process of understanding and tracking what stakeholders

need during a project?

5. What structured approach helps manage transitions in a project or organization?
6. Which process handles version control and change tracking in software projects?
7. What ensures smooth planning and delivery of software updates or new versions?
8. What is the first and most important step in software project management?
9. What skill does a manager use to bring different people together to work as a team?
10. What phase involves the actual performance of tasks in a software project?
11. Which management skill helps deal with delays due to unexpected client requests?
12. What is used to control the amount of money spent in a project?
13. What process continues even after the software has been deployed?
14. Poor management of----- can lead to loss of time, money or equipment.
15. What issue occurs when managers or tools make things more difficult than necessary?
16. What communication-related problem increases as team size grows?
17. What problem arises when strict rules stop team members from being innovative?
18. Which project management area involves identifying tasks, resources, and timelines?
19. Which type of management uses tools like risk registers and contingency plans?
20. What tool is used in requirement management to track changes to client needs?

Answers to Objective Type Questions

1. SPM
2. Conflict Management
3. Risk
4. Requirement
5. Change Management
6. Configuration Management
7. Release Management
8. Planning
9. Leading
10. Execution
11. Time
12. Budget
13. Maintenance
14. Resources
15. Complexity
16. Communication complexity (Overhead)
17. lack of flexibility or stifled creativity
18. Planning
19. Risk
20. Requirements tracking tool

Assignments

1. Explain the importance of Software Project Management (SPM) in successfully completing a software project with the help of a real-life example.
2. Describe the various types of management in SPM, such as conflict management, risk management, requirement management, change management, software configuration management, and release management.
3. What are the key components involved in planning a software project, and why is planning considered the foundation of successful project management?
4. What is the role of a Software Project Manager in leading a team, and how does effective leadership influence the success of a software project?

5. Discuss the significance of time management in SPM and explain how delays and changes in requirements are managed during a project.
6. Define risk management in the context of software projects and explain how early identification and handling of risks contribute to project success.
7. List and explain the major drawbacks or limitations of Software Project Management, such as high cost, complexity, communication overhead, and lack of creativity.
8. Why is software maintenance important after the project is delivered, and how does it help in ensuring long-term software performance and client satisfaction?

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.



PROJECT SIZE ESTIMATION TECHNIQUES

Learning Outcomes

- ◆ describe the aspect of project size estimation.
- ◆ list various project size estimation techniques.
- ◆ identify the importance of project size estimation.
- ◆ name the roles involved in estimating project size.
- ◆ explain how project size estimation supports planning and budgeting.
- ◆ describe different project estimation methods like Function Points and Use Case Points.

Prerequisites

In modern software engineering, accurately estimating the size of a software project is essential for successful project planning and execution. Without a clear understanding of the project's size, teams may face budget overruns, missed deadlines, or inefficient use of resources. Project size estimation helps in setting realistic goals, allocating appropriate manpower, and predicting overall effort and cost. For example, consider a government e-governance portal that is expected to handle thousands of daily user interactions, secure document uploads, and real-time data access across multiple departments. Estimating the size of such a complex project before development allows stakeholders to anticipate technical challenges, determine staffing needs, and plan the delivery schedule. Without size estimation, such large-scale projects risk failure due to underestimation of resources and time. Therefore, project size estimation acts as the cornerstone of informed decision-making and project success.

Keywords

Project Estimation, Software Engineering, Function Points, Use Case Points, COCOMO, Parametric Estimation, Wideband Delphi, Monte Carlo Simulation

Discussion

Project size estimation refers to the process of evaluating and defining the overall scope and the range of resources necessary to successfully complete a project. This involves a comprehensive assessment of multiple project components to determine the estimated effort, time, cost, and human or technical resources required. The goal is to understand how large or complex a project is before initiating development or execution. Accurate size estimation plays a critical role in ensuring effective planning, efficient resource allocation, and smooth project management. It enables stakeholders to set realistic timelines, budgets, and expectations, which are essential for the successful delivery of the project.

2.2.1 Introduction

In the dynamic and ever-evolving field of software engineering, accurately estimating the size of a project plays a crucial role in determining its overall success. Project size estimation is more than just predicting the number of lines of code or features that serves as the foundation for planning, budgeting, and scheduling. By understanding the scope and complexity of a software project early on, teams can more reliably forecast the resources, effort, time, and costs required. This foresight allows for better decision-making, realistic timelines, and effective resource allocation. Project Size Estimation Techniques, therefore, are essential tools that enable project managers and developers to assess feasibility and prepare for potential challenges before the development process begins. By incorporating these techniques, software projects are more likely to be delivered on time, within budget, and aligned with client expectations, ensuring efficient management and successful execution from the outset.

2.2.2 Importance of project size estimation

Accurate project size estimation plays a vital role in effective project management, serving as the foundation for several critical planning activities. First and foremost, it enables precise financial planning by helping project managers forecast the budget required for the entire lifecycle of the project. For example, in a software development project, estimating the size of the project allows the team to anticipate licensing costs, developer salaries, hardware, and other expenses, thus reducing the risk of financial shortfalls. Moreover, it supports resource planning by identifying the human, technical, and material resources necessary to complete the project efficiently. This ensures that the right resources are available at the right time. Project size estimation also contributes significantly to timeline creation, allowing managers to set realistic deadlines and

milestones. for example, if a mobile app is estimated to require six months of work, breaking it down into development, testing, and deployment phases becomes easier and more accurate.

In addition, size estimation plays a key role in risk identification, as it helps uncover potential challenges or constraints that could impact the project's progress. A larger project might involve more integration points, increasing the chance of technical issues or delays. It also aids in detailed project planning, ensuring that all components such as design, coding, testing, and documentation are accounted for and scheduled properly. Finally, it is crucial for planning quality assurance activities, as knowing the scope and complexity of the project helps determine the level of testing and review required to meet quality standards. In summary, project size estimation is not just a preliminary step, it is a strategic necessity that directly influences the success, efficiency, and reliability of a project.

2.2.3 Responsibility of Estimating Project Size

Estimating the size of a project is a collaborative effort involving several key roles, each contributing unique insights and expertise to ensure accuracy and realism in planning. The Project Manager plays a central role in coordinating the entire estimation process, ensuring that all perspectives are considered and that the estimates align with the overall project goals. Subject Matter Experts (SMEs) contribute their in-depth knowledge of specific domains, for example, a healthcare SME might estimate data-handling requirements in a hospital management system. Business Analysts help translate client needs into well-documented requirements, clarifying scope and objectives; for example, they may break down user stories and functional requirements for a mobile banking app. Technical Leads evaluate the technical components such as system architecture, integration points, and testing strategies. For example, in an e-commerce website, they would estimate the time and complexity involved in integrating payment gateways or managing server load. Developers provide detailed task-level estimates based on their understanding of the work they will carry out, such as writing backend APIs or implementing user interfaces. Financial Analysts assess cost-related factors such as developer salaries, third-party services, or equipment needs, producing financial projections for the project. Meanwhile, Risk Managers identify potential threats like resource shortages or delays, that could affect the project timeline or scope, incorporating buffer time and resources into the estimate. Lastly, Clients also play a vital role by providing information on business constraints, budget limits, and expectations, which shape the estimation process. For example, if a client requires a system to be launched within three months with limited resources, estimators must adjust their approach accordingly. Together, these roles ensure that project size is estimated as accurately and comprehensively as possible.

2.2.4 Methods of Project Estimation

Estimating the time, cost, and resources for a project is a key step in successful project planning. Since every project is different in size, complexity, and clarity of requirements, project managers use various estimation methods to suit specific needs. These methods help in predicting how much effort will be required and in identifying

possible risks early on. Some methods depend on expert insights, while others use data from past projects or mathematical models. Choosing the right estimation technique depends on how much information is available, how detailed the planning needs to be, and the experience of the team. Understanding the strengths of each method helps in making better decisions, reducing uncertainty, and improving the chances of completing the project on time and within budget.

2.2.4.1 Expert Judgment

Expert Judgment is a project estimation technique that relies on the insights and experience of professionals who are well-versed in the relevant domain. When detailed project information is scarce or unavailable, such as in the early planning phase, this method becomes especially valuable. A panel of experts evaluates the potential size and scope of the project based on similar past projects and their professional intuition. For example, if a new mobile application is being proposed but detailed requirements are not yet documented, experienced software engineers and project managers might be consulted to provide a rough estimate of the development time and resources needed, drawing on their knowledge of comparable app development efforts.

2.2.4.2 Analogous estimation

Analogous estimation is a project sizing technique that relies on comparisons with past, similar projects to predict the effort, cost, or time required for a current one. It is particularly useful when detailed information about the new project is limited, but reliable historical data from completed projects is available. For example, if a software development team previously built a mobile application for a retail business in six months, they might use that experience to estimate the timeline for a new, similar app for a different retail client. By drawing parallels between the two projects in terms of scope, complexity, and resources, the team can make a reasonably accurate forecast. This method helps in quick decision-making, especially during the early planning stages.

2.2.4.3 Bottom-up Estimation

Bottom-up Estimation is a detailed and structured approach used in project management where the entire project is broken down into smaller, more manageable components or tasks. Each of these individual tasks is then estimated separately in terms of the resources, time, and effort required. Once all the task-level estimates are completed, they are combined or aggregated to form the total estimate for the entire project. For example, consider a software development project. Instead of estimating the cost and timeline for the whole software at once, the project manager would first divide it into modules like user interface design, database setup, backend logic, and testing. Each of these modules is then analyzed, and their specific requirements are estimated independently. By summing these individual estimates, a more accurate and realistic overall project estimate is obtained. This method helps in improving accuracy and allows for better tracking and management of project progress.

2.2.4.4 Three-point estimation

Three-point estimation is a method used to estimate the size, time, or cost of a project by considering three different scenarios. Instead of giving just one guess, you think about three possibilities:

1. Optimistic estimate – This is the best-case scenario. It answers the question: What if everything goes perfectly?
2. Pessimistic estimate – This is the worst-case scenario. It considers delays or problems: What if many things go wrong?
3. Most likely estimate – This is the most realistic scenario, based on experience and common situations.

Once you have these three numbers, you can calculate an average expected value. A common way to do this is by using the PERT formula:

$$\text{Expected Value} = (\text{Optimistic} + 4 \times \text{Most Likely} + \text{Pessimistic}) / 6$$

This formula gives more weight to the "most likely" case because that's usually the most reliable estimate.

Example:

Imagine you are estimating how long it will take to write a report.

- ◆ If everything goes well, you might finish in 2 days (optimistic).
- ◆ If you face interruptions or delays, it might take 6 days (pessimistic).
- ◆ Based on past experience, you think it will probably take 4 days (most likely).

Using the PERT formula:

$$(2 + 4 \times 4 + 6) / 6 = (2 + 16 + 6) / 6 = 24 / 6 = 4 \text{ days}$$

So, the expected time to finish the report is 4 days.

This technique helps make better decisions by considering different outcomes rather than relying on a single guess.

2.2.4.5 Function Points

The Function Points method is a way to estimate the size of a software project by looking at what the software is supposed to do. Instead of measuring lines of code, this technique focuses on the functions or features that the software will provide to users. These functions include things like how many inputs the system takes (such as filling out a form), how many outputs it gives (like a report or result), how many inquiries it handles (for example, searching for a product), and how many files or databases it uses or updates. Each of these components is counted and given a weight depending on its complexity. Then, all the points are added up to estimate the size of the project. For example, if you're building a library management system, you would count functions like adding a new book (input), generating a list of overdue books (output), searching

for a book by title (inquiry), and maintaining the list of users and books (files). This method gives a more realistic picture of the project's size because it focuses on what the software will actually do for its users. Detailed discussion will be done in the next session.

2.2.4.6 Use Case Points

Use Case Points is a method used to estimate the size and effort needed for a software project. It works by looking at how many use cases the software will have. A use case is a specific way that a user interacts with the system to do something useful. For example, in an online shopping app, a use case might be "place an order" or "track a delivery." This method also considers how complex each use case is, some might be simple, like "log in," while others could be more complicated, like "generate a monthly sales report." It also looks at how many people or systems (called actors) interact with the software. For example, in a school management system, actors might include students, teachers, and admin staff. By counting all these use cases, their complexity, and the number of actors, you can estimate how big the project will be and how much work it might take to complete.

2.2.4.7 Parametric Estimation

Parametric estimation is a method used to estimate the size or effort needed for a project by using mathematical formulas. These formulas are based on specific details of the project (called parameters) and past data from similar projects. This approach helps make the estimation more accurate and reliable.

For example, imagine a construction company that wants to estimate how much it will cost to build a new house. If they know from past experience that building one square foot of a house usually costs \$150, they can simply multiply that number by the total square footage of the new house. So, if the new house is 2,000 square feet, the estimated cost would be $\$150 \times 2,000 = \$300,000$.

Similarly, in software development, if a team knows from previous projects that writing 1,000 lines of code usually takes 50 hours, and the new project is expected to have 10,000 lines of code, they can estimate that it will take about 500 hours.

In simple terms, parametric estimation uses known rates and past experience to predict how big, costly, or time-consuming a new project will be.

2.2.4.8 COCOMO

COCOMO, which stands for Constructive Cost Model, is a method used to estimate how much time, effort, and money will be needed to complete a software development project. It works like a formula or algorithm that uses different factors to make this estimate. For example, it looks at the size of the software, how complex it is, and the experience level of the development team. Just like a builder might estimate how long it will take to build a house based on its size and materials, software engineers use COCOMO to plan their work. Suppose a team is developing a simple mobile app with basic features. COCOMO can help them figure out how many people they'll need and how long it will take. If the project is bigger, like creating a full banking system,

COCOMO will calculate much higher time and cost based on the project's complexity. This helps project managers plan resources and budgets more accurately.

2.2.4.9 Wideband Delphi

Wideband Delphi is a method used to make better and more accurate estimates, especially when planning the size or effort of a project. It works by bringing together several experts who first give their estimates anonymously. This helps prevent anyone from being influenced by others. After collecting these estimates, the group meets to discuss the differences and share their reasoning in a respectful and cooperative way. Then, each expert updates their estimate based on what they learned during the discussion.

For example, imagine a software company wants to estimate how long it will take to build a new mobile app. Instead of asking just one person, they invite five experienced developers. Each developer writes down their time estimate without seeing what the others wrote. Then, they meet and talk about why they chose their numbers. One developer might say, "I estimated more time because I think the login feature will be complex," while another might reply, "I estimated less because we can reuse code from another project." After this conversation, they each revise their estimate, and the final numbers are more balanced and reliable.

This method is useful because it combines the wisdom of different experts while avoiding pressure or bias, leading to more thoughtful and realistic planning.

2.2.4.10 Monte Carlo Simulation

Monte Carlo Simulation is a useful technique for estimating the size of a project and identifying possible risks, especially when the project is complex and full of uncertainties. It works by using statistics and random sampling to simulate different possible outcomes. For example, if you are building a new software application and you're not sure how long each stage will take, you can use Monte Carlo Simulation to run thousands of different scenarios using a range of possible times for each task. This helps you see the chances of finishing the project on time, early, or late. It gives a better understanding of the risks involved and helps in making smarter decisions. This method is similar to rolling dice many times to see all the possible results and understanding which ones are more likely to happen.

Every method or technique has its own advantages and disadvantages. Some methods work better in certain situations, while others may not be as effective. Choosing the right technique depends on several important factors. For example, if the project is very complex, a more advanced method might be needed. If there is only a small amount of data available, a simpler technique may be more suitable. Also, the skills and experience of the team play a big role. If the team is experienced in using machine learning, they might choose a machine learning-based method. But if they are more comfortable with basic statistics, they may use a traditional statistical approach. In short, the best method is the one that fits the project's needs, the amount of data, and the team's strengths.

2.2.5 Estimating the Size of the Software

Estimating how big a software project will be is an important step in managing any software development process. This step is called software size estimation, and it plays a major role in planning the project. When a project manager knows how large or complex the software is likely to be, it becomes easier to figure out how much time, effort, and money will be needed to complete the work. For example, if you're building a simple mobile calculator app, it will obviously take less time and fewer people to build than a large online shopping website like Amazon. To make such predictions, project managers use different ways to measure the size of a project. These methods can include counting the number of features, how many lines of code might be needed, or how much functionality the software is expected to provide. Just like estimating how long it would take to build a house based on the number of rooms, estimating software size helps in planning resources better and avoiding surprises later.

2.2.5.1 Lines of Code (LOC)

Lines of Code (LOC) is a simple way to measure the size of a software project by counting the number of lines in its source code. There are different types of LOC used in software estimation:

- ◆ KLOC: This stands for "Thousand Lines of Code". For example, if a program has 5,000 lines, it is referred to as 5 KLOC.
- ◆ NLOC: This means "Non-comment Lines of Code", which excludes comments and only counts the actual code that performs functions. For example, if a file has 100 lines but 20 are comments, the NLOC would be 80.
- ◆ KDSI: This stands for "Thousands of Delivered Source Instructions". It focuses on the actual instructions delivered in the final product.

To estimate the size of a new software project, experts often compare it with older, similar systems. This helps them figure out how large each part might be, and then they add all the parts together to get the full size. However, estimating LOC just by reading the problem or requirements is not easy. Accurate numbers can only be known after the code has been written. This makes LOC less useful for early project planning, where decisions must be made before any code is written.

Also, two programs with the same number of lines may not be equally difficult to create. For example, a file with complex logic such as advanced math functions or many conditions that takes more effort to write than a file with simple steps like printing a message. So, LOC doesn't always show how hard a task is.

Different programmers write code in different ways. An experienced developer may write clean, efficient code using fewer lines, while a beginner might need more lines to do the same job. So, LOC can vary widely between individuals.

Advantages of LOC

- ◆ LOC is widely accepted and used in popular estimation models like COCOMO.



- ◆ It matches how developers think about code, making it easier for them to understand and relate to.
- ◆ It is simple to measure once the project is complete.
- ◆ It gives a clear, numeric way to represent software size.
- ◆ People around the world use and accept it.
- ◆ Easy to apply, especially after development.

Disadvantages of LOC

- ◆ Different programming languages use different amounts of code to do the same thing. For example, a task that takes 10 lines in Java might only take 3 lines in Python.
- ◆ There is no universal standard to define or calculate LOC.
- ◆ It's hard to estimate LOC in the early stages of a project, when planning and budgeting decisions are made.
- ◆ LOC does not work well when comparing projects developed in different languages or on different platforms, as the line count may not reflect the actual effort or complexity.

2.2.5.2 Number of Entities in ER Diagram

An Entity-Relationship (ER) diagram gives a still or non-changing picture of a project. It shows the important things (called entities) in the system and how they are related to each other. The number of entities shown in an ER diagram helps us guess how big or complex a project will be. A large project will usually have many entities. This is because each entity usually needs its own class or structure in the program, and more classes mean more code to write.

For example, in a simple library management system, you may have entities like Book, Student, Librarian, and Issue Record. But in a hospital management system, there could be more entities like Patient, Doctor, Nurse, Room, Treatment, Medicine, and so on. So, the hospital system would have more entities, and therefore require more programming work.

Advantages:

- ◆ You can get an idea of how big the project is early in the planning stage, even before writing any code. For example, if your ER diagram has 15 entities, you already know it might take more time and resources compared to a project with just 5 entities.
- ◆ The number of entities is not tied to the technology or programming language you are using. Whether you are coding in Java, Python, or C++, the entities stay the same.

Disadvantages:

- ◆ There are no fixed rules for how much each entity adds to the overall size. for example, the Doctor entity might need more details and features than the Room entity, so not all entities are equal in complexity.
- ◆ Like Function Point Analysis (FPA), this method is not commonly used by itself to calculate the full cost of a project. So, to make it more useful, it often has to be changed into Lines of Code (LOC) to estimate the actual effort and cost.

2.2.5.3 Total Number of Processes in DFD

A Data Flow Diagram (DFD) is a tool used to show how a software system works from a functional point of view. It helps us understand the key tasks or operations (called processes) that the software performs and how data moves between these processes. For example, in a library management system, processes like “Issue Book,” “Return Book,” and “Search Catalog” can be shown in a DFD. Each of these processes interacts with data, such as user details or book records.

The number of these processes shown in the DFD can be used to guess the overall size of the software. To do this, software developers look at similar processes they have worked on before. For example, if they have already developed a "User Login" process in another project, they can estimate how big or complex a similar process will be in the current software. They do this for each process in the DFD. After estimating the size of all the processes, they add them up to get the total estimated size of the software.

Advantages:

- ◆ The method works no matter which programming language is used. For example, whether you use Java or Python, the DFD will still help estimate the size based on functions.
- ◆ Big processes can be broken into smaller ones. for example, a process called “Manage User Account” can be split into “Create Account,” “Update Account,” and “Delete Account.” This makes the estimation more detailed and accurate.

Disadvantages:

- ◆ To estimate the size correctly, you need to study other similar processes. This can take extra time and effort. For example, you might need to look at several past projects to find the right comparisons.
- ◆ Not all software projects need a DFD. Some simple or small projects might not use this method, so this estimation approach would not apply in those cases.

2.2.5.4 Function Point Analysis

Function Point Analysis (FPA) is a method used to measure the size of a software project by counting the number and types of functions it performs. This method helps estimate the amount of work required to develop the software. It doesn't depend on the



programming language used, which makes it useful for comparing different projects. The process involves several steps to calculate the final value called the Function Point Count (FPC). Let's look at each step with simple explanations and examples.

1. Count the number of functions of each proposed type

In this step, we identify and count the number of functions the software will perform. These functions are divided into five categories:

- ◆ External Inputs: These are the ways data enters the system. For example, filling out a form on a website or logging in using a username and password.
- ◆ External Outputs: These refer to data leaving the system, like generating a report or receiving a confirmation email.
- ◆ External Inquiries: These are requests for information from the system that don't change any data, such as searching for products in an online store.
- ◆ Internal Files: These are logical files that are stored and used within the system. For example, a database table that keeps customer information.
- ◆ External Interface Files: These are files used by other systems that our system accesses. for example, our software might read data from another company's database.

2. Compute the Unadjusted Function Points (UFP)

Next, we assess how complex each function is. Each function type (input, output, inquiry, etc.) can be simple, average, or complex, based on how difficult it is to design or implement. A weighting factor is assigned based on this complexity. We multiply the number of functions in each category by the relevant weighting factor. For example, if you have 3 complex external inputs, and the weight for a complex input is 6, you get $3 \times 6 = 18$ points from that category. We do this for all categories and add them up to get the Unadjusted Function Points (UFP).

Here's a quick look at the weights: see table 2.2.1

Table 2.2.1 Function type

Function Type	Simple	Complex
External Inputs	3	6
External Outputs	4	7
External Inquiries	3	6
Internal Logical Files	7	15
External Interface Files	5	10

3. Find the Total Degree of Influence (TDI)

This step considers 14 general characteristics that affect the project's complexity. Each characteristic is rated from 0 (no influence) to 5 (strong influence). Some of these characteristics include:

- ◆ Data Communications: Does the system communicate with other systems?
- ◆ Performance: Are there special performance requirements?
- ◆ Online Data Entry: Can users input data directly into the system?

Other characteristics include Transaction Rate, End-User Efficiency, Reusability and more. For example, if your software is used by many users at the same time, the "Transaction Rate" score will be high. You add the scores of all these 14 characteristics to get the Total Degree of Influence (TDI), which will be a value between 0 and 70.

4. Compute the Value Adjustment Factor (VAF)

Once you have the TDI, you use a formula to find the Value Adjustment Factor (VAF). The formula is:

$$\text{VAF} = (\text{TDI} \times 0.01) + 0.65$$

So, if your TDI is 30, your VAF would be $(30 \times 0.01) + 0.65 = 0.95$.

5. Find the Function Point Count (FPC)

Finally, we calculate the actual Function Point Count using this formula:

$$\text{FPC} = \text{UFP} \times \text{VAF}$$

For example, if your UFP is 120 and your VAF is 0.95, then the FPC is $120 \times 0.95 = 114$. This final number gives you a standardized size estimate for your software project.

Advantages

- ◆ This method can be used early in the project, even before coding starts. For example, during the planning phase, you can use function points to estimate time and budget.
- ◆ It is independent of the programming language, which means you can use it whether your project is in Java, Python, or any other language.
- ◆ It allows easy comparison between different projects, even if they use different technologies like different databases or frameworks.

Disadvantages

- ◆ Function Point Analysis doesn't work well for real-time or embedded systems. For example, systems controlling medical devices or machines may not fit into this model easily.
- ◆ Many estimation models, like COCOMO, use Lines of Code (LOC). Since FPC is different from LOC, you would need to convert function points into LOC, which can be tricky and may reduce accuracy.



2.2.6 Proper time for starting Estimation

Estimating the size and effort of a project should not be done just once. It needs to happen at different important stages of the project to make sure the estimates stay accurate and useful. Let's look at each stage with simple explanations and examples.

2.2.6.1 Project Initiation

At the very beginning of the project, basic estimates are made to decide whether the project is possible and worth doing. For example, imagine a school planning to build a new computer lab. Before starting, they estimate how many computers are needed, how much space is required, and how much it might cost. This helps decide if the project should go ahead.

2.2.6.2 Project Planning

Once the project is approved, more detailed and accurate estimates are prepared. These help in creating a proper budget and setting deadlines. For example, after deciding to build the computer lab, the school now figures out the exact cost of each computer, wiring, tables, chairs, labor, and the time it will take. This helps create a clear plan and avoid surprises.

2.2.6.3 Project Execution

During the actual work, if there are major changes like adding more computers or choosing better furniture, the team must update the estimates. This is called re-estimation. For example, if the school decides to include a smartboard in the lab, they must re-calculate the cost and the extra time needed.

2.2.6.4 Project Monitoring and Control

Throughout the project, regular checks are done to see if things are going as planned. If there are delays or cost increases, adjustments may be needed. For example, if the delivery of computers is delayed, the team might review the schedule and budget again to stay on track.

2.2.6.5 Project Closeout

At the end of the project, the team compares the original estimates with what actually happened. This helps them learn for future projects. For example, if the lab ended up costing more because of unexpected expenses, this information is recorded so future projects can have better estimates.

2.2.7 Project Size Estimation Challenges

Estimating the size of a project is not always easy. Several reasons make this process difficult and can reduce the accuracy of the estimates. Below are some common challenges faced during project size estimation:

2.2.7.1 Unclear Requirements

One of the biggest problems is that the project requirements are not clear in the beginning. Sometimes, clients or stakeholders don't know exactly what they want, or they keep changing their minds during the project. For example, if a team is asked to build a mobile app but the client hasn't decided on key features like login options or payment methods, the development team cannot estimate how long or how big the project will be. These unclear requirements make it very hard to plan properly.

2.2.7.2 Lack of Historical Data

If a company has not worked on similar projects in the past, it becomes hard to guess how much time and effort the new project will take. Without past examples to compare, teams might guess too high or too low. For example, a new software company trying to build an online shopping website for the first time might not have enough information to make accurate estimates. This could lead to poor planning and missed deadlines.

2.2.7.3 Interdependencies

Some projects have many parts that depend on each other. When one task cannot start until another finishes, it creates a chain of dependency. For example, in building a house, the electrical wiring cannot begin until the walls are up. If any task in the chain is delayed, the whole project may slow down. Estimating such projects becomes complicated because one small change can affect many other parts.

2.2.7.4 Productivity Variability

It is also hard to guess how productive team members will be. People work at different speeds, and their availability may change due to sickness, holidays, or other projects. For example, if a key programmer unexpectedly takes leave, the project may slow down. Such changes make it difficult to estimate the size and time of the project accurately.

2.2.7.5 Risks

Every project has risks, but it's not easy to find and measure them in advance. Some risks might include technical problems, team changes, or unexpected costs. For example, if a company is developing a product using a new technology, there might be hidden bugs or integration issues. If these risks are not considered early, the project might need extra time or money later, which can throw everything off schedule.

2.2.8 Ways to Improve Accuracy in Project Size Estimation

Making a good guess about how big a project will be can be hard, but there are some ways to get better at it. Using different methods and following good habits can help make your estimates more accurate and reliable. Here are some important ways to improve how you estimate the size of a project:

2.2.8.1 Define Clear Requirements

First, it is very important to clearly write down what the project needs to do. This means gathering all the details and making sure nothing is missing. Talk to everyone involved in the project, like clients, managers, and team members, many times throughout the project to check that everyone agrees on the requirements. For example, if you are building a website, you should clearly state how many pages it will have, what features it needs, and how it should look before starting to estimate the time or cost.

2.2.8.2 Use Historical Data

Look at data from projects you have done before that are similar to the current one. This information helps you make better guesses because you can see what worked and what didn't in the past. For example, if your team previously built a mobile app in three months, you can use that information to estimate how long a new, similar app might take.

2.2.8.3 Use Estimation Techniques

There are different ways to estimate a project's size, and using more than one method can give you a clearer picture. Some popular techniques are:

- ◆ Analogue Estimation: Comparing the new project to a similar past project.
- ◆ Parametric Estimation: Using mathematical formulas based on project parameters, like cost per feature.
- ◆ Bottom-Up Estimation: Breaking the project into small tasks and estimating each one separately.
- ◆ Three-Point Estimation: Making three guesses: best case, worst case, and most likely and then finding an average.

For example, when building a house, you might use bottom-up estimation by calculating the time needed for the foundation, walls, roof, and so on, then adding those times together.

2.2.8.4 Break Down the Project

Split the project into smaller parts using a method called Work Breakdown Structure (WBS). This means dividing the whole project into clear, simple tasks that can be measured and estimated more easily. For example, instead of estimating the time for "developing software" as one big task, break it down into "writing code for login," "creating the database," and "designing the user interface." Smaller tasks make it easier to give a more accurate estimate.

2.2.8.5 Incorporate Expert Judgement

Ask experienced people who know about the work to help with estimates. Their knowledge and past experience can improve the accuracy of your guesses. For example, a software developer who has built many similar apps can give a better estimate than someone new to the job.

2.2.9 Future Aspects

The evolution of project size estimation will be influenced by advancements in technology and new approaches. Key trends likely to shape this future include:

- ◆ **Enhanced Technology:** Artificial intelligence (AI) may review historical projects and code to provide more precise predictions, taking into account the complexity of different project features.
- ◆ **Insight from Data:** Beyond simply counting lines of code, future estimates might incorporate variables such as user numbers, software type (e.g., mobile versus web applications), and data volume.
- ◆ **Human and AI Partnership:** Combining expert human judgment with AI capabilities can improve accuracy and decision-making in size estimation.
- ◆ **Collaborative Tools:** Platforms that support teamwork among remote or distributed teams will likely improve the accuracy and efficiency of project size estimation.
- ◆ **Agile Practices:** The increasing use of agile methodologies will encourage ongoing estimation and continuous refinement throughout the project lifecycle.

Recap

- ◆ Project size estimation helps predict effort, time, cost, and resources needed for a software project.
- ◆ Accurate size estimation supports better planning, budgeting, scheduling, and resource allocation.
- ◆ It helps identify risks and challenges early to avoid delays and cost overruns.
- ◆ Estimation is a team effort involving project managers, experts, developers, analysts, and clients.
- ◆ Expert Judgment uses the experience of professionals to estimate size when information is limited.
- ◆ Analogous Estimation compares the current project to similar past projects for quick estimation.
- ◆ Bottom-up Estimation breaks the project into small tasks and estimates each one separately.
- ◆ Three-point Estimation considers optimistic, pessimistic, and most likely scenarios to calculate an average.
- ◆ Function Points estimate size based on the software's features, inputs, outputs, and files.

- ◆ COCOMO uses mathematical formulas and project factors like size and complexity to estimate effort and cost.
- ◆ Estimating software size helps plan time, effort, and cost before starting a project.
- ◆ Lines of Code (LOC) counts the number of code lines but is hard to estimate early and varies by programmer skill.
- ◆ Number of entities in an ER diagram shows project complexity early, but entities differ in size and effort needed.
- ◆ Counting processes in a Data Flow Diagram (DFD) helps estimate software size based on functions performed.
- ◆ Function Point Analysis (FPA) measures size by counting and weighing software functions, independent of coding language.
- ◆ Estimation should happen at key stages: initiation, planning, execution, monitoring, and project closeout.
- ◆ Challenges in estimation include unclear requirements, lack of past data, task dependencies, varying productivity, and risks.
- ◆ To improve accuracy, define clear requirements, use historical data, apply multiple estimation techniques, and break down tasks.
- ◆ Expert judgment combined with estimation methods gives better size predictions.
- ◆ Future estimation may use AI and collaboration tools to refine size estimates continuously during the project.

Objective Type Questions

1. What estimation technique uses expert intuition when detailed data is unavailable?
2. Which estimation method compares the current project with similar past projects?
3. Which estimation technique breaks down the project into smaller tasks for detailed estimation?
4. The estimation method that uses optimistic, pessimistic, and most likely scenarios is called?
5. Which method estimates project size based on counting functions like inputs and outputs?

6. What estimation technique considers the number and complexity of use cases and actors?
7. Which technique uses mathematical formulas and past data to estimate size or effort?
8. What does COCOMO stand for?
9. Which method gathers anonymous expert estimates, discusses them, then revises estimates?
10. Which technique uses random sampling and simulations to predict project outcomes?
11. What does LOC stand for in software size estimation?
12. What does KLOC mean?
13. What does NLOC exclude when counting lines?
14. What does KDSI represent?
15. Which estimation method uses the number of entities in an ER diagram?
16. What does a Data Flow Diagram (DFD) primarily show?
17. What is counted in Function Point Analysis as External Inputs?
18. What is the acronym for Unadjusted Function Points?
19. What does TDI stand for in Function Point Analysis?
20. What formula is used to compute the Value Adjustment Factor (VAF)?
21. Which phase involves initial rough estimation of a project?
22. Which project stage involves updating estimates due to major changes?
23. What is a major challenge due to unclear project goals at the start?
24. What type of estimation compares the new project to similar past projects?
25. What structure is used to break a project into smaller tasks?
26. What is one disadvantage of using LOC for early project planning?
27. Which estimation method is independent of programming language?
28. What is the function of historical data in project size estimation?
29. What technology is expected to enhance future project size estimation?

Answers to Objective Type Questions

1. Expert Judgment
2. Analogous
3. Bottom-up
4. Three-point
5. Function Points
6. Use Case Points
7. Parametric
8. Constructive Cost Model
9. Wideband Delphi
10. Monte Carlo
11. Lines of code
12. Thousand of lines of code
13. Comments and blank lines
14. Instructions (Kilo Delivered Source Instructions)
15. Object Point (OP) estimation method
16. Processes
17. Processes or transactions where data enters the system from outside
18. UFP
19. Total Degree of Influence
20. $(TDI \times 0.01) + 0.65$
21. Initiation
22. Project Monitoring and Control stage (Execution)
23. Prioritization, Misaligned efforts and Scope creep
24. Analogue
25. WBS
26. Difficult to estimate accurately before the code is written
27. Function Point Analysis (FPA)
28. Reference (provide a reference for predicting the size, effort, cost, and duration of new projects)
29. Enhancing Estimation Accuracy

Assignments

1. Explain the importance of project size estimation in software engineering and how it impacts planning, budgeting, and scheduling of a software project.
2. Discuss the roles and responsibilities of different stakeholders involved in estimating the size of a software project.
3. Describe the Expert Judgment and Analogous Estimation techniques used for project size estimation, including their advantages and typical use cases.
4. Explain the Bottom-up Estimation method and how it improves the accuracy of software project size estimation.
5. Compare and contrast the Function Points and Use Case Points methods for estimating software project size, highlighting how each technique measures the project scope.
6. Explain the concept of Lines of Code (LOC) as a software size estimation method. Discuss its advantages and disadvantages in project planning.
7. How does the number of entities in an Entity-Relationship (ER) diagram help in estimating the size of a software project? What are the strengths and limitations of this method?
8. Describe how Function Point Analysis (FPA) is used to estimate software size. Outline the key steps involved in calculating Function Point Count (FPC).
9. At what different stages of a software project should size estimations be made, and why is it important to update estimates throughout the project lifecycle?
10. Identify and explain common challenges faced during software project size estimation and suggest ways to improve the accuracy of these estimations.

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.

4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.



CPM, PERT AND GANTT CHART

Learning Outcomes

- ◆ define what the Critical Path Method is
- ◆ list the steps to find the critical path.
- ◆ explain how a PERT chart helps identify the critical path.
- ◆ list the main benefits of using a PERT chart.
- ◆ recall the components of a Gantt chart.
- ◆ describe how a Gantt chart shows task dependencies and timelines.

Prerequisites

Effective planning and scheduling are vital in ensuring the success of any software development project. As projects grow in complexity, traditional methods of task tracking often fall short in managing dependencies, resource allocation, and timelines. This is where project management tools like CPM (Critical Path Method), PERT (Program Evaluation and Review Technique), and Gantt charts become essential. These tools help project managers visualize the sequence of tasks, estimate durations, identify bottlenecks, and monitor progress effectively. for example, consider a real-time example of a mobile app development project for online food delivery. The project involves multiple interdependent tasks such as UI design, backend API development, integration with payment gateways, and testing. Using CPM and PERT allows the team to determine the minimum project duration and critical tasks, while a Gantt chart provides a clear visual timeline for stakeholders to track each phase. Thus, these tools not only enhance control and communication but also reduce risks and ensure timely delivery.

Keywords

Critical Path, Task Dependencies, Work Breakdown Structure (WBS), Task Duration Estimation, Float (Slack), Network Diagram, Project Scheduling

Discussion

2.3.1 Critical Path Method in Project Management

In project management, time and resources are very important. Just like in real life, if you're cooking a big meal and only have a few pots and pans, you need to plan carefully so everything is ready at the right time. This is where the Critical Path Method (CPM) becomes very helpful. CPM is a practical and smart way to plan and manage big tasks or projects so that everything gets done on time. It helps project managers see which tasks are most important and which ones can be delayed without slowing down the whole project.

Imagine you are building a house. Some tasks, like pouring the foundation and must be done before others, like putting up the walls. If the foundation is late, the entire project gets delayed. CPM helps you find these key tasks that control the final deadline. These are part of the "critical path." Knowing this helps teams focus their time and resources on what matters most.

In this article, we'll break down how to identify the Critical Path in a project, understand why it's so important, and look at how it affects the success of real-world projects. We'll also look at easy examples to see how CPM works in action.

The Critical Path Method (CPM) is a useful tool in project management that helps determine the sequence of important tasks that directly affect the total time needed to complete a project. These key tasks form what is called the "critical path." If any of these tasks are delayed, the whole project could be delayed as well. You can think of the critical path as the backbone or spine of the project. Just like how a spine supports your body, the critical path supports the timeline of the project. For example, if you are building a house, pouring the foundation must come before building the walls. If the foundation work is delayed, then everything else will be delayed too. Using CPM is like having a treasure map that guides you through the most important steps. It helps you plan better by showing which tasks need to be finished on time so the project doesn't fall behind. It also helps in using your workers, tools, and time more wisely. For example, if you know that painting the walls can't start until the plaster dries, you can avoid wasting time and make sure workers are ready at the right moment. By identifying the critical path, project managers can set realistic deadlines, plan resources effectively, and prevent or reduce delays in the project.

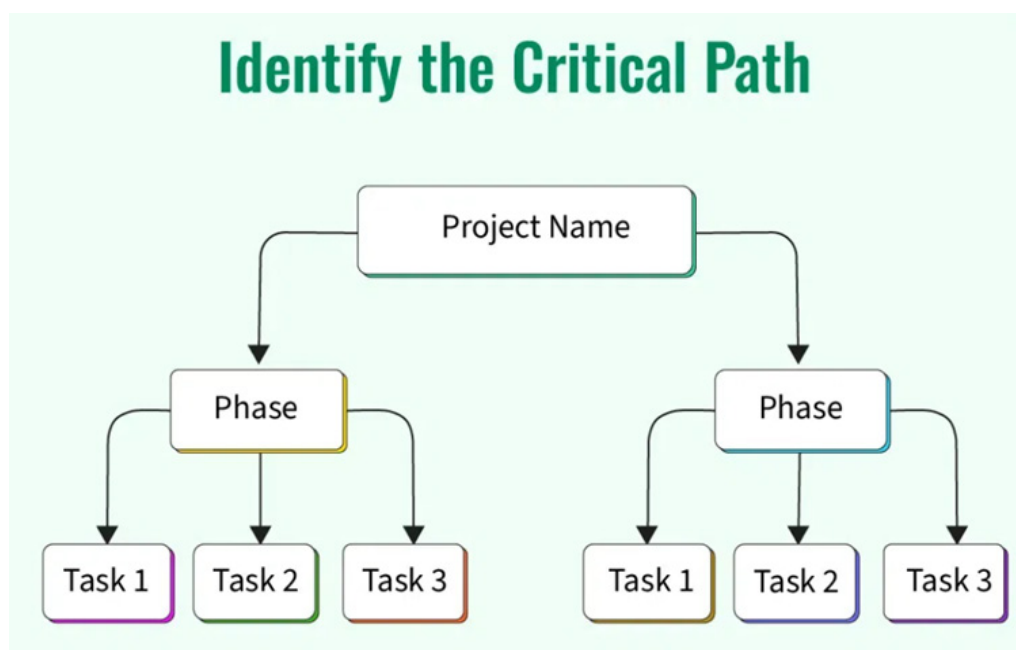
2.3.2 Identifying the Critical Path

To find the critical path in a project, you need to understand which tasks are most important for completing the project on time. These are called critical tasks, and they cannot be delayed without delaying the entire project. You also need to look at non-critical tasks, which can be delayed slightly without affecting the overall schedule. By comparing how long each task takes and seeing which ones are tightly connected, you can figure out the critical path.

2.3.2.1 Listing Your Activities

The first step is to write down all the tasks required to complete your project. This is done by creating a Work Breakdown Structure (WBS). A WBS is like a detailed to-do list that breaks the whole project into smaller, manageable parts. This list is very important because the Critical Path Method (CPM) uses it to calculate which tasks are critical and which are not.

For example, if you are planning a school event, your activities might include booking a venue, sending invitations, arranging food, decorating the place, and setting up audio equipment. Each of these activities needs to be listed clearly. If you forget to list an important task like arranging transport for guests. It could mess up the whole schedule later. Once all tasks are listed, you can move to the next step of finding out how they are connected and which ones are essential for keeping the project on track.



Identify the Critical Path

Fig. 2.3.1 Identification of Critical Path

Illustration:

Consider the scenario where the education department is crafting a new interactive blog article. Sample tasks in the WBS might include:

Table 2.3.1 Work Breakdown structure of a crafting of a new interactive blog article.

Task ID	Task	Duration (in days)
A	Draw up a plan	1
B	Create a draft	3
C	Edit the draft before submitting it.	2
D	Graphics for design posts	5
E	Visuals can be animated	3
G	Put up articles	2

On Starting up how we can do it let's try to identify task dependencies as possible once you have to give your general understanding of everything that needs to be done.

2.3.2.2 Identifying Dependencies

Start by picking out jobs that rely on each other, based on the Work Breakdown Structure (WBS). Some tasks must be completed before others can begin, while some can happen at the same time. These relationships are called dependencies.

Example:

Think about planning a birthday party.

- ◆ You need to book the venue (Task A) before you can send out invitations (Task B).
- ◆ Once the invitations are sent, you can plan the food menu (Task C).
- ◆ At the same time, you can decorate the venue (Task D).
- ◆ But, you can't set up the music system (Task E) until decorations are done.
- ◆ Finally, all of these menus, decorations, and music (Tasks C, D, and E) must be ready before the party begins (Task F).

The tasks that directly affect the timing of the final outcome form what's known as the critical path is the most important sequence that can't be delayed without affecting the entire schedule.

2.3.2.3 Creating a Network Diagram

A network diagram is like a flowchart that shows the order in which tasks need to happen. It uses boxes to show each task and arrows to show the direction and dependency between them.

For example, continuing with the birthday party:

- ◆ A box labeled “Book Venue” would have an arrow pointing to “Send Invitations.”
- ◆ Another arrow might point from “Send Invitations” to “Plan Menu.”
- ◆ Tasks like “Decorate Venue” and “Plan Menu” might happen side-by-side in the diagram, showing they can be done at the same time.

This diagram acts like a map for the project. Just as a GPS helps you find your route to a destination, a network diagram helps the project team follow the plan from start to finish. Before the project officially begins, you should add how long each task is expected to take.

2.3.2.4 Estimating Task Duration

To identify the critical path, you need to figure out how long each task will take. This is called task duration estimation.

You can estimate task time using:

- ◆ Experience – For example, if you've planned a party before, you know that decorating might take 2 hours.
- ◆ Past project data – Maybe last time, setting up music took 1 hour.
- ◆ Industry standards – You might read that sending out invites usually takes 3 days.

Another helpful method is to do a forward pass (calculating start and end times going forward) and a backward pass (working backward from the deadline to check for flexibility in tasks).

2.3.2.5 Calculating the Critical Path

While you can use software to find the critical path, it's also good to understand how to do it manually.

Here's how you do it step-by-step:

Step 1: Write down when each task starts and ends.

Let's say “Book Venue” starts at Day 0 and takes 1 day, so it ends at Day 1.

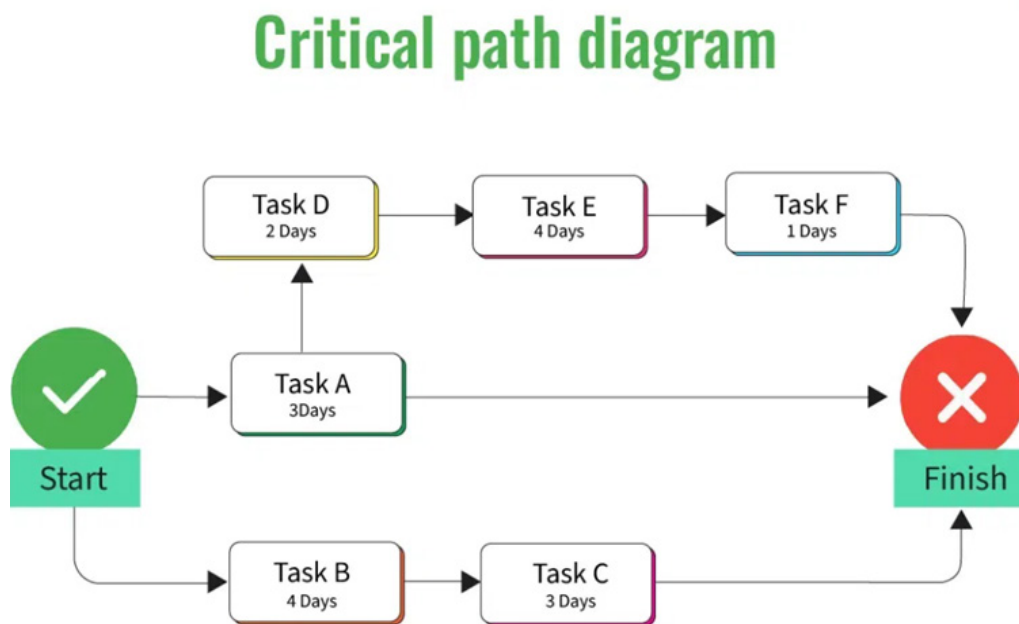
Then “Send Invitations” starts at Day 1 and takes 2 days, ending at Day 3.

Step 2: Continue this way for all tasks. Each task's start time is based on when its previous task ends. The end time is the start time plus its duration.

Step 3: After you've calculated all the timings, look at the longest sequence of connected tasks that determine the total project length. That is your critical path.

Example:

Critical path diagram Fig 2.3.2



Construct the actual project schedule around the identified critical path.

2.3.2.6 Calculating the Float

◆ Understanding Float Calculation

Float, also known as task slack, reflects how much flexibility a task has within a project schedule. It defines the amount of time a task can be delayed without impacting subsequent tasks or the overall project deadline. Calculating float helps assess schedule flexibility and manage potential risks. Both manual techniques and algorithms can be used to determine total and free float.

- ◆ **Total Float:** This is the amount of time a task can be delayed beyond its early start without delaying the project's finish date or breaching schedule constraints.

Formula: Total Float = LS - ES or LF - EF

LS- Latest Start, ES- Earlist start, LF- Latest Finish, EF- Earliest Finish

- ◆ **Free Float:** This measures how long a task can be delayed without affecting the start of the following task. It is particularly relevant when multiple tasks lead into a single successor in a network diagram.

Formula: Free Float = ES (next task) - EF (current task)

Why Project Managers Should Understand Float

- ◆ Keeps the project on track for timely completion
- ◆ Aids in identifying and focusing on critical tasks
- ◆ Provides a strategic advantage in handling risks and unexpected delays

Mastering float concepts enhances project planning and increases the likelihood of project success.

2.3.3 Applications of CPM in Software Development

Software development can be compared to solving a big and complex puzzle. Just like how some puzzle pieces need to be placed before others to complete the picture, certain tasks in software development must be finished before others can begin. This is where CPM, or Critical Path Method, becomes very helpful. CPM helps software development teams identify the most important tasks, the ones that directly affect the project's deadline. These tasks are part of what is called the "critical path."

For example, imagine a team is building a mobile app. They cannot test the app until the coding is done, and they cannot start coding until the design is approved. CPM helps the team figure out the correct order of these tasks and shows which ones must be completed on time to avoid delays. If designing the app takes longer than planned, the entire project might be delayed, so it is part of the critical path. On the other hand, creating the app icon might not delay the launch, so it's not on the critical path.

By using CPM, software development teams can plan better, manage time effectively, and reduce the chances of missing deadlines. It also helps project managers know where to focus their attention and resources, ensuring the most important tasks are completed without delays.

2.3.4 Benefits of Critical Path Identification

- ◆ Proactive Problem-Solving: Recognizing the critical path enables project managers to anticipate potential challenges early and take proactive steps to resolve them.
- ◆ Transparent Communication: Clearly sharing the critical path with clients and stakeholders fosters transparency, aligns expectations, and strengthens trust.
- ◆ Effective Risk Management: Understanding the critical path is vital for prioritizing and addressing risks associated with key project tasks.
- ◆ Building Trust Through Clarity: Openly discussing the critical path encourages a culture of honesty and helps establish strong, trust-based relationships with all involved parties.
- ◆ Enhanced Risk Navigation: Identifying the critical path equips project managers with the strategic insight needed to effectively manage risks,



particularly those tied to essential project activities.

2.3.5 Project Evaluation and Review Technique (PERT)

The Project Evaluation and Review Technique (PERT) is a method used to visually represent the sequence and timing of activities within a project. It serves as a scheduling tool designed to plan, coordinate, and manage tasks effectively. Essentially, PERT supports project management by offering a structured framework for planning and control, acting as a roadmap for the project. It helps identify all key components or milestones involved in the project.

In this approach, a PERT Chart is created to outline the timeline for each specified task. The way tasks or milestones are reported in the PERT Chart closely aligns with the structure used in the Work Breakdown Structure (WBS).

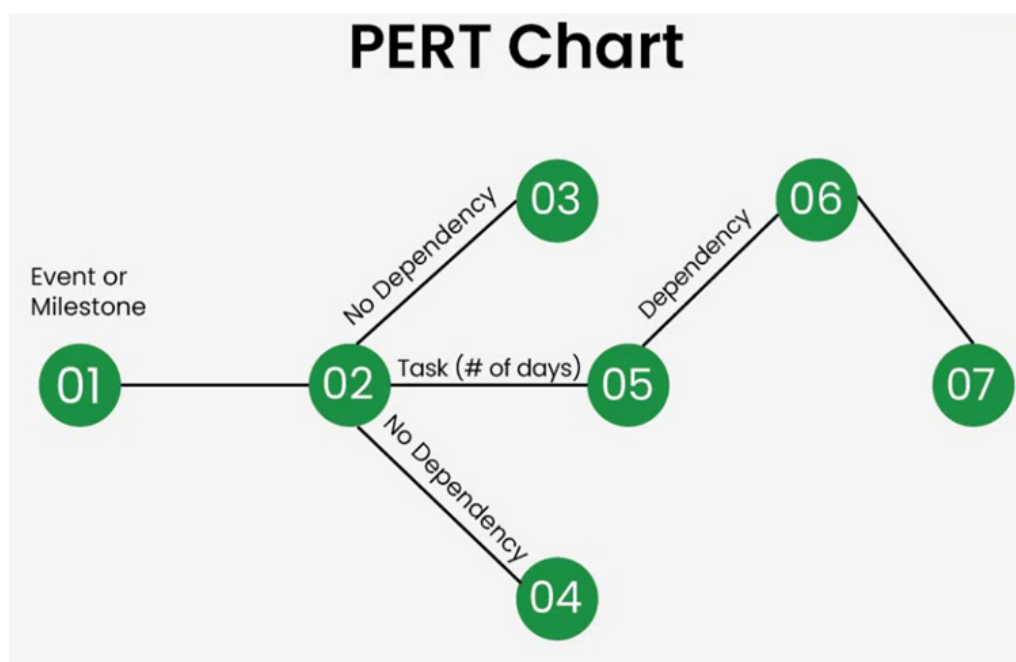


Fig 2.3.3 PERT chart

A PERT (Program Evaluation and Review Technique) chart is a project management tool designed to help plan and schedule project tasks by visually representing the order and timing of activities. It assists in organizing, scheduling, and coordinating the various tasks involved in a project. The main goal of a PERT chart is to identify the critical path, the sequence of key tasks that must be completed on time to avoid delays. This chart is created using data gathered during project planning stages, including effort estimation, selection of an appropriate software development model, and breaking down the project into smaller, manageable subtasks.

2.3.6 Information Included in a PERT Chart

A PERT chart typically includes the following key elements:

- ◆ Nodes: These represent tasks or milestones within the project. Each node

usually displays the task name and may also indicate how long the task is expected to take.

- ◆ **Arrows:** Arrows show the sequence in which tasks must be performed and highlight the dependencies between them. For example, an arrow from Task A to Task B means Task A must be completed before Task B can begin.
- ◆ **Time Estimates:** This refers to the predicted amount of time required to complete each task.
- ◆ **Critical Path:** This is the longest sequence of dependent tasks that determines the shortest possible duration to complete the entire project.
- ◆ **Milestones:** These are significant points or deadlines in the project schedule that mark major accomplishments or phases.

2.3.7 Working of a PERT Chart

A PERT (Program Evaluation Review Technique) chart is a helpful tool used in project planning and scheduling. It helps teams organize and understand all the tasks that need to be done in a project. The chart breaks the whole project into smaller, individual tasks. Each task is shown as a box or a circle, which is called a node. These nodes are connected by arrows that show the order in which tasks must be finished. For example, if you are planning a school event, one task could be "book the venue," and another task could be "set up decorations." You cannot decorate before booking the venue, so an arrow from the first task to the second shows that relationship.

The arrows show which tasks depend on the completion of others. By looking at the PERT chart, the team can clearly see which tasks must happen first and which ones can happen later or at the same time. One of the most important parts of using a PERT chart is finding the critical path. This is the longest path through the chart, showing the tasks that directly affect how long the project will take. If any task on the critical path is delayed, the whole project will be delayed. For example, in building a house, tasks like "lay the foundation" and "build the walls" might be on the critical path because they must be done in a specific order, and other work depends on them.

2.3.8 Creation of a PERT chart

Creating a PERT (Program Evaluation Review Technique) chart helps plan and manage complex projects. It shows the order of tasks and helps estimate the total time needed. To make one, follow these easy steps:

Step 1: Identify Project Tasks

The first step is to list all the tasks that need to be completed for the project. Think of the final goal and then break it down into smaller, manageable steps. For example, if your project is to organize a college seminar, some tasks could be "book the venue," "invite speakers," "prepare materials," and "promote the event." Talk to your team members to make sure you don't miss anything, and try to describe each task using simple and clear action words. This way, everyone understands what needs to be done.

Step 2: Define Task Dependencies

After listing the tasks, figure out which ones depend on others. This means identifying tasks that cannot start until another task is finished. For example, you cannot send out invitations until the seminar date and venue are confirmed. These connections help you decide the correct order in which the work should be done. Understanding task dependencies ensures that no step is started too early or too late.

Step 3: Estimate Timeline

Next, you estimate how long each task might take. For every task, think of three time frames: the best-case scenario (optimistic time), the worst-case scenario (pessimistic time), and the most realistic guess (most likely time). For example, booking a venue might take just one day if everything goes well, but could take up to five days if there are delays. A likely estimate might be three days. These estimates help you figure out how long the whole project will take and let you plan better.

Step 4: Calculate Critical Path

The critical path is the longest chain of tasks that must be done one after the other, without delay, to complete the project on time. It shows which tasks are most important and cannot be postponed. For example, if preparing seminar materials takes longer than expected and is on the critical path, it could delay the whole event. Identifying the critical path helps you focus on the most time-sensitive tasks and manage your priorities effectively.

Step 5: Manage Task Progress

Once the project starts, you need to keep track of how each task is going. Regularly check if tasks are being completed as planned. If there are any delays, fix the problems and adjust your schedule if needed. For example, if the printing of seminar materials is delayed, you might need to speed up the promotion process or find alternative printing options. Monitoring progress helps you avoid surprises and keeps the project on track.

By following these steps, you can create a useful and clear PERT chart that helps you organize your project, estimate time accurately, and complete tasks in the right order.

2.3.9 Properties of PERT Chart

The PERT (Program Evaluation and Review Technique) chart has several important properties that make it useful for project planning and management.

The main properties of PERT chart are as following:

- ◆ It serves as a base for obtaining the important facts for implementing the decision-making.

PERT charts help managers collect and organize all the important details they need to make smart decisions. For example, if a school is building a new computer lab, the PERT chart will show how long each task (like wiring, buying computers, setting up desks) will take. This helps school leaders decide when to start and who should do each job.

- ◆ It forms the basis for all the planning activities.

All planning in a project can start with the PERT chart. It lays out every task clearly. For example, if a company wants to launch a new product, the PERT chart will show every step from market research to advertising, also making it easier to plan each phase in the right order.

- ◆ PERT helps management in deciding the best possible resource utilization method.

PERT charts allow managers to assign the right people, tools, and time to each task. For example, if only one technician is available to install equipment, the chart helps schedule the work so that technician is used efficiently without delays.

- ◆ PERT takes advantage by using time network analysis techniques.

This technique means that PERT shows how long each part of a project will take and how they connect. It's like mapping out the shortest and fastest way to finish everything. For example, when planning a wedding, you can use a PERT chart to organize catering, decoration, and invitations by showing what can be done at the same time and what must wait for something else.

- ◆ PERT presents the structure for reporting information.

PERT charts make it easier to report and share project progress. If a team is building a mobile app, the chart can be used in meetings to show which parts are done and what still needs work, helping everyone stay updated.

- ◆ It helps the management in identifying the essential elements for the completion of the project within time.

With a PERT chart, managers can focus on the tasks that are most important for finishing the project on time. For example, when making a school magazine, the task of collecting student articles might be essential, so it gets top priority.

- ◆ It specifies the activities that form the critical path.

The critical path is the sequence of tasks that directly affect the final deadline. If any of these are delayed, the whole project will be delayed. For example, in constructing a house, laying the foundation, building walls, and putting the roof are on the critical path. Missing a deadline here affects the entire project.

- ◆ It describes the probability of completion of a project before the specified date.

PERT also helps estimate the chances of finishing a project on time. For example, if a construction project has some tasks with uncertain time durations, PERT can calculate the likelihood of completing everything before a set deadline.

- ◆ It describes the dependencies of one or more tasks on each other.

Some tasks depend on others being finished first. A PERT chart clearly shows these dependencies. For example, in preparing a stage show, you can't begin rehearsals until the script is written. The chart will show that rehearsal depends on script completion.



- ◆ It represents the project in graphical plan form.

A PERT chart uses a visual layout (like boxes and arrows) to show the whole project plan. This makes it easier to understand. For example, if a teacher is organizing a school trip, they can use a PERT chart to visually map out steps like getting permissions, booking transportation, and collecting student fees.

2.3.10 Drawbacks of the PERT Chart

The PERT chart comes with several limitations:

- ◆ Its intricate structure makes implementation challenging.
- ◆ Activity time estimates in PERT are often based on subjective judgment, which reduces accuracy.
- ◆ Managing and updating PERT charts can be both costly and complicated.
- ◆ The actual time distribution may differ from the assumed beta distribution used in PERT, leading to inaccurate predictions.
- ◆ PERT tends to underestimate the project's expected completion time, as delays in non-critical paths can potentially shift them into critical ones.

The Project Evaluation and Review Technique (PERT) is a very useful method for managing projects in a smart and organized way. It helps project teams clearly see the order in which tasks need to be done. For example, if you're building a house, you need to lay the foundation before you can start putting up the walls. PERT shows these kinds of task sequences and helps everyone understand which steps depend on others. One of the most important things PERT does is help identify the critical path. This is the set of tasks that must be finished on time for the whole project to be completed on schedule. If even one task on this path is delayed, the whole project may be delayed.

Using PERT makes it easier for organizations to plan their projects better. It also helps in using resources wisely, such as time, workers, and money. For example, if a school is planning to organize a big event, PERT can help them figure out how many people they need, what tasks must be done first, and how to stay on schedule. Overall, PERT increases the chances of finishing projects successfully by helping everyone stay focused and work more efficiently toward their goals.

2.3.11 Gantt Chart

The Gantt chart, named after American mechanical engineer and management consultant Henry Gantt, was developed by him in the 1910s. This visual tool was designed to assist in planning, managing, and monitoring production processes within manufacturing settings. Gantt's innovation played a significant role in shaping modern project management practices and continues to be extensively used across various industries today.

In project management, most projects include many different tasks and activities that need to be done. For example, if you are building a house, tasks might include designing the plan, laying the foundation, constructing the walls, and installing electricity. Each of

these tasks has a specific time it needs to start and finish. Sometimes, one task depends on the completion of another. So, managing all these tasks together can become difficult, especially when the project is big or complex.

As a project manager, it's important to keep track of all the tasks, how long each one will take, and how they are connected. If one task is delayed, it can affect the rest of the project. For example, if the foundation takes longer to build, the wall construction also gets delayed. That's why understanding the relationships between tasks is very important.

A Gantt Chart is a useful tool that helps us manage project timelines. It shows all the tasks in a project along with their start and end dates. You can also see which tasks are happening at the same time and which ones depend on others. For example, in a school science fair project, the Gantt Chart can help you plan when to do research, buy materials, build the model, and practice the presentation.

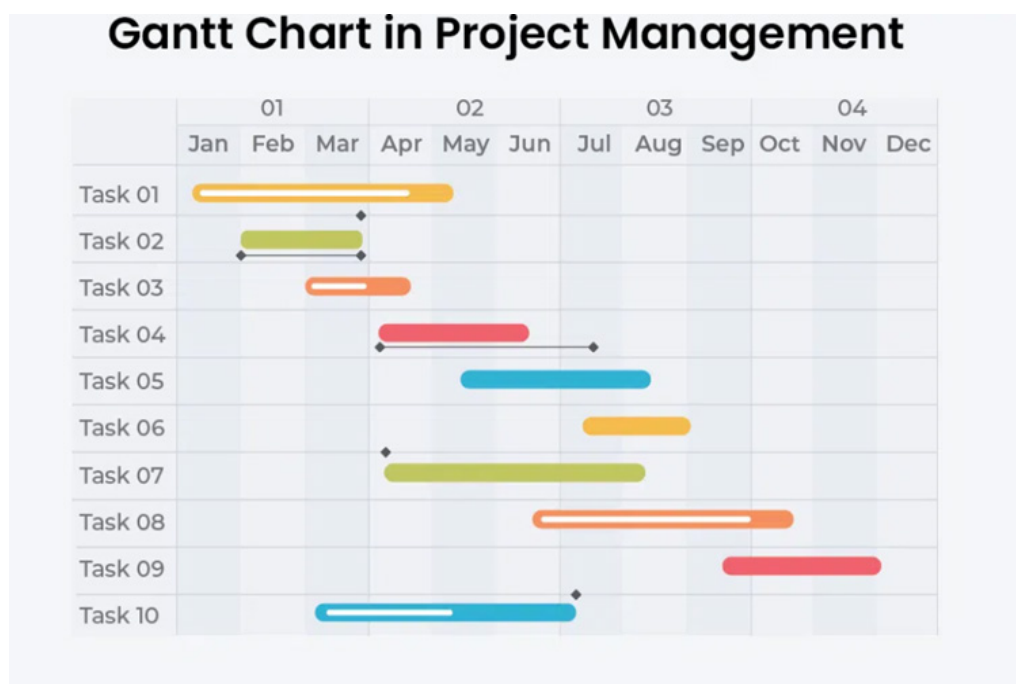


Fig 2.3.4 Gantt Chart

2.3.12 Understanding the Gantt Chart

A Gantt Chart is a powerful visual tool used in project management to outline and track the progress of tasks over time. It is typically represented as a horizontal timeline composed of color-coded bars, where each bar corresponds to a specific task within the project. These bars provide crucial information, such as the start and end dates of tasks, their duration, the individuals or teams responsible for them, and their deadlines.

By organizing tasks along a timeline, the Gantt chart clearly displays both individual task schedules and how they fit into the overall project framework. It distinguishes between terminal elements as tasks that cannot be broken down further, and summary elements, which represent groups of related tasks.

In essence, the Gantt chart serves as a comprehensive visual map that consolidates all project activities in relation to a fixed deadline, making it easier for project managers and teams to monitor progress, manage resources, and ensure timely completion of deliverables.

2.3.13 Uses of a Gantt Chart

A Gantt chart is a powerful visual tool used to plan, schedule, and track the progress of a project over time. By displaying tasks along a timeline, it provides project managers and also programme and portfolio managers, with a clear overview of the project's structure, key milestones, and deadlines. It plays a critical role in a wide range of project management activities, including resource allocation, task sequencing, time management, and performance tracking.

The primary uses of a Gantt chart include:

- ◆ **Breaking Down the Project into Manageable Tasks:**

It helps decompose complex projects into smaller, clearly defined activities or work packages, making planning and execution more efficient.

- ◆ **Identifying Task Dependencies:**

A Gantt chart allows managers to pinpoint which tasks are dependent on the completion of others. Understanding these relationships is essential for creating a logical and realistic workflow.

- ◆ **Allocating Resources Efficiently:**

By assigning specific tasks to team members and indicating timelines, the chart supports optimal use of available human and material resources.

- ◆ **Establishing Task Sequences and Durations:**

Gantt charts help determine the order in which tasks need to be performed and estimate how long each will take, forming the foundation of the project schedule.

- ◆ **Tracking Progress and Performance:**

As work progresses, the Gantt chart can be updated to reflect completed tasks. This allows for a comparison between planned timelines and actual performance, highlighting any delays, bottlenecks, or issues.

- ◆ **Enhancing Communication and Transparency:**

Project plans and progress can be easily shared with stakeholders, clients, and team members in a visually intuitive format. This transparency helps ensure that everyone is aligned on objectives, timelines, and expectations.

- ◆ **Promoting Team Collaboration:**

By clearly showing who is responsible for what and when, Gantt charts foster accountability and coordination among team members, ultimately contributing to smoother project execution.

2.3.14 Components of a Gantt chart

At first glance, a Gantt chart might look confusing. But once you learn how to read it, it becomes a very useful tool for tracking the progress of a project. It helps you understand what tasks have been completed, what is currently happening, and what still needs to be done to finish the project successfully.

A Gantt chart is made up of different parts that work together to show the full picture of a project. Understanding each of these parts will help you read and use a Gantt chart more effectively. Below are the main components of a Gantt chart:

Task List

This is a list of all the tasks or activities that need to be completed in a project. For example, if you are planning a school event, tasks might include “Book the venue,” “Send invitations,” “Arrange food,” and “Set up decorations.” These tasks are usually listed on the left side of the chart.

Timeline

The timeline shows the time period for the project. It could be in days, weeks, or months depending on how long the project will take. For example, if your project lasts for two weeks, the timeline might show dates from June 1 to June 14. This timeline runs across the top of the Gantt chart.

Dateline

The dateline is a vertical line that shows today’s date on the timeline. It helps you quickly see what tasks should already be done, which ones are in progress, and what tasks are coming up. For example, if today is June 5, the dateline will be placed at June 5 on the timeline.

Bars

Bars represent the duration of each task. The length of the bar shows how long the task is expected to take. For example, if “Book the venue” takes three days, a bar will stretch from June 1 to June 3. These bars are placed next to the tasks and aligned with the timeline.

Milestones

Milestones are important points or goals in the project. They usually mark the end of a major phase or the completion of a key task. For example, “Venue booked” or “Invitations sent” can be milestones. On the chart, milestones are often shown with a special symbol like a diamond.

Dependencies

Dependencies show the relationship between tasks. Some tasks can’t start until others are finished. For example, you can’t “Set up decorations” until the “Venue is booked.” These relationships are shown with arrows connecting the tasks.

Progress

Progress shows how much of a task has been completed. This is often shown by shading part of the task bar. For example, if “Send invitations” is halfway done, then half of its bar will be shaded. This helps track how far along the project is.

Resource Assigned

This part shows who is responsible for each task. It could be a person or a team. For example, “John” might be assigned to “Arrange food,” while “Sarah” handles “Send invitations.” This helps in managing people’s responsibilities and making sure everyone knows what they need to do.

2.3.15 Key Features of a Gantt Chart

A Gantt Chart offers several valuable features that make it a powerful tool for project management:

1. Effective Project Planning and Management

Gantt Charts provide a clear overview of the project timeline, enabling teams to schedule deliverables, monitor pending tasks, and adjust deadlines as needed.

2. User-Friendly Interface

With intuitive drag-and-drop functionality, it’s easy to modify task durations and update project timelines with minimal effort.

3. Seamless Integration with Existing Workflows

Gantt Charts can be incorporated into current project frameworks without starting from scratch, simplifying the transition and saving time.

4. Well-Suited for Managing Complex Projects

For projects with multiple layers of tasks, issues, and subtasks, Gantt Charts offer an organized structure that simplifies task management and enhances visibility.

5. Insightful Visual Timeline and Critical Path Analysis

In addition to showing task sequences, Gantt Charts highlight critical paths, helping project managers focus on essential activities that directly impact deadlines and outcomes.

2.3.16 Creating a Gantt Chart

Creating a Gantt Chart might seem complicated at first, but it becomes easier once you understand the process. Although the way you create it can vary depending on the size of your team, the current stage of your project, deadlines, and the structure of your tasks and subtasks, the overall method is quite straightforward. A Gantt Chart helps you plan, track, and manage your project by showing tasks along a timeline. Let’s look at the step-by-step process with simple examples to help you understand better.

Step 1: List out every task you will need to complete to finish the project.

The first thing you need to do is make a complete list of all the tasks that are part of your project. Think of it as writing a to-do list. For example, if you are building a website, you might list tasks like "Set up the backend server", "Connect the database", "Run testing", and "Launch the website". These are the basic steps you must complete to finish your project.

Step 2: Define chronological order and put each task in order of when it needs to be done.

Once you have your task list, arrange the tasks in the order they need to happen. This means deciding which task should be done first, second, and so on. Also, see if any task depends on another one. For example, you can't test a website until the backend and database setup is finished. So, "Testing" must come after "Backend Configuration" and "Database Integration".

Step 3: Estimate how much time it will take you to complete each task.

Now, look at each task and figure out how much time it will take to complete. Be realistic. If "Backend Configuration" normally takes three days, note that down. This step helps you understand how long the entire project might take and ensures that tasks are done at a good quality, as discussed during the planning phase.

Step 4: Assign a realistic start date and deadline for each task.

After estimating time, decide the start and end dates for each task. Make sure these dates are practical. For example, if "Testing" will take 2 days, and the previous task ends on June 10, then plan to start testing on June 11 and finish by June 12. Think about whether any task might take longer due to challenges, and adjust the dates accordingly.

Step 5: Fill up your Gantt Chart and continuously update it with actual timelines.

Now, take all this information and fill it into your Gantt Chart. You can use tools like Microsoft Excel, Google Sheets, or project management software like Trello or Asana with Gantt view. Draw bars that show when each task starts and ends. As your project moves forward, keep updating the chart to match the real progress. For example, if a task took longer than expected, adjust the timeline in the chart to stay on track.

By following these steps, It is easy to create and maintain a Gantt Chart that helps you manage your project smoothly.

Work Breakdown Structure (WBS)	Start date ▲	Due date ▲	Planned Duration ▲	Assignee	Time tracking	Due date
☐ ☐ ☒ ☒ ☒	☒ incl. subtasks*	☒ incl. subtasks*			☒ incl. subtasks*	☒ incl. subtasks*
ICT-API	15 Dec 2023	24 Dec 2023	6 wdays			24 Dec 2023
without Fix versions	15 Dec 2023	24 Dec 2023	6 wdays			24 Dec 2023
without Components	15 Dec 2023	24 Dec 2023	6 wdays			24 Dec 2023
AUTH-3	15 Dec 2023	24 Dec 2023	6 wdays*			24 Dec 2023
AUTH-13	18 Dec 2023		1 wdays*			
AUTH-14	18 Dec 2023	18 Dec 2023	1 wdays			18 Dec 2023
AUTH-15	18 Dec 2023	18 Dec 2023	1 wdays			18 Dec 2023
AUTH-1	18 Dec 2023	18 Dec 2023	1 wdays			18 Dec 2023
AUTH-2	18 Dec 2023	18 Dec 2023	1 wdays			18 Dec 2023
AUTH-5	18 Dec 2023	18 Dec 2023	1 wdays			18 Dec 2023
AUTH-6	18 Dec 2023		1 wdays*			

Fig 2.3.5 Gantt Chart with task visualization

2.3.17 Benefits of Using a Gantt Chart

The Gantt Chart offers several key benefits in project management: See fig 2.3.5

- ◆ **Clear Task Visualization:** It allows for easy recognition of individual tasks, their relationships, and their current progress, making it simpler to understand the overall project workflow.
- ◆ **Assessment of Resource Utilization:** Gantt charts enable the evaluation of how critical resources are being used. They help identify dependencies, analyze the impact of scheduling changes, and can include visual alerts to highlight potential scheduling conflicts.
- ◆ **Efficient Task Sequencing:** Project managers can effectively organize the sequence in which tasks should be executed. This helps in establishing task priorities and ensuring smoother execution during the project lifecycle.

2.3.18 Limitations of Gantt Charts

1. **Time-Intensive Setup:** Creating a Gantt Chart involves detailed planning of tasks, assigning resources, setting deadlines, identifying dependencies, and more. This process can be quite time-consuming, especially for large or complex projects.
2. **Can Become Overly Complex:** Including too much detailed information such as numerous tasks and timelines can make the chart cluttered and difficult to interpret, which hampers effective timeline management.
3. **Challenging to Maintain:** In projects that require switching between various tools and platforms, keeping the Gantt Chart updated becomes difficult. Frequent reconfiguration is needed whenever the workflow changes, making the chart hard to maintain consistently.

Gantt charts are valuable tools in project management, especially when handling complex projects with multiple tasks and dependencies. They are particularly useful in software development for organizing and tracking activities such as coding, testing, debugging, and interface design. By visually outlining tasks, timelines, and dependencies, Gantt charts help ensure work is completed in the correct order and on schedule and it is crucial when meeting tight deadlines like product launches. They also clarify who is responsible for each task, improving team coordination and accountability. Whether using traditional or agile approaches, Gantt charts support effective planning, scheduling, and progress monitoring to ensure timely and successful project delivery.

Recap

- ◆ CPM helps find the most important tasks that directly affect a project's completion time.
- ◆ The “critical path” is the longest sequence of tasks that cannot be delayed without delaying the whole project.
- ◆ Tasks not on the critical path can be delayed slightly without affecting the project deadline.
- ◆ Listing all tasks (Work Breakdown Structure) is the first step to using CPM.
- ◆ Identifying task dependencies shows which tasks must happen before others.
- ◆ A network diagram visually maps task order and dependencies with boxes and arrows.
- ◆ Estimating how long each task takes is key to calculating the critical path.
- ◆ The critical path helps managers focus time and resources on tasks that keep the project on schedule.
- ◆ Float (or slack) is the amount of time a task can be delayed without affecting the project deadline.
- ◆ CPM improves project planning, risk management, and helps prevent delays by identifying key tasks.
- ◆ PERT charts help estimate how long a project will take to complete.
- ◆ They show which tasks are most important to finish on time (critical path).
- ◆ PERT helps plan the order of tasks and shows which ones can be delayed.
- ◆ It gives a clear picture of task start and end dates.
- ◆ PERT is useful for organizing complex project information visually.
- ◆ But PERT charts can be hard to create and update, especially for big projects.
- ◆ Gantt charts show tasks on a timeline, making it easy to track progress.
- ◆ They help assign tasks, set deadlines, and see who is doing what.
- ◆ Gantt charts show which tasks depend on others and help avoid delays.
- ◆ They are easy to read but can become messy if the project is too detailed

Objective Type Questions

1. What does CPM stand for in project management?
2. What is the critical sequence of tasks called in CPM?
3. Which task type cannot be delayed without delaying the entire project?
4. What is the first step in identifying the critical path?
5. What kind of diagram shows tasks and their dependencies in CPM?
6. What does WBS stand for?
7. What is the term for tasks that can be delayed without affecting the project deadline?
8. What calculation is done to find the longest sequence of connected tasks?
9. What is the term for flexibility time allowed in a task schedule?
10. What is the formula for calculating total float?
11. Which method calculates the earliest and latest start times in CPM?
12. What project management technique uses nodes and arrows to show task order?
13. What is the full form of PERT?
14. Which chart is used to visualize task dependencies and timelines?
15. What is the main goal of a PERT chart?
16. What property of PERT helps estimate project completion probability?
17. What term describes tasks that rely on completion of previous tasks?
18. What step involves estimating the duration of each task in project planning?
19. What helps project managers to avoid wasting time on tasks that can be delayed?
20. What is the longest path through the network diagram called?
21. What chart is used to estimate a project's completion time?
22. Which chart helps in identifying critical path activities?
23. What type of distribution is assumed in PERT time estimation?
24. Which chart visually tracks project tasks over time?
25. Who developed the Gantt Chart?

26. What type of tasks in a Gantt chart cannot be broken down further?
27. What symbol is often used to represent milestones in Gantt charts?
28. Which line shows the current date on a Gantt chart?
29. What is the main visual component that represents task duration in a Gantt chart?
30. What shows the relationship between tasks in a Gantt chart?
31. What part of a Gantt chart indicates task completion status?
32. What tool helps decompose complex projects into smaller tasks?
33. Which chart allows managers to identify task dependencies?
34. What type of task in PERT can delay the entire project if it is delayed?
35. Which chart is known to underestimate project completion time?
36. Which chart offers better visualization for task relationships?
37. What is listed on the left side of a Gantt chart?
38. What is one key benefit of PERT in project management?
39. Which project management tool is helpful in resource allocation?
40. What is one major limitation of Gantt charts?

Answers to Objective Type Questions

1. Critical Path Method
2. Critical Path
3. Critical
4. Listing of Activities
5. Network Diagram
6. Work Breakdown Structure
7. Non-Critical
8. Critical Path
9. Float
10. $\text{Float} = \text{LST} - \text{EST}$
11. Forward Backward Pass

12. PERT
13. Project Evaluation Review Technique
14. PERT Chart
15. Plan and Schedule tasks
16. probabilistic time estimates property
17. Dependencies
18. Estimation
19. Float
20. CriticalPath
21. PERT
22. PERT
23. Beta
24. Gantt
25. Henry L. Gantt
26. Terminal
27. Diamond
28. Dateline
29. Bars
30. Dependencies
31. Progress
32. Gantt
33. Gantt
34. Critical
35. PERT
36. Gantt
37. Tasks or activities
38. Scheduling
39. Gantt
40. Complexity

Assignments

1. Explain the concept of the Critical Path Method (CPM) in project management. Why is identifying the critical path essential for successful project completion? Illustrate your answer with a suitable example.
2. Describe the steps involved in identifying the critical path of a project. How do task dependencies and duration estimation contribute to this process?
3. What is float in project management? Differentiate between total float and free float with formulas and examples. Why is understanding float important for a project manager?
4. Compare and contrast the Critical Path Method (CPM) and the Program Evaluation and Review Technique (PERT). How do both tools assist in planning and managing complex projects?
5. Describe the structure and components of a PERT chart. How does a PERT chart help project managers visualize and manage project tasks effectively?
6. Discuss the key benefits and drawbacks of using a PERT chart in project management. Provide suitable examples to illustrate how PERT assists in planning and decision-making.
7. Explain the concept of the critical path in a PERT chart. Why is it important for project managers to identify the critical path, and how does it influence project scheduling and resource allocation?
8. Describe the main components of a Gantt chart. How does each component contribute to effective project tracking and communication among team members?
9. Compare and contrast the uses of PERT charts and Gantt charts in project management. Under what project conditions would one be more suitable than the other? Justify your answer with real-world examples.
10. Outline the step-by-step process of creating a Gantt chart. Illustrate each step using a simple project scenario (e.g., organizing a school event or launching a website). How does regular updating of the chart affect project success?

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.



SOFTWARE REQUIREMENTS ELICITATION AND SRS

Learning Outcomes

The student will be able to

- ♦ define the term "requirements elicitation."
- ♦ identify key stakeholders involved in requirements elicitation.
- ♦ list common methods used for requirements elicitation.
- ♦ explain the importance of stakeholder engagement in requirements elicitation.
- ♦ describe the main activities involved in requirements elicitation.
- ♦ make aware of Software Requirements Specification Document

Prerequisites

In today's fast-paced software development environment, accurately understanding what users and stakeholders expect from a system is crucial for building successful applications. Software requirements act as the foundation for all subsequent design, coding, testing, and maintenance activities. Without a clear grasp of these needs, projects often face delays, increased costs, and unsatisfactory outcomes. Requirements elicitation is the structured process of uncovering and defining these essential needs from various stakeholders, ensuring the software product aligns with real-world demands. This process leads to the creation of a Software Requirements Specification (SRS) document, which serves as the definitive guide for developers and clients throughout the project lifecycle. For example, consider developing an online food delivery application. Accurately capturing requirements such as menu browsing, order placement, payment options, delivery tracking, and user feedback mechanisms early on is vital to avoid costly changes later and to deliver a product that meets user expectations effectively.

Keywords

Stakeholders, Functional Requirements, Non-functional Requirements, Interviews, Prioritization, Use Cases, Validation

Discussion

Requirements elicitation refers to the structured process of collecting, analyzing, and defining the functional and non-functional needs of a software system from various stakeholders such as clients, end-users, and domain experts. It is a foundational phase in software development that aims to capture what the software must do and how it should perform under specific conditions. The primary objective of this process is to establish a thorough and accurate understanding of the customer's expectations and business goals. This understanding helps ensure that the software product developed aligns with the actual needs of its users and supports their tasks effectively. By engaging stakeholders early and clearly identifying their requirements, requirements elicitation reduces the risk of misunderstandings, missed features, or costly revisions later in the project. Ultimately, it lays the groundwork for a well-defined set of specifications that guide the design, implementation, and validation of the software system.

2.4.1 Requirement Elicitation Concept

Requirements elicitation is the essential and often challenging process of gathering information about what a software system must accomplish, typically conducted through interaction with users, clients, and various stakeholders. Actually it is one of the most communication intensive and error-prone activities in software engineering, it demands a strong and effective collaboration between customers and developers to ensure the system aligns with user needs. This process occurs early in the software development life cycle and involves identifying, collecting, analyzing, and refining the requirements that will shape the software's design and functionality. Engaging stakeholders from different parts of the organization which includes business owners, end-users, and technical experts is crucial, as their perspectives help in forming a complete understanding of the system requirements. The goal of requirements elicitation is to produce a clear, concise, and well-defined set of requirements that can be used as a reliable foundation for subsequent development stages. However, eliciting accurate requirements is complex, as users may not always be able to articulate their needs clearly, especially when it comes to critical aspects such as safety and dependability. To address these challenges, a variety of techniques are employed, such as interviews, surveys, user observations, workshops, brainstorming sessions, use cases, role-playing, and prototyping, each of which helps uncover different facets of the system requirements.

2.4.2 Significance of Requirements Elicitation

Alignment with Business Goals:

Requirements elicitation ensures that software development efforts align with the

organization's broader strategic objectives. Understanding the business environment enables the creation of solutions that deliver real value to the enterprise.

Enhancing User Satisfaction:

Involving end users in the requirements-gathering process helps developers better understand their needs and expectations. This leads to the creation of software that is more user-friendly and widely accepted, ultimately increasing user satisfaction.

Reducing Costs and Development Time:

Clear and accurate requirements minimize misunderstandings and reduce the likelihood of rework during development. This efficiency results in lower costs and helps ensure that the project stays on schedule.

Meeting Legal and Regulatory Standards:

In regulated industries, effective requirements elicitation is vital to ensure compliance with relevant laws and industry standards. This is especially important in sectors such as healthcare, finance, and aerospace, where regulatory adherence is non-negotiable.

Improved Traceability and Documentation:

Well-documented requirements provide a foundation for traceability throughout the software lifecycle. This traceability supports effective testing, validation, and maintenance by linking each component of the system to its corresponding requirement.

2.4.3 Activities involved in requirement elicitation

Requirements elicitation involves several important activities that help in accurately understanding and defining what a software system should do. One of the first steps is gaining a clear understanding of the broader domain or environment in which the system will operate. This provides valuable context and helps ensure the system fits well within its intended application area. Next, it is essential to deeply explore and analyze the specific problems faced by the customer that the system is meant to solve. Without a thorough grasp of the customer's challenges, the resulting solution may fall short of expectations. Another key activity is identifying how the system will interact with external elements, such as other software systems, hardware components, or regulatory requirements. Additionally, requirements elicitation includes a detailed examination of what users need from the system, which may involve gathering input through interviews, surveys, or observations. Lastly, it is important to define any constraints that could affect the development of the system, such as technical limitations, budget restrictions, timeframes, or compliance standards. All of these activities work together to ensure that the requirements gathered are complete, realistic, and aligned with both user expectations and business goals.

2.4.4 Methods used for Requirements Elicitation

There are various techniques available for eliciting requirements. See fig 2.4.1 Some of the commonly used methods include:

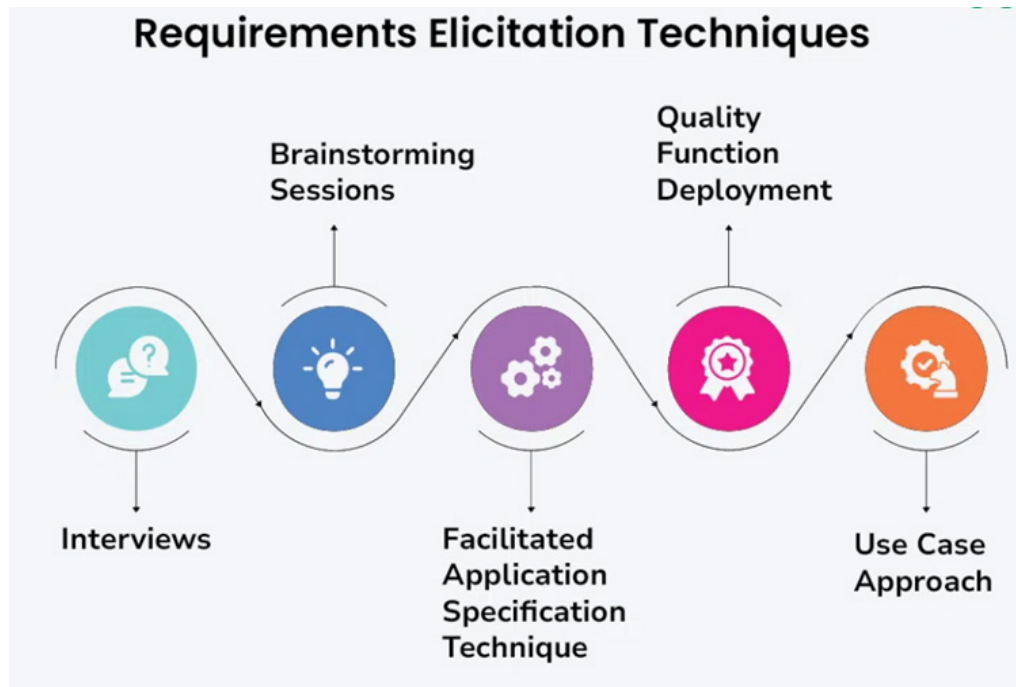


Fig 2.4.1 Requirements elicitation Techniques

2.4.4.1 Interviews

Interviews are a commonly used method in requirements elicitation to gain insights into what customers expect from a software system. The main purpose of an interview is to gather detailed information directly from stakeholders about their needs, problems, and desired features. Since it is not feasible to speak with every stakeholder, a few representatives are chosen based on their knowledge, experience, and ability to speak on behalf of a larger group. Interviews can be conducted in two main formats: open-ended and structured. In open-ended interviews, there is no fixed set of questions, allowing for a free-flowing conversation that helps uncover important information, often through context-free questions such as "What problems do you face in your daily workflow?" On the other hand, structured interviews follow a planned agenda with specific questions designed to guide the conversation toward targeted topics. For example, a structured interview might include questions like "What features would you like to see in the new inventory system?" or "What data do you frequently access in your current application?" In some cases, interviewers may use a detailed questionnaire to ensure consistency across multiple interviews. This approach helps in collecting comprehensive and focused feedback for software development.

2.4.4.2 Brainstorming Sessions

Brainstorming sessions are collaborative group activities designed to encourage the free flow of ideas and promote creative thinking. This technique provides a platform where participants can openly share their perspectives, helping to uncover a wide range of potential requirements or solutions. To ensure the session remains productive and fair, a skilled facilitator is usually needed to manage group dynamics, prevent bias, and resolve any conflicts that may arise during the discussion. Every idea presented

during the session is carefully recorded and made visible to all participants, often using whiteboards, sticky notes, or digital tools. For example, in a software project, team members might suggest features like user login, data export, or mobile support, all of which are noted during the session. After the brainstorming concludes, the collected inputs are reviewed and organized into a formal document, often listing the proposed requirements along with their relative priority, if known. This final document serves as a reference point for the development team as they begin planning and building the system.

2.4.4.3. Facilitated Application Specification Technique

The Facilitated Application Specification Technique (FAST) is a collaborative approach used in software development to reduce misunderstandings between developers and customers. Its main goal is to close the expectation gap, that is, the difference between what developers believe they need to create and what the customers actually expect to receive. To achieve this, the technique brings together a team of stakeholders, including users, developers, and business analysts, to actively participate in the process of gathering requirements. Each participant is asked to list items that are related to the system in three categories: elements that are part of the environment surrounding the system, outputs produced by the system, and inputs or tools used by the system. For example, in a library management system, the environment might include users and physical books, outputs could be issued book records or overdue notifications, and inputs might include barcode scanners or membership IDs. After everyone presents their lists, the team combines all inputs, removes duplicate items, and then breaks into smaller groups to create detailed mini-specifications. These are then consolidated to produce a preliminary draft of the complete system specification. This process ensures that the final documentation reflects a well-rounded understanding of requirements from multiple perspectives.

2.4.4.4 Quality Function Deployment

Quality Function Deployment (QFD) is a technique used in software engineering that focuses on ensuring customer satisfaction by prioritizing requirements that hold the most value for the user. This method classifies requirements into three categories based on how they influence customer perception. The first type, known as normal requirements, includes the essential functions and goals of the software as discussed directly with the customer. For example, in a result management system, normal requirements might include entering marks and calculating results. The second type, expected requirements, refers to those features that are assumed by the customer and are so basic that they may not be explicitly mentioned. An example of this would be ensuring that the system is protected from unauthorized access which is something users naturally expect. The third type, exciting requirements, covers features that go beyond user expectations and create a sense of delight when included. For example, if the system detects unauthorized access and responds by automatically backing up data and shutting down all processes, it would impress the customer by providing extra security. By addressing all three types of requirements, QFD ensures that the final product not only meets basic expectations but also adds value through thoughtful and innovative features.

2.4.4.5 Use Case Approach

The Use Case approach is a method that combines both written descriptions and visual diagrams to help better understand the requirements of a system. Unlike detailed technical specifications, use cases focus on what the system should do rather than how it does it, providing a clear functional overview. The main elements involved in use case design include actors, use cases, and use case diagrams. An actor represents any external entity such as a person or a machine that interacts with the system. For example, in an online shopping system, a customer could be a primary actor who uses the system to place orders, while a payment gateway could be a secondary actor that the system interacts with to process payments. Primary actors are those who require help from the system to achieve their goals, while secondary actors assist the system in performing certain functions. Use cases describe the sequence of actions between these actors and the system, capturing who does what and when. For example, a use case for the customer might be "Place Order," showing step-by-step interactions. A use case diagram visually represents these interactions: actors are shown as stick figures, use cases as ovals, and lines connect actors to their related use cases, illustrating relationships clearly. The effectiveness of this technique heavily depends on the experience and collaboration of everyone involved, including analysts, developers, users, and customers, to ensure all system functionalities are accurately captured.

2.4.5 Steps of Requirements Elicitation

Following are the Steps of Requirement Elicitation. See fig 2.4.2

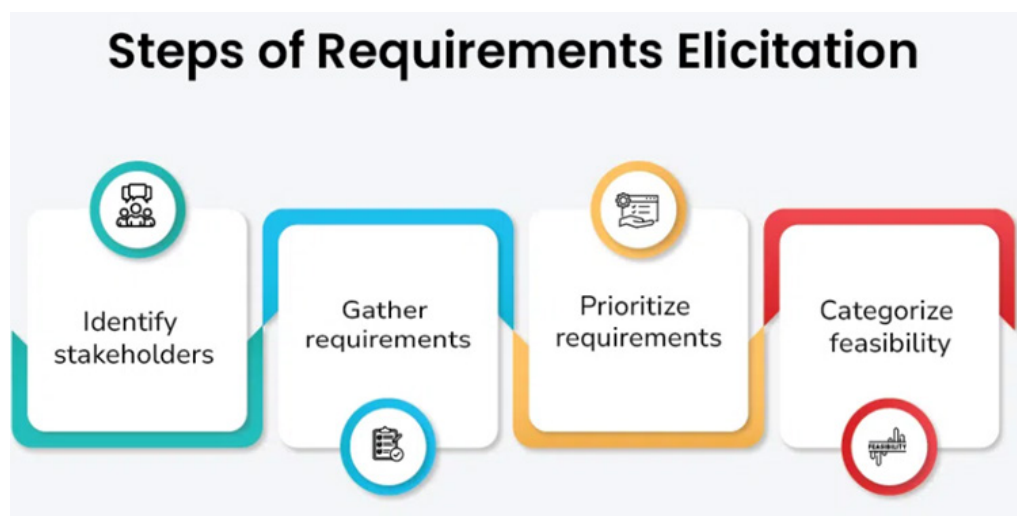


Fig 2.4.2 Steps of Requirements Elicitation

2.4.5.1 Identify Stakeholders

The initial step in the requirements elicitation process is to identify all the relevant stakeholders involved in the project. Stakeholders are individuals or groups who will either use the software, contribute to its development, or be affected by its implementation. This group can include end-users, software developers, project managers, clients, and other parties with an interest in the project's success. For example, in developing an

online shopping platform, stakeholders might range from customers browsing the site, to the developers building the system, and the marketing team planning promotions. Engaging all these stakeholders from the very beginning is crucial because it ensures that a variety of perspectives and needs are considered, leading to a more complete and accurate definition of the system's requirements. Early involvement helps avoid misunderstandings later in the project and promotes collaboration across different teams.

2.4.5.2 Gather Requirements

After identifying the stakeholders involved in a project, the next crucial step is to collect their requirements. This process involves gaining a clear understanding of what each stakeholder expects the system or product to achieve. Requirements can be divided into two main types: functional requirements, which describe the specific actions or tasks the system must perform, and non-functional requirements, which define how the system should behave or perform, such as speed, security, or usability. To effectively gather these requirements, various techniques can be employed, such as conducting one-on-one interviews to get detailed insights, distributing surveys to gather feedback from a larger group, or organizing group discussions where stakeholders can share and prioritize their needs collaboratively. For example, if developing an online shopping website, functional requirements might include the ability to add items to a cart and process payments, while non-functional requirements could involve ensuring fast page load times and secure handling of user data. By using these methods, developers can collect comprehensive and accurate requirements to guide the design and development process.

2.4.5.3 Prioritize Requirements

After collecting all the requirements, it's important to prioritize them because not every requirement holds the same level of importance. Prioritization helps focus efforts on the features that are most critical to the success of the system, especially when there are constraints such as limited time or budget. For example, a banking app must first include essential functions like secure login and transaction processing ("Must-have"), while additional features like personalized themes or advanced analytics may be considered less urgent ("Could-have"). A widely used method for prioritizing requirements is to classify them into categories such as "Must-have," which are essential for the system to function; "Should-have," which are important but not critical; "Could-have," which are nice to include if possible; and "Won't-have," which are not necessary for the current development cycle. This structured approach helps teams manage resources effectively and deliver the most valuable parts of the system first.

2.4.5.4 Categorize Feasibility

Once requirements are prioritized, the next step is to assess their feasibility by categorizing them as achievable, deferred, or impossible. Achievable requirements are realistic and can be implemented within available resources like adding a login feature within the timeline. Deferred requirements are valuable but postponed due to constraints like time or technical readiness, such as advanced analytics saved for future updates. Impossible requirements, such as supporting outdated hardware, are removed

due to feasibility issues. This categorization helps focus on realistic goals while managing expectations. Effective requirements elicitation supports this by involving stakeholders, gathering detailed system and user information, prioritizing based on impact, and documenting requirements clearly. Validation through stakeholder reviews ensures accuracy, and the process remains iterative, allowing for ongoing refinement and adaptation to changes. Close collaboration, regular communication, and flexibility make this a dynamic, ongoing activity crucial to delivering successful software solutions.

2.4.6 Benefits of Requirements Elicitation

Having clear and well-defined requirements is crucial because it helps to clarify and refine what the customer truly needs from the software. For example, if a client wants an online store, clear requirements specify features like payment options and delivery tracking, preventing confusion later. This clarity improves communication and collaboration among all stakeholders, including developers, clients, and users, ensuring everyone is on the same page. As a result, the likelihood of creating high-quality software that fulfills customer expectations significantly increases. Clear requirements also prevent misunderstandings that can cause costly delays or rework by setting realistic expectations from the start. Additionally, they help identify potential risks and challenges early in the development process, allowing the team to plan solutions in advance. This thorough understanding supports the creation of a detailed and accurate project plan, which keeps the development on track. When users see that their needs are understood and addressed, their confidence in the software and the development team grows. Moreover, well-defined requirements can reveal new business opportunities, such as adding features that open up new markets or revenue streams. Ultimately, clear requirements form the foundation for successful software development, benefiting both the creators and users of the system.

2.4.7 Drawbacks of Requirements Elicitation

The process of requirements elicitation can often be time-consuming and costly, as it involves thorough investigation and multiple interactions with stakeholders. It demands specialized skills and expertise to effectively gather and analyze the necessary information. Moreover, requirements elicitation is vulnerable to changes in business needs, which can alter the project scope and priorities mid-way. Political and organizational factors within a company can also influence the process, potentially causing delays or complications. When stakeholders are not fully committed or engaged, it may lead to a lack of buy-in, reducing the effectiveness of the elicitation effort. Conflicting priorities and interests among different groups can further complicate the identification of clear requirements. Sometimes, if the process is not carefully managed, the requirements collected may be incomplete or inaccurate, which can cause problems later in development. For example, if user needs are misunderstood, the final software might miss key features. Such issues often result in higher development costs and lower efficiency, as teams must spend extra time fixing errors or reworking parts of the system. Therefore, clear communication, proper planning, and active stakeholder involvement

are essential to overcome these challenges.

2.4.8 Software Requirement Specification (SRS) Document

A well-structured Software Requirement Specification (SRS) is essential for the successful development of any software system. The SRS serves as a comprehensive document that clearly outlines all the requirements the software must meet. These requirements include both functional aspects, such as specific features or actions the software should perform, and non-functional aspects, like performance, security, or usability criteria. To create an effective SRS, it is important to follow a clear format or structure, which often includes sections such as an introduction, overall description, detailed requirements, and other relevant information. This format helps organize the document in a way that is easy to understand and use. For example, in a banking application, a functional requirement might specify that users must be able to transfer money between accounts, while a non-functional requirement could state that transactions should be completed within two seconds. Communication and collaboration between customers and developers are vital throughout this process to ensure that the SRS accurately reflects the customers' needs and expectations. This ongoing interaction helps avoid misunderstandings and sets a solid foundation for building the software.

SRS Format includes

- ◆ Document Title
- ◆ Author(s)
- ◆ Affiliation
- ◆ Address
- ◆ Date
- ◆ Document Version etc.

Based on the information collected through interactions with users and stakeholders, a Software Requirements Specification (SRS) document is created. This document outlines the detailed requirements for the software, including any necessary changes or improvements that must be made to enhance the product's quality and meet the customer's expectations. For example, if during discussions it is found that the customer needs faster processing or additional features, these requirements are incorporated into the SRS to guide the development team. The SRS acts as a formal agreement that clearly defines what the software should do, helping ensure that the final product aligns with the customer's needs and delivers a high-quality solution.

2.4.9 Comprehensive Overview of Software Requirements Specification (SRS) Document

The Software Requirements Specification (SRS) document outlines all key aspects necessary for designing, developing, testing, and maintaining a software product. Each section in the SRS provides crucial information to ensure that the software aligns with

user expectations and technical standards. Below is a detailed and simplified explanation of each major component of the SRS document.

Purpose of the Document – This section outlines the primary reason for creating the document, clearly explaining its objective and necessity.

Scope of the Document – This part provides an overview of the document's function, highlighting its goals, the benefits it offers to the customer, and including estimates for development time and cost.

Overview – This section gives a brief summary of the product, offering a general insight into its features and intended functionality.

General Description

This section introduces the product in general terms. It highlights what the software is intended to do, who the users are, and why the software is important. It discusses the purpose and goals of the product from the user's point of view. Additionally, it provides an overview of the user community which describes who will use the system, what their characteristics are (such as age, skills, or education level), and what benefits the product offers to them. For example, if the product is an online education platform, the users may include students, teachers, and administrators, each with different needs and expectations from the system.

Functional Requirements

Functional requirements describe exactly what the software should do. This includes all operations and activities the system must perform. It outlines how the system responds to specific inputs and what outputs should be generated. These functions can include tasks like calculating results, processing user data, or displaying information. Each requirement must specify the data inputs, their sources, units of measurement, and valid input ranges. For example, in a banking app, a functional requirement might be: "The system must allow users to transfer money between accounts with real-time balance updates."

Interface Requirements

This section explains how the software interacts with other systems or users. Interfaces could be between two software programs or between a user and the software. It describes the form of communication such as APIs, graphical user interfaces (GUIs), or system messages. For example, a weather app might communicate with an external weather database via an API to display current weather data. The interface must ensure that data is accurately sent and received between different components.

Performance Requirements

Performance requirements define how well the software must perform under certain conditions. This includes speed, response time, memory usage, and acceptable error limits. These requirements help in measuring the system's efficiency and effectiveness. There are two types of performance requirements: static (do not depend on execution, like memory usage) and dynamic (depend on execution, like how fast the system

responds). For example, a stock trading platform might need to update stock prices every second with a maximum response time of 500 milliseconds.

Design Constraints

This section lists any limitations that the design team must consider while developing the software. Constraints could include the use of specific programming languages, hardware compatibility, or regulatory standards. For example, a software developed for a medical device must comply with strict health regulations and may need to use approved hardware components only. Such constraints limit flexibility but ensure that the product meets external requirements or client expectations.

Non-Functional Attributes

Non-functional requirements are qualities the system must have, beyond what it does functionally. These include attributes such as security (protection against unauthorized access), reliability (system should work without failure), portability (ability to run on different platforms), and scalability (capacity to grow with user base). For example, an e-commerce website must ensure data security for online payments and be reliable during high-traffic sales events.

Preliminary Schedule and Budget

This part provides an initial estimate of the time and cost required to complete the project. It outlines major milestones, expected delivery dates, and the financial resources needed. For example, the document might state that the project will be completed in six months with an estimated budget of \$50,000. This helps stakeholders plan and allocate resources effectively.

Appendices

The appendices section contains supplementary information that supports the document. This can include references to articles, books, or websites used, definitions of technical terms, acronyms, or abbreviations. For example, the term “API” might be defined as “Application Programming Interface” for clarity. This section ensures that all readers, regardless of their background, can understand the document fully.

This detailed structure ensures that the SRS document acts as a complete guide for developers, testers, and clients, setting clear expectations and minimizing confusion during the software development process.

2.4.10 Different Purposes of the SRS Document

◆ For the Development Team

The Software Requirements Specification (SRS) document is very important for the developers who build the software. It clearly tells them what features to include and how the product should behave. Without it, they may not know exactly what the customer wants. For example, if a company is building an online shopping website, the SRS will describe functions like "Add to Cart," "Secure Payment," and "Order Tracking," so the developers can create them correctly.

◆ For the Testing Team

The SRS helps the testing team create test plans to check if the software works properly. Since it describes how the software should behave from the outside (external behavior), testers can use it to make sure all the features perform as expected. If the SRS says users should be able to reset their password via email, the testers will create a test to see if that function works correctly.

◆ For Maintenance and Support Staff

People who maintain and support the software after it's released also use the SRS. It helps them understand what the software is designed to do, so they can fix bugs or help users more effectively. For example, if a user reports that the "Search" function is not working, the support staff can look at the SRS to check how that feature is supposed to work.

◆ For the Project Manager

Project managers use the SRS to plan the project, estimate how much time and effort will be needed, and decide on the required resources. It helps them break the project into smaller tasks and assign them to the team. For example, If the SRS shows that the software needs a complex reporting feature, the manager can plan for more time and assign experienced developers to handle it.

◆ For the Customer

Customers rely on the SRS to understand what the final software product will look like and what features it will include. It gives them a clear idea of what they are paying for. If a school is ordering a student management system, they can refer to the SRS to make sure it includes modules like attendance, grading, and report cards.

◆ As a Contract Between Developer and Customer

The SRS also acts like a formal agreement between the developers and the customer. Once both sides agree to the document, it becomes a reference for what the developers must deliver. If any disagreements occur later, they can use the SRS to resolve them. For example, if a feature is not included in the final product, but is clearly written in the SRS, the customer can ask for it to be added.

◆ For Documentation Purposes

The SRS is useful for documentation. It keeps a complete record of the system's requirements, which is helpful for future updates or changes. If a new team member joins the project later, they can read the SRS to quickly understand the software's goals and functions. For example, when a company updates its mobile app after a year, the developers can refer to the original SRS to maintain consistency.

Each of these uses highlights how the SRS document plays a key role in the success of a software project, by guiding different teams and ensuring everyone has a shared understanding of the final product.

Software development requires a well-structured Software Requirement Specification

(SRS). It helps stakeholders communicate, provides a roadmap for development teams, guides testers in creating effective test plans, guides maintenance and support employees, informs project management decisions, and sets customer expectations. The SRS document helps ensure that the software meets functional and non-functional requirements, resulting in a quality product on time and within budget.

Recap

- ◆ Requirements elicitation is the structured process of collecting and defining the needs of a software system from stakeholders.
- ◆ It aims to understand what the software must do (functional) and how it should perform (non-functional).
- ◆ Early stakeholder engagement reduces misunderstandings and costly changes later in the project.
- ◆ The process is communication-intensive and requires strong collaboration between developers and stakeholders.
- ◆ Requirements elicitation occurs early in the software development lifecycle.
- ◆ Stakeholders include clients, end-users, business owners, and technical experts.
- ◆ Techniques for elicitation include interviews, surveys, workshops, brainstorming, use cases, role-playing, and prototyping.
- ◆ It ensures alignment of software features with business goals and user expectations.
- ◆ Involving users increases software usability and user satisfaction.
- ◆ Clear requirements reduce development costs and shorten timelines by minimizing rework.
- ◆ In regulated industries, requirements elicitation helps ensure legal and compliance standards are met.
- ◆ Well-documented requirements improve traceability for testing, validation, and maintenance.
- ◆ Key activities include understanding the domain, analyzing customer problems, and defining system constraints.
- ◆ Prioritization of requirements (Must-have, Should-have, Could-have) helps manage scope and resources.
- ◆ Feasibility analysis categorizes requirements as achievable, deferred, or impossible.

- ◆ Requirements elicitation is iterative, involving continuous refinement as new feedback emerges.
- ◆ Active communication and collaboration throughout the process ensure shared understanding.
- ◆ Benefits include clearer customer needs, improved communication, risk identification, and opportunity discovery.
- ◆ Challenges include high time and cost investment, changing business needs, organizational politics, conflicting stakeholder interests, and risks of incomplete or inaccurate requirements.
- ◆ The SRS document lists important details like the title, author, date, and version, plus sections about what the software should do and how it should work.
- ◆ It helps everyone understand what the software needs to do, based on what users and customers want.
- ◆ It explains both the features the software must have and the qualities it should follow, like speed and security.
- ◆ Different teams, like developers, testers, and project managers, use the SRS to do their jobs well.
- ◆ The SRS is like a contract between the customer and developers and helps keep track of all the software requirements for future work.

Objective Type Questions

1. What is the primary goal of requirements elicitation?
2. Which phase in software development does requirements elicitation occur?
3. Name a common technique used to gather requirements by directly talking to stakeholders.
4. Which stakeholder group is involved in providing input during requirements elicitation?
5. What type of requirements describes how the system should perform rather than what it should do?
6. What is the name of the method that uses group sessions to generate ideas during elicitation?

7. Which approach uses actors and interactions to describe system requirements?
8. What is the term for evaluating whether a requirement can realistically be implemented?
9. What document is created to capture all the requirements in a structured format?
10. What is the key factor to ensure accuracy during requirements elicitation?
11. Which technique involves participants listing environment elements, inputs, and outputs?
12. What is the name of the prioritization category for absolutely essential requirements?
13. What is a major benefit of clear requirements elicitation related to project cost?
14. Which role is primarily responsible for managing group dynamics in brainstorming sessions?
15. What type of requirement focuses on system speed, security, or usability?
16. What process is iterative and involves continuous refinement of requirements?
17. What risk is reduced by engaging stakeholders early in the requirements elicitation?
18. What is the challenge called when developers misunderstand what customers expect?
19. What method helps link system components to their respective requirements?
20. What is an example of a non-functional requirement?
21. Which section of the SRS outlines the software's expected functions?
22. What type of requirements defines the system's speed and memory usage?
23. Who uses the SRS to create test plans?
24. What acts as a formal agreement between the developer and the customer in software projects?

Answers to Objective Type Questions

1. Understanding
2. Early
3. Interviews
4. All Stakeholders
5. Nonfunctional
6. Brainstorming
7. Usecase
8. Feasibility
9. SRS
10. Collaboration
11. FAST
12. Must have
13. Savings
14. Facilitator
15. Nonfunctional
16. Requirements Elicitation and Analysis
17. Misunderstandings
18. Expectation gap
19. Traceability
20. Performance
21. Functional
22. Performance
23. Testers
24. Contract

Assignments

1. Explain the concept of requirements elicitation and discuss why it is considered one of the most communication-intensive activities in software engineering.
2. Describe the significance of requirements elicitation in software development and how it contributes to aligning software solutions with business goals.
3. List and explain at least four key activities involved in the requirements elicitation process. Provide examples to illustrate each activity.
4. Discuss the different methods used for requirements elicitation, such as interviews, brainstorming sessions, and the Facilitated Application Specification Technique (FAST). How do these methods complement each other?
5. Explain the steps involved in requirements elicitation, starting from stakeholder identification to categorizing feasibility. Why is each step important for the overall success of the project?
6. What are the main features of effective requirements elicitation? How do these features help in producing accurate and reliable requirements documentation?
7. Analyze the benefits and drawbacks of requirements elicitation. How can the challenges of this process be mitigated to ensure successful software development?
8. Define what a Software Requirement Specification (SRS) document is. Discuss its importance and outline the typical structure of an SRS document.

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.

BLOCK 3

Software Design





Software Design

Learning Outcomes

At the end of this unit, the learner will be able to:

- ♦ define software design and its main levels.
- ♦ list the phases involved in the software design process.
- ♦ identify different types of module coupling.
- ♦ familiarise the principles of good software design.
- ♦ recall various types of module cohesion

Prerequisites

Imagine you are constructing a house. Before you start laying bricks or installing windows, you need a detailed plan or blueprint. This plan guides the construction process, showing where each element should go and how they will fit together. Similarly, when creating software, developers do not dive into coding right away. Instead, they begin with software design, which is like drawing a blueprint for the software.

Just as a house blueprint helps builders know where each part should be placed, software design helps developers understand how to structure the software. It involves planning how different parts of the software will function together, much like how the rooms, doors, and windows of a house are carefully planned. By taking time to plan, developers can create software that is easier to understand, more efficient, and simpler to improve or fix later.

In this unit, you will learn about the steps and phases of software design. You will see how breaking the software into smaller, manageable components makes it easier to develop, test, and maintain. Just as different parts of a house, like the kitchen, bathroom, and living room, are designed to fit together, the components of software must also work seamlessly together.

Keywords

Agents, Proto type Abstraction, Functional Independence Cohesion, Coupling

Discussion

3.1.1 Introduction to Software Design

Software design is the process of thinking and planning how a software application should work before writing the actual code. It helps developers understand how to organize the software, divide the work into smaller parts (like modules or components), and decide how these parts will work together. The goal of software design is to create a clear structure, so the software is easy to build, understand, and maintain. It involves both:

- ◆ High-level design (also called architecture): Deciding the major parts or components of the system and how they interact.
- ◆ Low-level design: Focusing on details of individual parts, like algorithms and functions.

Imagine you are going to build a calculator app that can perform basic operations like addition, subtraction, multiplication, and division. Before writing any code, you create a plan for how the app should work. In the high-level design, you decide that the app will have two main parts: a user interface, where the user enters numbers and sees the results, and a calculation engine, which performs the actual math operations. In the low-level design, you plan the specific functions that the calculation engine will include, such as add (a, b), subtract (a, b), multiply (a, b), and divide (a, b). This structured plan is called the Software Design, and it helps keep your work organized and easy to understand. It also makes it easier for others to collaborate on the project or make improvements in the future.

3.1.2 Software Design Process

The design phase in software development focuses on converting customer requirements, outlined in the Software Requirement Specification (SRS) document, into a format that can be implemented using a programming language. During this phase, developers carefully plan how to build the system. Rather than immediately starting to code, they analyze the complex requirements, break them down into simpler, manageable parts, and design the overall system structure. This process acts like a blueprint, helping developers determine how different components will be organized and how they will interact to form a functional software system.

The software design process can be categorized into the following three stages or levels of design as shown in Fig 3.1.1.

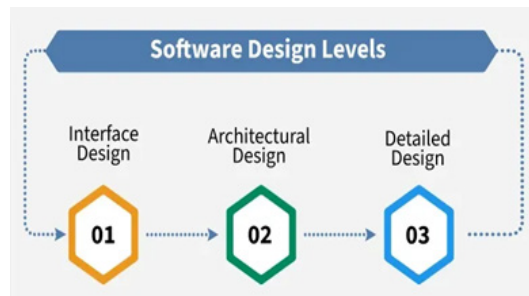


Fig 3.1.1 Software Design Levels

3.1.2.1 Levels of Design

1. Interface Design

Interface design defines how a system will interact with its external environment. At this stage, designers do not focus on the internal workings of the system instead, they treat the system as a "black box." The goal is to concentrate on how the system communicates with users, devices, and other systems, which are collectively known as Agents. This design focuses on the exchange of information between the system and these agents.

In an online food delivery application, the interface design focuses on how the system communicates with various agents such as the user (who places the order), the restaurant (which gets the order details), and the delivery person (who receives the pickup and delivery information). It does not deal with the internal processes of these agents, but rather with how information is exchanged through the app's interface.

2. Architectural Design

Architectural design defines the overall structure of a software system. It focuses on identifying the main components, what each one does, how they interact, and how they connect through interfaces. At this stage, the internal details of components are not considered. It builds on interface design by adding structure, but still avoids the deep internal design, which comes later.

In the architectural design for an online shopping platform, it is about planning the main parts of the system and how they work together. These parts include things like the user interface (for browsing and ordering), the product catalog, the shopping cart, the payment system, order tracking, and the database. At this stage, we focus on what each part does and how they communicate with each other, usually through APIs, without worrying about the detailed inner workings.

3. Detailed Design

Detailed design focuses on specifying the internal structure and behavior of each major system component. It involves breaking down components into smaller program units, assigning responsibilities to each unit, and designing user interfaces. This phase also defines the different states a unit can be in and how those states change, as well as how data and control signals are exchanged between units. Additionally, it includes

organizing data, managing scope and visibility of program elements, and selecting appropriate algorithms and data structures for implementation.

In an online library system, detailed design means planning how each part works inside. In the book borrowing system, it includes steps like checking if a book is available, recording who borrowed it, and setting a return date. It also decides how data like book lists or user info is stored and how rules like late fees are calculated.

3.1.3 Software Design Phases

Imagine you are designing a mobile app for a library. First, you talk to the librarians and students to understand what they need, like searching books or checking due dates. Then, you plan by doing surveys to know what features are most important. Next, you sketch simple screens (wireframes) showing how users will move through the app. After that, you create a technical plan that decides how the app will connect to the library database. You design a clean, easy-to-use interface so anyone can use it without confusion. Finally, you build a basic version (prototype) to test if everything works well before making the full app.

Software design phases are the structured steps followed during the design stage of software development to ensure that the final product is efficient, user-friendly, and meets all requirements. These phases help convert user needs into a working system plan.

The phases in software design are shown in Fig 3.1.2.



Fig 3.1.2 Design Phases

1. **Understanding Project Requirements:** The team first learns what users and the business need to ensure the design meets expectations.
2. **Research & Planning** – Information is gathered through interviews and surveys to better understand user needs.
3. **Designing the Software** – Visual tools like wireframes and flowcharts are created to plan how the system will work.
4. **Technical Design** – A detailed technical plan is developed, outlining how different parts of the system will function together.

5. **User Interface Design** – Focus is on creating a simple, user-friendly layout and navigation.
6. **Prototyping** – Early models of the software are built to test and refine ideas before full development.

3.1.4 Design Principles

Software design principles focus on helping developers manage the complexity of the design process more effectively. By keeping complexity under control, these principles not only make the design process easier but also help minimize the chances of errors. The principles of software design are shown in Fig 3.1.3.

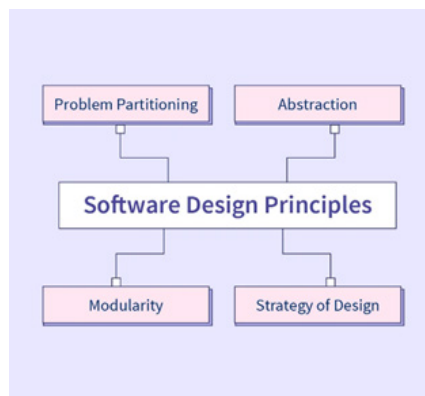


Fig 3.1.3 Software Design Principles

1. Partitioning

The problem partitioning principle in software design means breaking a big or complicated problem into smaller, easier parts. Instead of building the whole system at once, developers split it into separate sections or modules that are easier to work on one by one. This method makes the software:

- ◆ Easier to understand
- ◆ Easier to design, test, and fix
- ◆ Easier to update or add new features

In an online shopping website, the problem partitioning principle is applied by dividing the system into smaller parts like the user interface, product management, shopping cart, payment system, order management, and database. Each part can be developed and tested separately, making the system easier to build, understand, and maintain. Although these parts are separate, they work together to deliver a complete shopping experience.

2. Abstraction

Abstraction is a technique that allows a designer to focus on a component high-level functionality without getting into the details of its internal implementation. It can be applied both to existing elements and to components being designed. There are two main types of abstraction

◆ **Functional Abstraction**

A module is defined by the tasks it performs, while the details of how the tasks are carried out (the algorithm) are hidden from the user. Functional abstraction is key in function-oriented design approaches.

◆ **Data Abstraction**

The details of data elements are hidden from users, allowing them to interact with the data without needing to know how it is structured or stored. Data abstraction is fundamental to object-oriented design approaches.

Abstraction in a TV remote control allows you to change the channel or adjust the volume without needing to know how the remote works internally or how the TV processes the signals. You interact with the high-level functions, like pressing a button, without being concerned about the details of the technology behind it. Similarly, the TV hides the internal data about channels, allowing you to select a channel without knowing how it stores or processes the information.

3. Modularity

Modularity refers to breaking down software into separate, distinct modules that can be named and addressed individually. These modules are later combined to form a complete, functional system. This approach makes a program more manageable and easier to understand.

The key qualities of a modular system are:

- ◆ Each module is a self-contained unit that can work with other applications.
- ◆ Each module has a clear, specific purpose.
- ◆ Modules can be compiled separately and stored in a library.
- ◆ Modules should be easier to use than to create.
- ◆ From the outside, modules should be simpler to use than their internal workings.

4. Design Strategy

A good system design strategy involves organizing program modules in a way that makes them easy to develop and modify later. Structured design methods help developers manage the size and complexity of programs. Analysts provide instructions on how the code should be structured and how different parts of the code should integrate to form the complete system.

There are two main approaches to designing a system:

- 1. Top-down Approach:** This approach begins by identifying the main components of the system, then breaking them down into smaller, more detailed sub-components as shown in Fig 3.1.4.

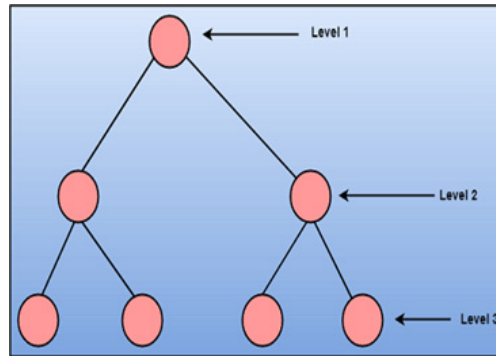


Fig 3.1.4 Top-down Approach

2. **Bottom-up Approach:** The bottom-up approach starts by focusing on the smaller, detailed components first and gradually builds up to higher-level parts of the system as shown in Fig 3.1.5. This method is especially useful when working with an existing system.

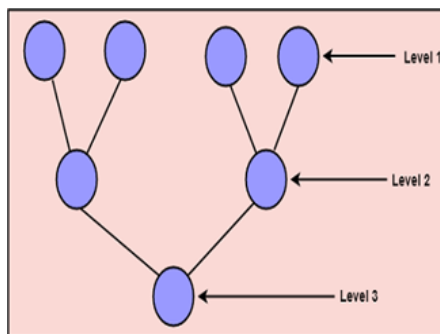


Fig 3.1.5 Bottom-up Approach

3.1.5 Modular Design

Modular Design helps simplify complex software systems by breaking them into smaller, manageable parts (modules). This makes development faster and easier since different parts can be developed at the same time.

1. Functional Independence

Each module in the system is designed to handle a specific task with minimal interaction with other modules. This makes the system easier to build, test, and maintain. Independent modules reduce the chance of errors spreading and can even be reused in other software.

Functional independence is evaluated using:

- ◆ Cohesion: How well the tasks within a module relate to each other.
- ◆ Coupling: How much a module depends on other modules.

2. Information Hiding

This principle means that a module hides its internal data and logic from other

modules, exposing only what is necessary. This protects one module's data from being accidentally altered by another, which is especially useful during testing and maintenance. Because modules do not share unnecessary details, updates can be made with fewer side effects on the rest of the system.

Think of building software like assembling a team to complete a big project. Instead of giving one person everything to do, you divide the work into smaller tasks and assign each to a different team member (module). Each person focuses only on their part without needing to know all the details of what others are doing this is functional independence. If someone does their job well (high cohesion) without depending too much on others (low coupling), the whole team works better. Also, each person keeps their own notes and methods private unless it is really needed this is information hiding. It helps avoid confusion and makes it easier to fix problems or make changes later without affecting the whole project.

3.1.5.1 Coupling

Coupling refers to how closely connected two software modules are as shown in Fig 3.1.6. If modules are tightly coupled, they rely heavily on each other to function. On the other hand, loosely coupled modules have minimal reliance on one another. Uncoupled modules work completely independently without any connection between them.

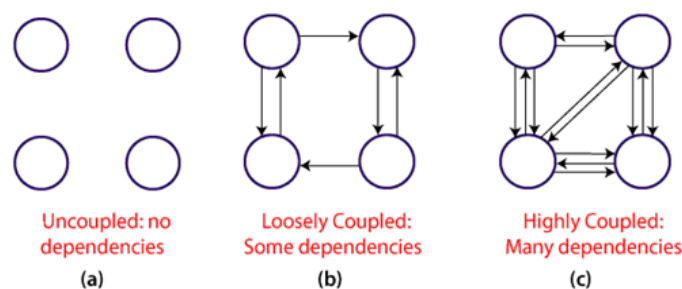


Fig 3.1.6 Coupling Techniques

A good software design aims for low coupling, meaning the modules interact as little as possible. Coupling is determined by how many connections or data exchanges exist between modules. When there are many calls between modules or a large amount of shared data, coupling becomes high. High coupling often leads to more errors in the system, making it harder to maintain and test.

Consider a pizza ordering system. In a tightly coupled system, a single person, say the cashier, takes the order, prepares the pizza, bakes it, and delivers it. If the cashier is unavailable, the whole system breaks down because everything depends on that one person. On the other hand, in a loosely coupled system, each task is handled by a different person: the cashier takes the order, the chef makes the pizza, and the delivery person delivers it. Each person only needs to know their own role and pass along necessary information. This way, if one person is changed or unavailable, the rest of the process can still continue smoothly. Similarly, in software, when modules are loosely coupled, they can function and be modified more easily without affecting the entire system.

Types Of Coupling

The various types of module coupling are shown in Fig 3.1.7.

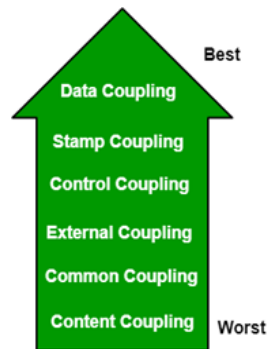


Fig 3.1.7 Types of Coupling

1. Content Coupling

This is the highest (worst) level of coupling. One module directly accesses or modifies the internal data or logic of another. This makes the system fragile and hard to maintain.

2. Common Coupling

Modules share the same global data. If one module changes the shared data, it can affect others. It reduces independence and makes debugging difficult.

3. External Coupling

Modules rely on external systems or interfaces to interact. This type of coupling is due to interactions with external hardware or software.

4. Control Coupling

One module controls the behavior of another by passing control information (like flags). This reduces the flexibility of the modules.

5. Stamp (Data Structure) Coupling

Only parts of a data structure are used by a module, but the entire structure is passed. This can lead to unnecessary dependencies.

6. Data Coupling

Modules share data by passing only the needed information. This is considered a good and low form of coupling.

7. No Coupling (Uncoupled)

Modules operate independently with no data or control shared. This is the ideal scenario for modular software.

3.1.5.2 Cohesion

Cohesion refers to how closely the elements within a single module are related to each other as shown in Fig 3.1.8. It indicates how well the functions or parts within

that module work together to perform a specific task. In a highly cohesive module, all the components are focused on a single purpose and are tightly connected. Cohesion is usually described in terms like "high cohesion" or "low cohesion," and it is considered better to have high cohesion for better organization, easier maintenance, and improved code quality.

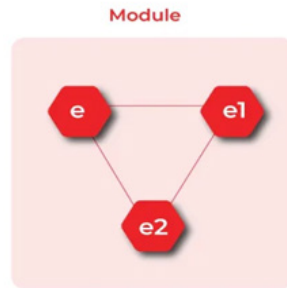


Fig 3.1.8 Cohesion

Types of Modules Cohesion (from weakest to strongest)

The different types of Cohesion are shown in Fig 3.1.9

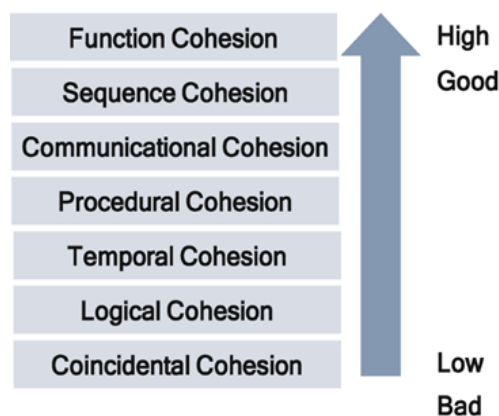


Fig 3.19 Types of Cohesion

1. Coincidental Cohesion

A module exhibits coincidental cohesion when it carries out a collection of tasks that are only weakly related or possibly not related at all. These tasks are grouped together without a clear or meaningful connection between them.

Example: function that prints the date, logs in a user, and calculates tax—all in one place, without any logical connection between the tasks.

2. Logical Cohesion

Tasks that are logically categorized to do the same thing, even if they are different by nature. Example: A single function that handles all types of user input (mouse, keyboard, voice) based on a control flag.

3. Temporal Cohesion

Elements are grouped because they are executed at the same time. Example: Initialization functions that start up different unrelated systems when a program launches.

4. Procedural Cohesion

Elements are grouped because they always follow a specific sequence of execution.

Example: Function that reads a user's input, validates it, and then stores it in a file—all steps follow a specific sequence to complete a related task.

5. Communicational Cohesion

Tasks that operate on the same data or contribute to the same output. Example: A module that reads a record from a database, processes it, and formats the result for display.

6. Sequential Cohesion

The output from one part of a module is the input for the next.

Example: A function that reads user input, processes the data, and then displays the result, each step using the output of the previous one.

7. Functional Cohesion

Every element in the module works together to perform a single, clearly defined task. Example: Function that calculates the area of a circle given its radius; every part of the function contributes directly to that single purpose.

Recap

- ◆ Software design is the planning step taken before any coding begins.
- ◆ It includes both high-level design (system architecture) and low-level design (detailed components).
- ◆ The design process transforms software requirements into a structured system plan.
- ◆ There are three main levels of design: interface design, architectural design, and detailed design.
- ◆ Software design follows several phases including understanding requirements, research, planning, UI design, and prototyping
- ◆ Software design follows several phases including understanding requirements, research, planning, UI design, and prototyping.
- ◆ Key design principles include partitioning, abstraction, modularity,

and the use of structured strategies.

- ◆ Top-down and bottom-up are two approaches to designing software systems.
- ◆ Modular design divides software into independent modules to simplify development and maintenance.
- ◆ Coupling measures how closely connected different modules are, with lower coupling being better.
- ◆ Types of coupling include content, common, external, control, stamp (data structure), data, and no coupling.
- ◆ Cohesion measures how well tasks within a module relate to each other, with high cohesion being desirable.
- ◆ Types of cohesion include coincidental, logical, temporal, procedural, communicational, sequential, and functional cohesion.

Objective Type Questions

1. What is the first step before coding in software development?
2. What type of design focuses on system structure?
3. What type of design focuses on internal module details?
4. What design principle involves breaking problems into smaller parts?
5. Which principle hides internal details from other modules?
6. What design principle encourages using self-contained units?
7. Which approach starts design from the top components?
8. What is the preferred level of coupling in software design?
9. Which type of cohesion is the strongest?
10. Which type of coupling is the weakest (best)?
11. What type of cohesion groups tasks that happen at the same time?
12. What type of coupling involves sharing global data?
13. What type of cohesion uses the output of one task as input for the next?
14. What type of coupling directly accesses another module's code?
15. What type of cohesion has unrelated tasks grouped together?

Answers to Objective Type Questions

1. Software Design
2. Architectural Design
3. Detailed Design
4. Partitioning
5. Information Hiding
6. Modularity
7. Top-down Approach
8. Low Coupling
9. Functional Cohesion
10. No Coupling
11. Temporal Cohesion
12. Common Coupling
13. Sequential Cohesion
14. Content Coupling
15. Coincidental Cohesion

Assignments

1. Explain the different levels of software design with suitable examples.
2. Discuss the phases of the software design process and how they help in building a user-friendly and functional system.
3. What is modular design? Explain the concepts of functional independence, cohesion, and coupling with examples.
4. Differentiate between the Top-Down and Bottom-Up approaches in software design. In what situations would each be preferred?
5. Identify and explain the different types of cohesion and coupling in software design. Why is high cohesion and low coupling desirable?

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.



Software Design Strategies

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ identify the different software design strategies used in software development.
- ◆ list the characteristics and tools used in structured design.
- ◆ define the process of designing a user interface.
- ◆ recognize the types of user interfaces and their characteristics.

Prerequisites

Imagine you are about to build an app. You know its features, but how do you organize everything before coding? Jumping into coding without a plan can lead to a messy system. This is where software design strategies come in, they help you break down complex problems, organize functions, and structure your app effectively.

If you have worked with functions, you know how they perform specific tasks, but organizing them well is key. With object-oriented design, you can structure your app around real-world objects, like customers and products, ensuring they work well together. And when it comes to the user interface, you have seen apps that are easy to use and others that are confusing. A good UI design makes your app intuitive and user-friendly.

Studying these strategies will help you plan your projects effectively, ensuring they are easy to develop, maintain, and scale, while also providing a smooth experience for users.

Keywords

Structured Design, Function-Oriented Design, Object-Oriented Design, User Interface Design, Modular Design, System Architecture

Discussion

3.2.1 Strategies of Software Designing

Software design strategies are systematic methods or approaches used by developers to plan how a software system should be structured and built before actual coding begins. These strategies help break down complex problems into manageable parts, define how different components will interact, and ensure that the system meets its requirements efficiently and is easy to maintain.

They guide decisions about:

- ◆ How to organize the software into modules or components
- ◆ How to handle data and functions
- ◆ How parts of the system communicate with each other

Various strategies used in software design are

- ◆ Structured Design
- ◆ Function Oriented Design
- ◆ Object Oriented Design
- ◆ User Interface Design

3.2.1.1 Approaches of Software Design

Choosing the right strategy means picking the best way to design software based on what the project needs. Some projects may need a quick solution, while others may need to be flexible or easy to update later. By looking at the project's goals and size, developers can decide whether to use structured, function oriented, or another method that works best.

1. Structured Design

Structured Design is a method of solving complex problems by breaking them down into smaller, organized parts. It focuses on designing a clear and effective solution by dividing the problem into manageable modules. This approach improves understanding of how the problem is addressed and helps designers focus more accurately on each part of the solution.

Structured design uses the "divide and conquer" method splitting the main problem into smaller sub-problems. Each of these smaller problems is solved separately using

individual solution modules. These modules are arranged in a hierarchical structure and interact with one another to complete the overall system. To ensure efficiency, structured design follows key principles:

- ◆ Cohesion: Ensures that all related functions are grouped within the same module.
- ◆ Coupling: Manages the communication between different modules.

Good design aims for high cohesion and low coupling to make systems easier to maintain and understand.

Characteristics of Structured Design

1. Problem is broken down into smaller, manageable parts.
2. Uses a top-down approach for system development.
3. Emphasizes logical structure and modular design.
4. Promotes reusability and easier debugging.
5. Encourages clear documentation and better project understanding.
6. Supports independent testing of each module.

Tools Used in Structured Design

1. Data Flow Diagrams (DFDs) – to show the flow of information through the system.
2. Structure Charts – to represent the hierarchical arrangement of modules.
3. Pseudocode – for outlining logic in plain language before coding.
4. Flowcharts – to visualize the flow of control in the system.

A simple example of structured design can be seen in a Library Management System. In this approach, the system is broken down into main modules such as User Management, Book Management, and Transaction Management. Each of these modules is further divided into smaller sub-modules; for example, User Management may include functions like adding, updating, and deleting users, while Book Management handles adding new books, updating book information, and searching for books. Transaction Management covers issuing books, returning them, and checking fines. This structure follows the divide and conquer principle, where each module focuses on a specific part of the system, making the design more organized and easier to understand, develop, and maintain. It also ensures high cohesion within modules and low coupling between them, which are key characteristics of a good, structured design.

2. Function Oriented Design

In function-oriented design, the entire system is broken down into smaller units called Functions. Each function performs a specific, meaningful task, and the system is viewed as a collection of these interconnected functions from a top-level perspective.

This design approach builds on the principles of structured design, especially the "divide and conquer" method, which simplifies complex systems by dividing them into manageable parts.

The system is divided into smaller functional modules, and each module hides its internal details, offering abstraction. These modules can communicate with one another by sharing information or using globally accessible data.

One important feature of this approach is that when a function is called, it may alter the state of the system, which can sometimes lead to issues if other modules are sensitive to state changes. Therefore, function-oriented design is most suitable when the system is input-driven rather than state-dependent.

Design Process:

- ◆ The system is analyzed in terms of how data moves through it, typically shown using a Data Flow Diagram (DFD).
- ◆ DFDs show how functions transform input data and affect the overall system.
- ◆ The system is logically broken down into smaller functional units based on the tasks they perform.
- ◆ Each function is then explained in detail

A Payroll Processing System is a function-oriented design where the system is broken into functions like collecting data, calculating salary, and generating pay slips. Each function handles a specific task and shares data as needed. The focus is on how data flows and is processed, not on maintaining system state.

3. Object Oriented Design (OOD)

Object-Oriented Design focuses on the entities (objects) and their properties, rather than just the functions of a software system. The entire software solution is built around the objects involved in the system and how they behave.

Key Concepts of Object-Oriented Design

- ◆ **Objects:** These are the core components in the design. Objects represent real-world entities such as a person, bank, company, or customer. Each object has its own attributes (data) and methods (actions it can perform).
- ◆ **Classes:** A class is a blueprint or general description for objects. An object is a specific example of a class. The class defines what attributes and methods an object will have.
- ◆ **Encapsulation:** This means combining the attributes and methods into a single unit (object) and protecting them from outside interference. This is also known as information hiding, because it restricts direct access to the internal details of the object.
- ◆ **Inheritance:** This allows a class to use or inherit attributes and methods from another class. A subclass can extend the features of a superclass, making it

easy to reuse code and create new, more specific classes from existing ones.

- ◆ **Polymorphism:** This feature lets different objects respond to the same method name in different ways. The correct method is chosen based on the number or type of arguments. It helps in writing flexible and reusable code.

Design Process in Object-Oriented Design

1. **Requirement Analysis:** The design is created based on the system requirements or existing models like system sequence diagrams.
2. **Object Identification:** Objects are recognized and grouped into classes based on their shared attributes.
3. **Class Relationships:** The relationships between different classes are established, including how they inherit or interact with one another.
4. **Framework Design:** A general application framework is built to organize and support the structure of the system.

Imagine an Online Shopping System. In this system, entities like Customer, Product, Order, and Shopping Cart are represented as objects. Each object has attributes (like product name, price, or customer address) and methods (like addToCart or placeOrder). These objects are grouped into classes, and inheritance is used—for instance, different types of users like Admin and Buyer can inherit from a common User class. This design focuses on modeling real-world entities, promotes code reuse through inheritance, and keeps data secure using encapsulation.

4. User Interface Design

User Interface (UI) Design is the process of creating the visual layout and interactive elements of a software application or website that users interact with. It focuses on making the user's experience simple, efficient, and enjoyable when using a product.

In UI design, designers decide how things like buttons, menus, icons, and screens look and behave. The goal is to make the interface easy to understand, visually appealing, and user-friendly.

Characteristics

- ◆ It defines how users interact with the system.
- ◆ It involves elements like layout, colors, fonts, icons, and responsiveness.
- ◆ It aims to improve usability, accessibility, and overall user satisfaction

Types of User Interfaces

1. Graphical User Interface (GUI)

Involves visual elements like windows, icons, menus, and buttons. It is user-friendly and widely used in modern applications.

2. Command Line Interface (CLI)

A text-based interface where users interact by typing commands. It is powerful but requires technical knowledge.

3. Touch User Interface

Used in touch-enabled devices like smartphones and tablets. It relies on gestures like tap, swipe, and pinch.

4. Voice User Interface

Allows users to interact through voice commands. Common in virtual assistants like Siri and Alexa.

Consider a Mobile Banking App. In this app, the user interface is designed to allow customers to easily check their account balance, transfer funds, pay bills, and view transaction history. The main screen might include buttons for each feature, with clear labels and icons for easy navigation. Input fields are provided for entering amounts or account numbers, and there are confirmation dialogss to ensure users are aware of important actions (like transferring funds). The interface follows principles like consistency (similar elements are used across screens), simplicity (only necessary information is shown), and responsiveness (adapts well to different screen sizes). This design ensures that users can interact with the system easily, efficiently, and with minimal confusion.

Recap

- ◆ Software Design Strategies:
 - These are methods used by developers.
 - They help plan how a software system should be structured before coding.
 - These strategies help break down complex problems.
 - They also ensure the system is easy to maintain.
- ◆ Approaches of Software Design:
 - The right design strategy is chosen based on the project's needs.
 - It is also based on the project's goals.
 - Common approaches include Structured Design, Function-Oriented Design, Object-Oriented Design, and User Interface Design.

- ◆ Structured Design:
 - This design focuses on breaking problems into smaller parts.
 - It uses a top-down approach.
 - It encourages high cohesion and low coupling between modules.
 - Tools like Data Flow Diagrams and Flowcharts are used to visualize the system.
- ◆ Function-Oriented Design:
 - This design divides the system into smaller functions.
 - These functions perform specific tasks.
 - The system focuses on how data moves through it.
 - The system is usually input-driven, not state-dependent.
- ◆ Object-Oriented Design (OOD):
 - This design organizes the system around objects.
 - Objects represent real-world entities.
 - Key concepts include objects, classes, encapsulation, inheritance, and polymorphism.
 - This design promotes code reuse and flexibility.
- ◆ User Interface (UI) Design:
 - This design focuses on creating user-friendly interfaces.
 - These interfaces should be visually appealing.
 - It involves designing buttons, icons, and layout elements.
 - The goal is to enhance usability and user satisfaction.
 - Types of UIs include Graphical User Interface (GUI), Command Line Interface (CLI), Touch User Interface, and Voice User Interface.

Objective Type Questions

1. What method helps developers plan the structure of a software system?
2. Which design approach divides the problem into smaller, manageable parts?
3. Which design strategy focuses on real-world entities and their behaviors?
4. What is used in structured design to show the flow of information?
5. What approach uses a top-down strategy for system development?
6. What design strategy is input-driven, focusing on data movement?
7. What principle ensures that related functions are grouped within the same module?
8. What type of interface involves visual elements like icons and menus?
9. What design principle helps in reducing dependency between modules?
10. Which design approach involves organizing components into classes and objects?

Answers to Objective Type Questions

1. Software Design Strategies
2. Structured Design
3. Object-Oriented Design
4. Data Flow Diagrams
5. Top-down
6. Function-Oriented Design
7. Cohesion
8. Graphical User Interface
9. Low Coupling
10. Object-Oriented Design

Assignments

1. Explain the concept of Structured Design. What are its key characteristics and tools used? Support your answer with a suitable example.
2. Describe the Function-Oriented Design approach. How does it differ from Structured Design? Illustrate with a real-world system.
3. Define Object-Oriented Design and explain its key concepts. Provide an example to support your explanation.
4. What is User Interface Design? Discuss its importance, characteristics, and the different types of user interfaces with suitable examples.
5. Compare and contrast the four software design strategies. When would you use each strategy?

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.



Design Approaches

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ define the top-down approach in software design.
- ◆ list the main characteristics of the top-down approach.
- ◆ identify the advantages of the bottom-up approach.
- ◆ recall the steps involved in stepwise refinement.

Prerequisites

Imagine you are helping your friend organize a school event. You do not start by setting up the stage right away. First, you make a plan, decide what the event is about, list the tasks like arranging chairs, preparing a speech, inviting guests, and then assign who does what. You break a big task into smaller, manageable steps. You have probably done something like this in real life, maybe while planning a birthday party, a group project, or even a trip with friends.

In programming, we do something very similar. When building software, developers do not write all the code at once. Instead, they plan it out, break it into smaller pieces, and decide which parts to build first. This method of designing a system step-by-step is what we call a design approach.

Keywords

Top-Down Approach, Bottom-Up Approach, Stepwise Refinement, Modularization, Procedural Abstraction

Discussion

3.3.1 Approaches in Software Design

Approaches in software designing refer to the methods used to plan and structure a software system. These approaches guide how a system is broken down into parts and built up into a complete solution.

The two main approaches are the top-down approach, which starts with the overall system and breaks it into smaller parts, and the bottom-up approach, which begins with detailed components and integrates them into a full system. Choosing the right approach helps ensure the software is well-organized, efficient, and easier to develop and maintain.

3.3.2 Top-Down Approach

The Top-Down Approach is a method of problem-solving and system design where the system is viewed as a whole, and then broken into smaller sub-systems or modules as shown in Fig 3.3.1. The process continues recursively until each module is simple enough to be implemented and understood independently.

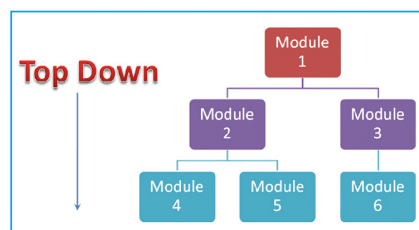


Fig 3.3.1 Top-Down Approach

In a Student Report System, at the top level, the problem is to generate a student's report. This is broken down into major tasks such as inputting student details, entering marks, calculating results, and displaying the report. Each of these tasks is further divided into smaller sub-tasks—for example, calculating results includes computing total marks, average, and assigning grades. This process continues until each task can be directly translated into programming statements. The top-down approach helps in organizing complex problems into manageable steps, making development, and debugging easier.

3.3.2.1 Characteristics of Top-Down Approach

- ◆ **High-Level to Low-Level Design:** Starts from the general system and moves toward more specific details.
- ◆ **Modularization:** Divides the system into functional modules.
- ◆ **Abstraction:** Initially focuses on what each part should do rather than how it does it.
- ◆ **Step-by-Step Refinement:** Each module is refined step-by-step until it is

ready for coding

3.3.2.2 Process of Top-Down Approach

a. System Overview

Start by understanding the complete system's purpose and the major functions it needs to perform. Create a general outline of the entire software.

b. Decomposition into Modules

Break down the system into large components or modules based on major functions (e.g., User Management, Payment Processing).

c. Further Refinement

Each large module is divided into smaller sub-modules. This continues until the modules are simple and independent enough to be coded.

d. Design Specification

Each final module is defined in detail. Inputs, outputs, processes, and data flow for each are described clearly.

3.3.2.3 Advantages of Top-Down Approach

- ♦ **Clarity and Organization:** Makes it easier to understand the structure of the whole system.
- ♦ **Early Detection of Design Flaws:** Design issues can be identified before implementation.
- ♦ **Improved Modularity:** Easier to develop, test, and maintain each module independently.
- ♦ **Promotes Team Collaboration:** Different modules can be assigned to different teams.

3.3.2.4 Disadvantages of Top-Down Approach

- ♦ **Less Focus on Low-Level Details Initially:** Important implementation details may be overlooked in early stages.
- ♦ **May Require Frequent Redesigns:** As the design becomes more detailed, early assumptions may need to be revisited.
- ♦ **Not Suitable for All Projects:** In some agile or rapidly changing environments, a more flexible, bottom-up approach may be better.

3.3.3 Bottom-Up Approach

In a calculator application, you might first create and test small functions like add (), subtract (), multiply (), and divide (). Once these are working properly, you integrate them into a full calculator system with a user interface.



The bottom-up approach in software design starts by developing the basic or low-level components first. These small, detailed modules are tested and combined to form larger, more complex systems as shown in Fig 3.3.2. This method is especially useful when reusable components or existing codebases are available.

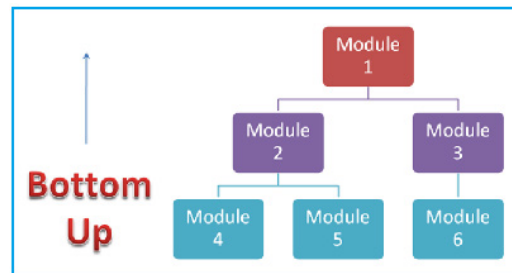


Fig 3.3.2 Bottom-Up Approach

3.3.3.1 Characteristics of Bottom-Up Approach

In this approach, development begins with:

- ◆ Independent modules or functionalities
- ◆ These modules are then integrated upward
- ◆ The process continues until the complete system is formed

It is like building a house starting with bricks and gradually forming walls, rooms, and finally the whole structure.

3.3.3.2 When to Use Bottom-Up Approach

This approach is best used when:

- ◆ Existing components or libraries are available
- ◆ The system evolves from previously developed systems
- ◆ Developers are more familiar with how individual components work

3.3.3.3 Advantages of Bottom-Up Approach

- ◆ **Reusability:** Makes use of existing components
- ◆ **Reliability:** Well-tested components increase system stability
- ◆ **Faster Prototyping:** Small modules can be tested early
- ◆ **Parallel Development:** Different modules can be developed simultaneously

3.3.3.4 Disadvantages of Bottom-Up Approach

- ◆ **Lack of Big Picture:** May lose focus on the overall system structure early on

- ♦ **Integration Challenges:** Assembling modules may require extra effort
- ♦ **Difficult for New Systems:** Not ideal if the full system design is not known

3.3.4 Stepwise Refinement

Stepwise refinement is the process of developing a program by repeatedly breaking it down into more detailed steps as shown in Fig 3.3.3. Each step adds more information and gets closer to an actual implementation in a programming language.

Think of planning a vacation. At first, your goal is simply: “Plan a vacation.” That is the high-level idea. Next, you refine it: “Book flights, reserve a hotel, create an itinerary.” Then you go even deeper: for booking flights, you refine further “Compare ticket prices, choose dates, enter passenger details, make payment.” Each step becomes more detailed as you move forward. This is how stepwise refinement works in software design, starting from a broad concept and breaking it into manageable, detailed steps until the solution is fully defined and ready to be implemented.

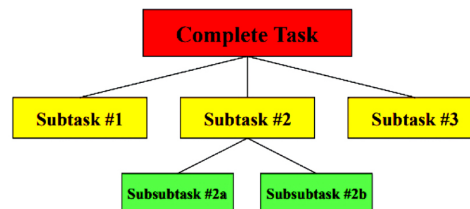


Fig 3.3.3 Stepwise Refinement

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

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth

3.3.4.1 Process of Stepwise Refinement

1. **Begin with High-Level Abstraction**-Start with a macroscopic view of the function. This is often called a Procedural Abstraction. At this stage, the focus is on what the system should do, not how it should do it.
2. **Decompose into Sub-Functions**-Break the high-level function into sub-functions or modules, each addressing a part of the overall functionality. This helps form a hierarchical structure of procedures.
3. **Successive Refinement**-Each sub-function is then refined further, adding detail in a step-by-step manner, until the system design reaches the level of actual programming language statements.

3.3.4.2 Advantages of Stepwise Refinement

- ◆ Improves clarity and understanding of system functionality.
- ◆ Simplifies problem-solving by focusing on smaller parts.
- ◆ Supports modular and structured programming.
- ◆ Allows early detection of design flaws.
- ◆ Helps in writing reusable and maintainable code.

Recap

- ◆ Software design approaches help plan and organize a software system effectively.
- ◆ Two main methods are Top-Down and Bottom-Up approaches.
- ◆ The Top-Down Approach starts with the overall system and breaks it into smaller parts.
- ◆ It helps simplify complex problems by dividing them into manageable modules.
- ◆ It uses modularization, abstraction, and step-by-step refinement.
- ◆ The process involves: understanding the system, breaking it into modules, refining modules, and writing detailed specifications.
- ◆ Benefits include better organization, early error detection, easier maintenance, and teamwork.
- ◆ Drawbacks include less initial focus on low-level details and possible redesign needs.
- ◆ The Bottom-Up Approach starts with building small components first, then combining them into the full system.
- ◆ Useful when you have existing code, libraries, or reusable components.
- ◆ It supports early testing, reuse, and parallel development.
- ◆ Challenges include difficulty in seeing the overall system early and integration issues.
- ◆ Stepwise Refinement breaks problems into smaller steps gradually, adding more detail at each stage.

- ◆ Starts with a high-level view, breaks into sub-functions, and refines until code-ready.
- ◆ Advantages of stepwise refinement include better clarity, easier debugging, modularity, and maintainable code.

Objective Type Questions

1. What approach breaks a system from general to specific?
2. Which approach begins with building individual modules?
3. What is the design technique that uses abstraction and refinement?
4. Which approach helps in early detection of design flaws?
5. What is the first step in Stepwise Refinement?
6. What term describes splitting a system into smaller units?
7. What is the key benefit of modular design?
8. Which approach promotes faster prototyping?
9. What concept focuses on what a module does, not how?
10. What helps expose lower-level details during design progression?

Answers to Objective Type Questions

1. Top-Down
2. Bottom-Up
3. Refinement
4. Top-Down
5. Abstraction
6. Modularization
7. Maintainability
8. Bottom-Up
9. Abstraction
10. Refinement

Assignments

1. Differentiate between the Top-Down and Bottom-Up approaches in software design. Provide suitable examples for each.
2. Explain the process of the Top-Down approach in software design. What are its main characteristics, and what are the advantages and disadvantages of using this approach?
3. Describe the Bottom-Up approach. Under what circumstances is this approach more beneficial than the Top-Down approach?
4. What is Stepwise Refinement in software design? Explain the process with a suitable example, and list its advantages.
5. How do abstraction and refinement complement each other in the context of software design? Explain with reference to stepwise refinement.

Suggested Reading

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

Reference

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.
3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.



Software Analysis and Design Tools

Learning Outcomes

At the end of this unit, the learner will be able to:

- ◆ Define software analysis and design.
- ◆ List the tools used in software analysis and design.
- ◆ Identify the symbols used in Data Flow Diagrams (DFDs) and ER models.
- ◆ Recall the components of a Data Dictionary.
- ◆ Familiarise the types of Data Flow Diagrams and Mapping Cardinalities.

Prerequisites

Think of designing software like building a house. You would not start by randomly placing bricks, wires, and pipes. First, you would meet with an architect, create a blueprint, and decide where everything should go. This plan ensures that the house is strong, functional, and meets the needs of those who will live in it. Similarly, software development begins not with coding, but with thoughtful planning. This planning phase uses Software Analysis and Design Tools to shape ideas into clear, structured models.

For instance, a Data Flow Diagram serves as a roadmap, visually tracing how data moves through a system—where it comes from, where it goes, and how it is processed. It is like tracing water pipes in a building plan. A Context Diagram zooms out to give you the big picture. It treats the entire system as a single process and shows how it interacts with users or other systems like showing your house on a street map and identifying your neighbors. Then there is the Data Dictionary, a crucial tool that acts like a detailed catalog or user manual. It explains every piece of data used in the system: what it represents (such as "student ID" or "exam date"), its format (text, number, date), and its origin. This helps everyone from designers to testers stay on the same page. Using these tools ensures that all stakeholders understand the system before any development begins. It reduces errors, saves time, and results in software that is more aligned with user needs.

Keywords

Data Flow Diagram, Structure Chart, Decision Table, Entity Relationship Model, Cardinality

Discussion

3.4.1 Introduction to Software Analysis and Design

Software analysis and design act as an essential link between user-friendly requirement specifications and the actual coding process. These specifications define both what the software should do (functional) and how well it should perform (non-functional), but they are not directly executable by machines. Through the analysis and design phase, these requirements are organized into a systematic structure suitable for development. This step is a key part of the software development lifecycle, aiding in the clear understanding, planning, and modeling of the system. Using analysis and design tools ensures the final software meets user expectations, performs reliably, and upholds quality standards.

Some commonly used tools for software analysis and design are

- ◆ Data Flow Diagram
- ◆ Structure Charts
- ◆ Decision Table
- ◆ Entity Relationship Model
- ◆ Data Dictionary

3.4.2 Data Flow Diagram

A Data Flow Diagram (DFD) visually illustrates how data moves within a system. It provides an easy-to-understand graphical representation that is useful for both programmers and end-users. Using symbols like arrows, circles, and rectangles, a DFD shows how data is input, processed, stored, and output across different parts of the system. This visualization aids developers, stakeholders, and users in understanding system requirements and designing process flows effectively.

3.4.2.1 Types of Data Flow Diagrams

- ◆ **Logical DFD:** Focuses on what the system does, mapping out data movement and processes without including technical or physical aspects. It highlights high-level functions and data exchanges.
- ◆ **Physical DFD:** Emphasizes how the system operates in reality, including devices, files, databases, and manual procedures. It reflects actual data

handling and implementation details.

3.4.2.2 Key Elements of a DFD

- ◆ **Entities:** Entities act as the origin or destination for data and are illustrated using rectangles labeled with their names.
- ◆ **Process:** Any operations or actions performed on data are shown using circles or rectangles with rounded edges.
- ◆ **Data Storage:** There are two common symbols for data storage—either an open rectangle missing its two short sides or a rectangle with one open side.
- ◆ **Data Flow:** Arrows represent the direction of data movement. The tail of the arrow indicates where the data comes from, and the head shows where it is going.

Symbols used for the elements of DFD is shown in Fig 3.4.1.

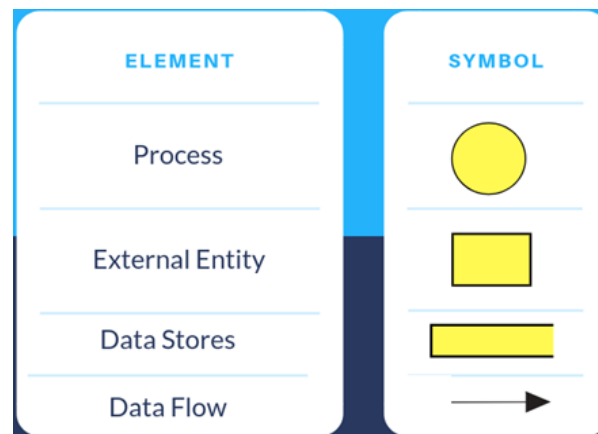


Fig 3.4.1 DFD Symbols

3.4.2.3 Levels of DFD

1. **Level 0 (Context Diagram):** Provides a broad overview of the system, treating it as a single process with its interactions shown via data flowing to and from external entities as shown in Fig 3.4.2.



Fig 3.4.2 Level 0

2. **Level 1 DFD:** Breaks down the main process from Level 0 into sub-processes, giving a clearer picture of how the system functions internally as shown in Fig 3.4.3.

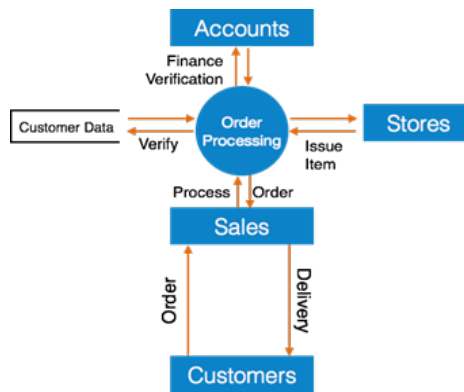


Fig 3.4.3 Level 1

3. **Level 2 DFD:** Further refines the Level 1 processes into more detailed sub-processes, offering a deeper look into the system's inner workings

3.4.3 Structure Charts

A structure chart is developed from a Data Flow Diagram and offers a more detailed view of the system. It breaks the entire system down into its smallest functional units and provides an in-depth description of each module's functions and sub-functions, offering more detail than a DFD.

The structure chart displays the system in a hierarchical format, where each level performs a specific function or task. Symbols used in structure charts are

1. **Module-** A module refers to a process, task, or subroutine within a system as shown in Fig 3.4.4. A control module directs the flow to multiple sub-modules. Library modules are reusable components that can be accessed and called from any other module in the system.

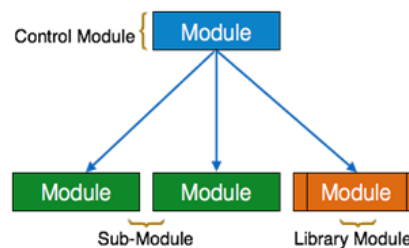


Fig 3.4.4 Module

2. **Condition-** It is shown using a small diamond shape at the bottom of a module as shown in Fig 3.4.5. This indicates that the control module can choose one of the subroutines to execute depending on a specific condition.

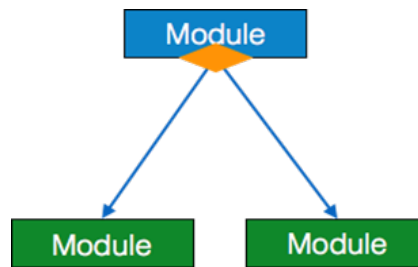


Fig 3.4.5 Condition

3. **Jump**- An inward-pointing arrow toward a module is used to indicate that control will enter the sub-module at an intermediate point rather than at the beginning as shown in Fig 3.4.6.

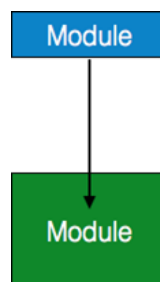


Fig 3.4.6 Jump

4. **Loop**- A curved arrow signifies a loop within the module, indicating that all the sub-modules enclosed by the loop will be executed repeatedly as shown in Fig 3.4.7.

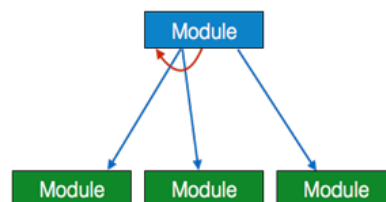


Fig 3.4.7 Loop

5. **Data Flow**-Data flow is represented by a directed arrow ending in an open circle, indicating the movement of data between module as shown in Fig 3.4.8.

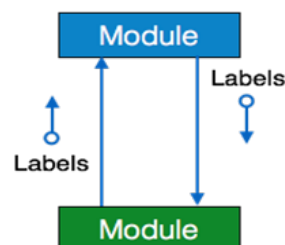


Fig 3.4.8 Data Flow

6. **Control Flow-** It is shown using a directed arrow that ends with a filled circle, signifying the transfer of control between modules as in Fig 3.4.9.

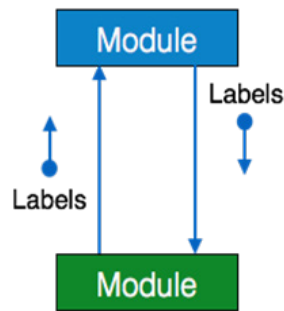


Fig 3.4.9 Control Flow

3.4.4 Decision Table

A decision table organizes conditions and their corresponding actions in a clear, tabular layout.

It serves as an effective tool for identifying and avoiding errors, allowing similar information to be grouped together. By combining such tables, it simplifies and streamlines the decision-making process.

3.4.4.1 Creating a Decision Table

To build a decision table as in Fig 3.4.10, a developer should follow these four basic steps:

1. List all potential conditions that need consideration.
2. Identify the actions that should be taken for each condition.
3. Generate all possible combinations of rules based on the conditions.
4. Assign the appropriate action(s) to each rule.

After creation, the decision table should be reviewed by end-users and can later be refined by removing any duplicate rules or repeated actions.

IF	Decision Rule 1	Decision Rule 2	Decision Rule 3	ELSE
Condition 1	Y	>40		
Condition 2	N	<100		
Condition 3	N	<0		
THEN				
Action 1	Y	N		N
Action 2	N	Set State to "fail"		Y

Fig 3.4.10 Decision Table

A Decision Table is read vertically, where each column represents a decision rule. The top row of conditions is read as starting with "IF," and each following condition as "AND." The first row of actions begins with "THEN," followed by "AND" for additional actions.

For example, Rule 1 would read:

IF Condition 1 is true,

AND Condition 2 is false,

AND Condition 3 is false,

THEN perform Action 1.

3.4.5 Entity Relationship Model

The Entity-Relationship (ER) model is a type of database model that represents real-world entities and the relationships between them. It helps in translating real-world scenarios into a structured database framework. The ER model defines a collection of entities, their attributes, constraints, and the relationships connecting them.

This model is especially useful for the conceptual design phase of a database and is typically illustrated using diagrams for better visualization and planning. ER Model can be represented as in Fig 3.4.11

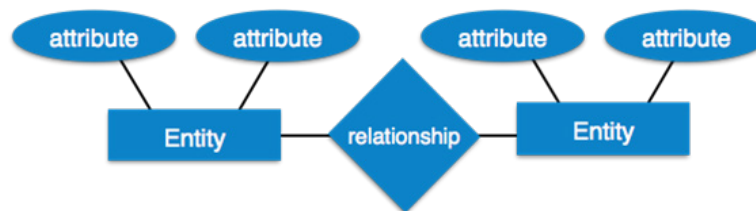


Fig 3.4.11 E R Diagram

3.4.5.1 Components of E R Model

◆ Entity:

An entity in an Entity-Relationship (ER) model represents a real-world object or concept that can be distinctly identified. Each entity possesses certain characteristics known as Attributes, and every attribute has a specific set of possible values, referred to as its Domain.

In a school database, a student would be considered an entity. This student entity may have attributes such as name, student ID, age, and class.

◆ Relationship:

A relationship defines the logical link or association between two or more entities. These relationships illustrate how entities interact with one another. The number of connections between entities is described by Mapping Cardinalities.

3.4.5.2 Types of Mapping Cardinalities

- ◆ **One-to-One:** One entity is related to only one entity of the other type.
- ◆ **One-to-Many:** One entity is related to multiple entities of the other type.

- ♦ **Many-to-One:** Many entities are linked to a single entity of the other type.
- ♦ **Many-to-Many:** Multiple entities are associated with multiple entities from another type.

3.4.5.3 Symbols Used in ER Model

E R symbols are standardized graphical notations used in ER diagrams to visually represent the structure of a database. The symbols are shown in Fig 3.4.12

Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

Fig 3.4.12 E R Model Symbols

3.4.6 Data Dictionary

A data dictionary is a centralized repository that contains detailed information about data, such as definitions, formats, and relationships. It helps both users and analysts understand how data is structured, used, and processed within a system. The data dictionary clarifies the meaning of system model terms and offers insights into the system's inputs, outputs, storage, and computing processes.

3.4.6.1 Key Components

- ♦ **Data Elements:** These include primary and secondary names, descriptions of content and control items, as well as internal and external data stores.
- ♦ **Data Store:** Refers to the location where data is held, which may include files, tables, or other storage systems, representing data that enters or leaves the system.
- ♦ **Data Processing:** Involves handling and transforming data, and can be:
 - Logical Processing: Performed at the user or application level.
 - Physical Processing: Happens at the system/software level and is typically hidden from users.

Table 3.4.1 shows the data dictionary for a Student Management System

Data Element	Description	Data Type	Size	Source
Student_ID	Unique ID assigned to a student	Integer	6	Student Registration
Name	Full name of the student	String	50	Student Input Form
Age	Age of the student	Integer	2	Student Input Form
Class	Current class/grade	String	10	Student Input Form
Teacher_ID	Assigned teacher's ID	Integer	6	Internal Assignment

This dictionary helps developers and users understand each data field clearly, including where it comes from, its type, and usage.

Recap

- ◆ Software analysis and design serve as the bridge between user requirements and actual coding, helping structure the system for development.
- ◆ Tools like DFDs, structure charts, decision tables, ER models, and data dictionaries are used to ensure the software meets quality and user expectations.
- ◆ A Data Flow Diagram (DFD) graphically shows how data moves through a system, aiding both developers and users in understanding processes.
- ◆ Logical DFDs show what the system does, while Physical DFDs illustrate how the system operates in real-world terms.
- ◆ DFDs use elements like entities, processes, data stores, and data flows to represent system components.
- ◆ Level 0 DFD (Context Diagram) gives an overall view of the system, while Levels 1 and 2 break it into more detailed sub-processes.
- ◆ A Structure Chart provides a hierarchical breakdown of the system into modules, showing control and data flow.

- ◆ Symbols in structure charts include modules, conditions, jumps, loops, data flow, and control flow indicators.
- ◆ A Decision Table presents conditions and related actions in a tabular form to support clear and error-free decision-making.
- ◆ An Entity Relationship (ER) Model represents real-world objects and their relationships in a structured database framework.
- ◆ Entities have attributes and domains, while relationships define interactions between entities.
- ◆ Mapping cardinalities in ER models include one-to-one, one-to-many, many-to-one, and many-to-many relationships.
- ◆ Standard ER diagram symbols visually represent entities, attributes, and relationships.
- ◆ A Data Dictionary acts as a repository of information about data elements, stores, and processing methods.
- ◆ It includes components like data elements, data stores, and details about logical and physical data processing.

Objective Type Questions

1. Which diagram provides a high-level view of the entire system as a single process?
2. What shape is typically used to represent an external entity in a DFD?
3. What chart displays the system in a hierarchical module-based format?
4. What symbol is used in a structure chart to represent a decision point?
5. What type of DFD focuses on real-world implementation?
6. What type of diagram shows relationships between database entities?
7. What document contains definitions and formats of data elements?
8. What symbol in DFD represents the movement of data?
9. What is used to describe all possible combinations of decisions and outcomes?
10. What term is used for the smallest functional unit in a structure chart?

Answers to Objective Type Questions

1. Context Diagram
2. Rectangle
3. Structure Chart
4. Diamond
5. Physical DFD
6. Entity Relationship Diagram
7. Data Dictionary
8. Arrow
9. Decision Table
10. Module

Assignments

1. Explain the role of software analysis and design in the software development lifecycle. Why is this phase critical before starting the actual coding process?
2. Differentiate between a Logical DFD and a Physical DFD. Provide examples to illustrate how they are used in system design.
3. Describe the key symbols and components used in a Data Flow Diagram (DFD). How do these elements help in understanding system processes?
4. What is a Structure Chart? How does it differ from a Data Flow Diagram, and what are the main symbols used in a Structure Chart?
5. Discuss the components of an Entity-Relationship (ER) Model. Explain different types of mapping cardinalities with examples.

Suggested Reading

1. Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice Hall.
2. Fairley, R. E. (2009). *Managing and Leading Software Projects*. Wiley.

3. Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2002). *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall.
4. van Vliet, H. (2008). *Software Engineering: Principles and Practice* (3rd ed.). Wiley.
5. McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction* (2nd ed.). Microsoft Press.

Reference

1. Pressman, R. S., & Maxim, B. R. (2020). *Software Engineering: A Practitioner's Approach* (9th ed.). McGraw-Hill Education.
2. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson Education.
3. Jalote, P. (2005). *An Integrated Approach to Software Engineering* (3rd ed.). Springer.
4. Pfleeger, S. L., & Atlee, J. M. (2010). *Software Engineering: Theory and Practice* (4th ed.). Pearson.
5. Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley

BLOCK 4

Coding and Testing





Coding standards and Code review

Learning Outcomes

The learners will be able to:

- ◆ recall and list the key principles and guidelines of coding practices.
- ◆ explain the importance of adhering to coding standards in software development.
- ◆ apply appropriate coding standards to write clean, readable, and maintainable code.
- ◆ analyze a given piece of code to identify deviations from established coding standards.

Prerequisites

Consider the development of an online food delivery app. Through coding, the design is transformed into essential features like restaurant listings, order placement, and real-time delivery tracking. Meanwhile, testing ensures that each of these features operates smoothly, orders are processed accurately, payments are secure, and the app remains stable without crashes. Together, effective coding and testing contribute to the overall success, quality, and sustainability of software products.

Testing and coding are two essential pillars in software development. Coding transforms design ideas into functional software by writing clear, efficient, and maintainable code to build the app's features. Testing, on the other hand, ensures that the coded software works correctly, reliably, and securely. It identifies bugs, checks performance, and verifies usability before the product reaches users. Together, effective coding and thorough testing guarantee a high-quality, stable, and user-friendly software experience.

Standardization helps maintain clarity, efficiency, and consistency in various professional tasks like writing, communication, file management, and organization. Following common standards ensures that everyone understands

the processes clearly, reducing confusion and improving teamwork. It also increases efficiency and minimizes errors. This applies to coding as well, where standardized practices in syntax, structure, and documentation make code more readable, easier to debug, and simpler to maintain. Overall, using standard approaches in both administrative and technical work leads to better accuracy and reliability.

Keywords

Efficiency, Modularity, Reuseability, Scalability, Code Integration, Code Revision, Post-Merge Validation.

Discussion

4.1.1 Introduction to Coding Standards and Code Review

Imagine a team of developers working together to build a banking application. Since multiple developers are writing different parts of the code, it's important that everyone follows the same style and rules. This is where coding standards come in. For example, the team may decide that all variables should use meaningful names like `accountBalance` instead of short names like `ab`, and that every function must have clear comments explaining what it does. They might also have rules for proper indentation, error handling, and avoiding shortcuts like hardcoding values. These standards make the code easier to read and understand, not just for the original developer, but for anyone who works on it later.

Before the code is finalized, the team performs a code review. For instance, when one developer writes the feature to calculate loan interest, another team member reviews the code to check if it follows the agreed standards, contains no logical errors, handles all edge cases (like zero interest rates), and has no security issues. The reviewer might suggest improvements, such as optimizing a loop or adding extra validation for user input.

By following coding standards and conducting code reviews, the banking app becomes more reliable, secure, and easier to maintain. This process reduces the risk of bugs, improves performance, and ensures that the application meets the high-quality standards required for handling sensitive financial data.

4.1.2 Coding

In Software Engineering, coding refers to the process of writing instructions, also known as source code, using a specific programming language (such as Python, Java, C++, etc.) to create software programs. It is the step where the software design is translated into a working product. Coding is also known as programming in software



engineering. In software engineering, coding is one of the core activities that turns software design into a working system. It is a crucial phase in building applications, websites, or systems that meet user needs and requirements.

Coding involves implementing the logic, algorithms, and features described in the design documents. Developers write code to handle tasks like user input, data processing, database interactions, and output display. It ensures the software performs the desired functions efficiently, reliably, and securely.

4.1.2.1 Role of Coding in Software Engineering

In software engineering, coding plays a vital role in turning a system design into real functional software. Imagine a team working on a banking application that allows users to check account balances, transfer money, and pay bills online. After the design phase, where all the features and user interfaces are carefully planned, the developer starts coding, writing precise instructions in a programming language to implement those features. This code tells the system exactly what to do and how to do it, such as verifying user credentials, processing transactions, or generating account statements. In this way, coding acts as a bridge between the bank's requirements (like secure transactions and real-time balance updates) and the final product that customers use. If the code is efficient and well-organized, it helps the software run smoothly, reduces errors, and makes testing and future maintenance easier and less expensive. Writing clean, readable code also allows other developers to understand and enhance the system over time, ensuring the banking software remains secure, reliable, and scalable as customer needs grow.

4.1.2.2 Steps Involved in the Coding Phase

Coding, in essence, is a structured process of converting an idea or a problem into a series of instructions a computer can understand and execute. It involves several key steps, from understanding the problem to testing and debugging the solution. Here's the different steps involved in coding using a simple example, developing a calculator program that adds two numbers:

a. Problem Definition and Analysis

The first step is to clearly define and understand the problem you want to solve. This involves identifying the required inputs, expected outputs, and any special conditions or constraints. Suppose we need a calculator that takes two numbers and returns their sum. Here inputs are two numbers and the output is the sum of those two numbers.

b. Planning and Design

After understanding the problem, the next step is to carefully plan the solution. This involves outlining the steps needed and deciding on the best approach to tackle the problem. A good strategy is to break down the main problem into smaller, manageable tasks, making it easier to solve each part individually. You also need to choose suitable algorithms and data structures that can efficiently handle each task. To visualize the logic clearly before coding begins, it's helpful to create flowcharts or write pseudocode, providing a clear blueprint for the program's structure and flow. In our calculator

example,

- Get two numbers from the user.
- Add them.
- Display the result.
- You might draw a simple flowchart or write pseudocode like:

Start → Input two numbers → Add them → Output result → End.

c. Choosing a Programming Language

Select the most suitable programming language based on the project's nature, application type, and personal or team expertise. For a basic calculator, we could use Python because it's simple and quick for small programs.

d. Writing the Code

In this phase, the actual coding takes place. Using the chosen language, you convert the planned design into functional instructions that the computer can execute.

Example in Python:

```
num1 = int(input("Enter first number: "))  
num2 = int(input("Enter second number: "))  
sum = num1 + num2  
print("The sum is:", sum)
```

e. Testing and Debugging

After coding, the program must be tested thoroughly to find and fix errors or bugs. This includes unit testing, integration testing, and sometimes user acceptance testing. Run the program with different inputs, like 2 and 3, or 10 and -5 to check if it adds numbers correctly. If there's any mistake like input type errors, fix it.

f. Documentation

Good documentation is essential. It explains how the code works, making it easier for others and your future self to understand, maintain, and update the program.

g. Maintenance and Optimization

Even after deployment, the code may need updates. Maintenance includes fixing bugs, improving performance, and adding new features as user needs evolve. you might want to improve the calculator by adding more functions like subtraction, multiplication, or handling invalid inputs.

h. Deployment

Finally, the software is deployed to a production environment, such as a server or app store, making it accessible for users to interact with.

4.1.2.3 Goals of Good Coding in Software Engineering

Good coding aims to create software that is reliable, maintainable, and adaptable. Different Goals are:

- a. **Correctness:** Correctness is essential because code must function exactly as intended and meet all the defined requirements without errors.
- b. **Readability:** Readability means code should be clear, well organized, and easy for others (or the same developer later) to understand and update.
- c. **Efficiency,** meaning the code should use system resources wisely and perform tasks quickly without unnecessary processing.
- d. **Reusability** is also vital, well written code should be modular so that components can be reused in other parts of the project or even in future projects.
- e. **Scalability** ensures that the code can handle increased loads or be extended with new features without requiring a complete rewrite. Together, these goals lead to high-quality, professional software development.

4.1.2.4 Characteristics of Programming Language

For implementing our design into code, we require a high level functional language. A programming language is needed to communicate with computers, as they only understand binary (0s and 1s). It acts as a translator that allows humans to write instructions in a readable and logical form, which the computer can then execute. Programming languages simplify complex tasks, increase productivity, and help solve real-world problems efficiently. They also make it easier to write code that works across different devices and systems.

A programming language should have the following characteristics:

- a. **Readability:** A good high level language will allow programs to be written in some methods that resemble a quite English description of the underlying functions. The coding may be done in an essentially self-documenting way.
- b. **Portability:** High level languages, being virtually machine independent, should be easy to develop portable software.
- c. **Generality:** Most high level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.
- d. **Brevity:** Language should have the ability to implement the algorithm with less amount of code. Programs in high level languages are often significantly shorter than their low level equivalents.
- e. **Error checking:** A programmer is likely to make many errors in the

development of a computer program. Many high level languages invoke a lot of bugs checking both at compile time and run-time.

- f. **Cost:** The ultimate cost of a programming language is a task of many of its characteristics.
- g. **Quick translation:** It should permit quick translation.
- h. **Efficiency:** It should authorize the creation of an efficient object code.
- i. **Modularity:** It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self consistency among these modules.
- j. **Widely available:** Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

4.1.3 Coding Standards

Coding standards are a set of rules and guidelines that developers follow when writing code. These standards define how the code should be formatted, named, structured, and documented, ensuring consistency, readability, and quality across a software project.

In a software project like a banking application, following coding standards is extremely important for clarity and teamwork. Imagine a situation where different developers are writing code without any agreed upon rules: one developer names a variable `Amt`, another writes `amount To Be Transferred`, and another simply uses `a`. Even though all three mean the same thing, the inconsistency makes the code confusing, hard to read, and difficult to maintain.

To solve this problem, coding standards define a common format that every developer must follow. For example, the team might agree that all variables should use camelCase naming and descriptive names so everyone uses `amount To Transfer` consistently throughout the project. This way, the code becomes uniform, readable, and easier to debug or update later.

By enforcing such standards, it also becomes easier for new team members to quickly understand the codebase without getting confused by different styles. In large projects like banking systems, where accuracy, security, and long term maintenance are critical, coding standards are not just helpful, they are essential.

4.1.3.1 Key Aspects of Coding Standards

Coding standards are essential guidelines that promote clarity, consistency, and quality in software development. They help teams write code that is easy to read, maintain, and scale. Key aspects include:

1. **Naming Conventions:** Consistent and meaningful naming is essential for writing clear and understandable code. Using descriptive names for variables, functions, and classes helps others (and your future self) easily grasp what the code does. For example, using `customer Name` is far more

informative than using a vague abbreviation like `cn`. Following naming rules like camelCase or snake case uniformly across the project improves overall readability.

2. **Indentation and Formatting:** Proper indentation and formatting make the code visually organized and easier to navigate. Indenting code blocks inside loops, conditionals, and functions helps highlight the logical structure of the program. Consistent formatting also prevents confusion, especially when multiple developers work on the same codebase.
3. **Commenting and Documentation:** Adding comments to explain complex sections of code is a good practice, especially for logic that isn't immediately obvious. In addition to inline comments, proper documentation for each module and function should describe its purpose, input, and output. This makes the codebase more maintainable and easier for others to understand or extend.
4. **Error Handling:** Effective error handling is a key part of coding standards. Developers should use consistent methods to catch and manage errors or exceptions, ensuring the program can handle unexpected situations gracefully without crashing. This improves the robustness and reliability of the software.
5. **Code Structure:** A well-structured codebase follows principles like modularity and separation of concerns. This means breaking down the program into smaller, reusable functions or modules, each handling a specific task. Such structure not only improves readability but also makes testing, debugging, and future updates much easier.

4.1.3.2 Coding Guidelines in Software Engineering

Coding guidelines provide general recommendations on coding style to enhance the readability, understandability, and maintainability of software. These practices help developers write clean and consistent code, especially in collaborative environments.

Here are some key coding guidelines:

- ◆ **Keep the code simple and easy to follow:** Avoid overly complex coding styles, as they make programs harder to debug and maintain. Readable code saves time and effort during updates and troubleshooting.
- ◆ **Use meaningful identifiers for specific purposes:** Every variable or function should have a clear and descriptive name that reflects its purpose. Reusing the same identifier for different tasks can confuse readers and make future improvements more difficult.
- ◆ **Document the code properly:** Comments should be included to explain important sections or logic within the code. Well-documented code enhances understanding, especially for those reviewing or maintaining it later.
- ◆ **Keep functions short and focused:** Long functions are often difficult to read and debug. It's better to break down large functions into smaller ones, each responsible for a single, specific task.

- ◆ **Avoid using the GOTO statement:** Using GOTO can lead to unstructured, confusing code. It hampers readability and makes debugging more challenging. Structured control flow (like loops and conditionals) should be used instead.

4.1.4 Code Review

Code review is a systematic process in which a team of developers collaboratively examines another developer's code. The goal is to identify errors, provide constructive feedback, and ensure that the code adheres to established standards and guidelines. This process helps improve the overall quality, stability, maintainability, and security of the software, leading to better project outcomes. Additionally, code reviews encourage knowledge sharing and learning within the team, fostering a culture of collaboration and continuous improvement.

4.1.4.1 Purpose of Code Review

Code review is an essential practice in software development, carried out for several important reasons. First and foremost, it helps identify bugs, errors, and issues early in the development cycle before the code reaches production making fixes easier and less costly. It also encourages developers to write clean, efficient, and maintainable code, as reviewers provide feedback to ensure adherence to coding standards and best practices.

By following consistent coding patterns across the team, code reviews enhance code readability and maintainability, making the entire codebase easier to understand and manage. Additionally, they serve as a platform for knowledge sharing, allowing teams to spread domain knowledge and reinforce coding guidelines across projects.

Code reviews promote a sense of shared ownership among team members, as reviewers become partly responsible for the quality of the code. This collective responsibility improves software reliability. The process also encourages collaboration and refinement, helping teams continuously improve their development practices.

Moreover, code reviews can contribute to project documentation, since comments and discussions often provide valuable insight into the logic and design choices. Ultimately, they are a key part of quality assurance, ensuring the software meets both functional and non functional requirements. Over time, regular reviews lead to continuous improvement within the team, making the development process more effective and mature.

4.1.4.2 Types of Code Review

Code reviews can be conducted through various methods, each suited to different team structures, project needs, and development workflows. These include formal inspections with structured meetings, informal peer reviews, over the shoulder reviews where a developer walks a peer through the code, email pass-around where code is shared and discussed asynchronously, and modern collaborative tools such as pull requests on platforms like GitHub or GitLab. Each method has its own strengths, and selecting the appropriate one depends on factors like team size, code complexity, and project deadlines.

The types of code reviews are listed below

- ◆ **Pull Requests (PR):** In Git, the developers raise a PR to incorporate changes to the code. It should be reviewed prior to the changes being merged with the base code.
- ◆ **Pair Programming:** It is a type of review in which two developers work on the same computer. One of them writes the code and the other one reviews it in real time. It is a highly interactive form of code review.
- ◆ **Over the Shoulder Review:** It is the type of review in which one developer in the team is requested to review the code of another developer by sitting together and going through the code on the computer.
- ◆ **Tool Aided Reviews:** It is a type of review conducted by tools like Github, GitLab, BitBucket, Crucible etc.
- ◆ **Email Based Reviews:** It is a type of review in which the code changes sent over email for review. The feedback of the code review is also delivered in email.
- ◆ **Checklist Reviews:** It is a type of review in which the reviewers follow the list of checklist items for the review process.
- ◆ **Ad Hoc Review:** It is an informal way of review. A developer may be requested to have a quick look at the code and provide feedback not formally.
- ◆ **Formal Inspection:** It is a type of review in which an already existing process is followed. It is mostly done by an inspection team and is guided by proper documentation.

4.1.4.3 Different Steps of code review

Code review is a structured process that ensures the quality and reliability of code before it becomes part of the final software. It typically follows a series of well defined steps:

Step 1: Submission for Review

Once a developer finishes writing a piece of code, they initiate the review process by notifying the team or submitting a review request using a version control system or code review tool. This marks the beginning of the code evaluation phase, allowing peers to examine the code for quality, functionality, and adherence to standards.

Step 2: Reviewer Assignment

Reviewer assignment involves selecting one or more team members to examine the submitted code. These reviewers are typically chosen based on their expertise, experience, and familiarity with the project to ensure a thorough and effective evaluation.

Step 3: Access and Setup

In the access and setup phase, reviewers utilize suitable tools or integrated development environments (IDEs) to open and navigate the codebase. This setup ensures they can efficiently review the code, understand its context, and provide constructive feedback.

Step 4: Use of Checklists

To ensure consistency and thoroughness during the review process, reviewers often use predefined checklists or coding guidelines. These checklists typically cover aspects such as naming conventions, code formatting, error handling, security practices, and overall code structure.

Step 5: Thorough Inspection

Reviewers inspect the code for logical errors, syntax mistakes, performance issues, and potential security flaws. This ensures the code is stable, scalable, and efficient.

Step 6: Providing Feedback

During the feedback phase, reviewers provide comments, suggestions, and observations directly within the review tool. This feedback should be clear, respectful, and constructive, aiming to help the author understand the issues and make necessary improvements effectively.

Step 7: Developer Collaboration

Developer collaboration involves open communication between the developer and reviewers to clarify feedback, explain design choices, and reach a consensus on the required changes. This dialogue helps ensure mutual understanding and leads to more effective code improvements.

Step 8 : Code Revisions

Code revisions are made by the developer in response to the reviewers' feedback. The updated code is then resubmitted for further review, and this cycle may be repeated as needed until all issues are resolved and the code meets the required standards.

Step 9 : Approval for Merge

Once the reviewers are satisfied with the quality and completeness of the revised code, they grant approval for it to be merged into the main codebase. This indicates that the code meets all required standards and is ready for integration.

Step 10 : Code Integration

Code integration involves merging the reviewed and approved code into the main codebase using version control systems such as Git. This step ensures that the new code becomes part of the overall project and is accessible to all team members for further development or deployment.

Step 11 : Post-Merge Validation

Post-merge validation involves running tests after the code is integrated into the main codebase. This ensures that the new changes function as expected and do not introduce any issues or regressions that could impact existing features or functionality.

Step 12: Documentation

Documentation involves recording the entire review process, including comments, decisions, and suggestions, for future reference. This comprehensive record is valuable for audits, helps onboard new team members, and supports continuous improvement by providing insights into past decisions and review practices.

Recap

- ◆ Coding : coding refers to the process of writing instructions, also known as source code, using a specific programming language (such as Python, Java, C++, etc.) to create software programs.
- ◆ Steps Involved in the Coding Phase
 - Problem Definition and Analysis
 - Planning and Design
 - Choosing a Programming Language
 - Writing the Code
 - Testing and Debugging
 - Documentation
 - Maintenance and Optimization, Deployment
- ◆ Goals of Good Coding in Software Engineering :Good coding aims to create software that is reliable, maintainable, and adaptable. Different Goals are: Correctness, Readability, Efficiency, Reusability, Scalability.
- ◆ Coding standard : Coding standards are a set of rules and guidelines that developers follow when writing code. These standards define how the code should be formatted, named, structured, and documented, ensuring consistency, readability, and quality across a software project.
- ◆ Key Aspects of Coding Standards: Naming Conventions, Indentation and Formatting, Commenting and Documentation, Error Handling, Code Structure
- ◆ Code review : Code review is a systematic process in which a team of developers collaboratively examines another developer's code.

Objective Type Questions

1. What is the primary purpose of coding standards?
2. Which of the following is NOT typically part of a coding standard?
3. What is code review?
4. Which tool is commonly used for code reviews?
5. What is one major benefit of code reviews?
6. What is typically checked during a code review?
7. Why are meaningful variable names recommended in coding standards?
8. What role does commenting play in coding standards?

9. Which phase comes immediately after code writing but before deployment in a standard software development cycle?
10. Which statement should be avoided in structured programming?

Answers to Objective Type Questions

1. To ensure consistency and readability
2. Hardware configuration
3. Examining code written by others for errors and improvements
4. GitHub Pull Requests
5. Helps catch bugs early
6. Easier maintenanceAll of the above
7. To improve code clarity
8. Explains the logic and improves understanding
9. Testing
10. GOTO

Assignments

1. Explain the importance of coding standards in a collaborative software development environment.
2. Describe the steps involved in the coding phase of software development. Illustrate your explanation with an example, such as building a basic calculator or a banking feature like “check account balance”.
3. Discuss the role of coding in software engineering.
4. Differentiate between coding standards and coding guidelines.
5. Imagine you are part of a development team working on a banking application.
6. List and explain any five characteristics of a good programming language that make it suitable for enterprise software development like banking systems.
7. What are the key goals of writing good code in software engineering? Briefly explain each goal with respect to a banking application (e.g., correctness, readability, reusability, efficiency, and scalability).

8. Discuss the importance of commenting and documentation in coding.
9. List and explain the different types of code reviews.
10. Describe the complete code review process, from submission to code integration.

Suggested Reading

1. Steve McConnell – *Code Complete: A Practical Handbook of Software Construction* (2nd Edition), Publisher: Microsoft Press, 2004
2. Frederick P. Brooks Jr. – *The Mythical Man-Month: Essays on Software Engineering*, Publisher: Addison-Wesley, 1995
3. Tom DeMarco & Timothy Lister – *Peopleware: Productive Projects and Teams*, Publisher: Addison-Wesley, 2013
4. Gene Kim, Jez Humble, Patrick Debois, and John Willis – *The DevOps Handbook*, Publisher: IT Revolution Press, 2016

Reference

1. Sommerville, Ian. *Software Engineering* (10th Edition), Pearson Education, 2015.
2. Pressman, Roger S., and Maxim, Bruce R., *Software Engineering: A Practitioner's Approach* (8th Edition), McGraw-Hill Education, 2014.
3. Rajib Mall, *Fundamentals of Software Engineering* (5th Edition), PHI Learning Pvt. Ltd., 2018.
4. Pankaj Jalote, *An Integrated Approach to Software Engineering* (3rd Edition), Springer, 2005.
5. Fenton, Norman E., and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous and Practical Approach* (2nd Edition), Brooks/Cole, 1998.



Software reliability and Quality Management

Learning Outcomes

The learners will be able to:

- ◆ Recall and list the key principles and guidelines of standard coding practices
- ◆ Explain the importance of adhering to coding standards in software development
- ◆ Apply appropriate coding standards to write clean, readable, and maintainable code
- ◆ Analyze a given piece of code to identify deviations from established coding standards

Prerequisites

Software Reliability and Quality Management are crucial aspects of software engineering that ensure the development of dependable and high-quality software products. Software reliability refers to the probability that software will function correctly without failure over a specified period under specified conditions. It focuses on minimizing errors, crashes, and system failures to enhance user satisfaction and trust.

On the other hand, Quality Management involves the systematic process of planning, controlling, and improving software quality throughout the development lifecycle. It encompasses various activities like quality assurance, quality control, and continuous improvement practices. Together, software reliability and quality management help in delivering robust, efficient, and maintainable software systems that meet customer requirements and industry standards.

Keywords

Software Reliability, Mean Time to Failure, Mean Time to Repair, Failure Rate, Quality Assurance.

Discussion

4.2.1 Introduction to Software Reliability

Software reliability refers to the probability that a software system will function without failure under specified conditions for a certain period of time. It is a key aspect of software quality that measures the software consistently, performs the correct operations, handles unexpected conditions gracefully, and avoids failures during its operation. In critical systems like online banking, high reliability is essential, as even a single malfunction can have serious consequences.

Consider a real time example of an online banking system such as SBI YONO, HDFC Bank app, or ICICI iMobile. Suppose a user logs in to the banking app to check their balance and transfer ₹10,000 to a friend. For the software to be considered reliable, it must correctly authenticate the user, accurately display the account balance, securely process the fund transfer, and provide confirmation through SMS or email. This entire process should work smoothly and correctly every time, regardless of network conditions or system load.

In contrast, an unreliable system may crash during a transaction, deduct money without crediting the recipient, display incorrect balance information, or allow unauthorized access due to weak security. Such failures can lead to user dissatisfaction, financial losses, and damage to the bank's reputation.

4.2.1.1 Functional And Non Functional Requirements

Functional requirements specify what the system should do the features, services, and operations the software must support.

Examples:

- ◆ User login and authentication
- ◆ Search functionality
- ◆ Report generation
- ◆ Data input validation
- ◆ Transaction processing

Non-functional requirements specify how the system performs its functions, focusing

on system qualities or constraints.

Examples:

- ◆ Performance: Response time must be under 2 seconds.
- ◆ Scalability: Must handle 10,000 users simultaneously.
- ◆ Security: All data must be encrypted in transit.
- ◆ Usability: The system should be user-friendly for non-technical users.
- ◆ Reliability: 99.9% uptime required.
- ◆ Maintainability: Code must be modular and easy to update.

System functional requirements may specify error checking, recovery features, and system failure protection. System reliability and availability are specified as part of the non-functional requirements for the system.

4.2.1.2 Failure and Fault in Terms of Software Reliability

In the context of software reliability, the concepts of failure and fault are closely connected but distinct. A fault (also known as a bug) is an internal defect in the software's code, design, or logic that has the potential to cause incorrect behavior. These faults typically arise from errors during development, such as logic mistakes, misinterpreted requirements, or missing conditions. While a fault may exist in the system, it does not always lead to a visible issue unless it is activated during execution. A failure, on the other hand, is the observable deviation from the software's expected behavior when a fault is triggered during runtime. Failures directly affect the system's reliability, as they represent moments when the software does not function correctly. In summary, a fault is the root cause, and a failure is the visible outcome, reducing the software's perceived reliability when it occurs.

4.2.1.3 Key Features of Software Reliability

The features of software reliability highlight the key qualities that make a software system dependable and trustworthy. These features include consistency, fault tolerance, accuracy, and the ability to recover from failures. Together, they ensure that the software performs reliably under varying conditions, meets user expectations, and maintains overall system stability throughout its operational life.

1. **Error-Free Operation:** Error-free operation is a crucial aspect of reliable software, ensuring that the system functions without bugs, glitches, or failures that could disrupt user tasks. This characteristic helps maintain smooth performance and enhances user trust by preventing unexpected interruptions during usage.
2. **Consistent Performance:** Consistent performance is a key feature of reliable software, ensuring that it behaves predictably and uniformly across various environments, platforms, and usage scenarios. This consistency builds user confidence and allows the software to perform effectively regardless of the

conditions in which it is used.

3. **Dependability Over Time:** Dependability over time is an essential feature of reliable software, ensuring that it continues to function properly even as it is used over extended periods. This includes maintaining stability and performance under stress, varied user behavior, or increasing loads, thereby ensuring that the software remains trustworthy and effective in the long run.
4. **Error Detection and Recovery:** Reliable software has robust mechanisms for detecting errors and recovering from failures. It should handle unexpected issues gracefully without crashing, allowing it to maintain operations or restore itself to a stable state.
5. **Minimal Downtime:** Reliable software ensures minimal downtime or outages, meaning users can access the system and perform their tasks with as little interruption as possible. This is particularly crucial for mission-critical systems, where availability is key.
6. **Scalability and Adaptability:** A reliable system is scalable, meaning it can handle growing user demands or increasing amounts of data without compromising its performance. It is also adaptable to new features or changes in the operating environment, maintaining reliability despite changes or updates.
7. **Security and Robustness:** Software reliability also entails being secure and robust, meaning it can handle potential security breaches or malicious attacks without significant damage or failure. Reliable software must have strong safeguards in place to protect sensitive data and ensure smooth operation.

4.2.1.4 Factors Affecting Software Reliability

Software reliability is influenced by several factors that can either enhance or diminish its performance. These factors are critical to understanding how reliable a software system can be and what may cause it to fail or perform inconsistently. Some key factors affecting software reliability include:

1. **Software Design and Architecture:** The overall design and structure of the software significantly impact its reliability. A well designed system with clear modular components and a robust architecture is easier to maintain and test, making it less likely to encounter reliability issues. Poor design or overly complex structures can lead to bugs, difficulties in maintenance, and increased chances of failure.
2. **Code Quality:** The quality of the code is fundamental to software reliability. Well written, clean, and efficient code is less likely to contain errors. Conversely, poorly written or complex code can introduce bugs that affect performance and stability. Additionally, the use of proper coding practices and adherence to coding standards contribute to creating more reliable systems.
3. **Testing and Validation:** Thorough testing and validation of the software

are essential to ensure reliability. The more rigorous the testing (including unit tests, integration tests, and stress tests), the more likely the software is to be free from critical errors. Inadequate testing or overlooking edge cases can result in unanticipated failures or performance issues during real-world usage.

4. **User Input and Interaction:** The way users interact with the software also impacts its reliability. Poorly designed user interfaces or failure to anticipate user errors can lead to system crashes or unexpected behavior. Ensuring that the software can handle a variety of inputs and user actions gracefully helps in preventing issues from arising.
5. **Environmental Factors:** The environment in which the software operates, such as hardware, operating systems, or network configurations, can affect its reliability. Variations in hardware, differences in system configurations, and environmental conditions (like network latency or resource availability) can lead to issues such as performance degradation or crashes.
6. **Complexity of the System:** Software systems that are highly complex tend to be less reliable due to the greater number of interactions, dependencies, and potential failure points. The complexity of managing different components, services, or modules increases the chances of bugs and operational failures, which can negatively impact reliability.
7. **Maintenance and Updates:** Continuous maintenance and timely updates are essential for maintaining software reliability. As software evolves, new bugs can be introduced, or previous issues might reappear. Regular updates to fix known issues, optimize performance, and add new features help keep the software reliable over time.
8. **Hardware Reliability:** The reliability of the hardware running the software also plays a crucial role. If the hardware is prone to failure, such as frequent disk crashes or memory issues, it can negatively affect the performance and stability of the software, regardless of its own quality.
9. **External Dependencies:** Software often depends on external libraries, third-party services, or integrations. The reliability of these external components can directly affect the overall reliability of the system. If these dependencies are unreliable, they can cause failures in the software, even if the core software itself is designed and coded perfectly.
10. **Human Factors:** Human errors in the development, deployment, or maintenance phases can lead to reliability issues. Mistakes made during coding, testing, or system configuration can introduce bugs that affect the software's performance. Additionally, poor communication among team members or lack of adherence to best practices can lead to reliability problems.
11. **Concurrency and Synchronization Issues:** In multi threaded or distributed systems, issues related to concurrency and synchronization can significantly affect reliability. Incorrect synchronization between different threads or

components can lead to race conditions, deadlocks, and other errors that make the software unreliable.

4.2.1.5 How to Improve Software Reliability

- ◆ Adopt coding standards and best practices.
- ◆ Conduct rigorous and continuous testing.
- ◆ Perform regular code reviews.
- ◆ Use fault-tolerant design techniques.
- ◆ Monitor software behavior after deployment and fix issues promptly.

4.2.1.6 Reliability Metrics

Reliability metrics in software engineering are quantitative measures used to assess the reliability of a software system that is, how consistently and accurately the software performs its intended functions without failure over time.

These metrics help in:

- ◆ Evaluating software quality
- ◆ Identifying areas for improvement
- ◆ Predicting software behavior in the real world
- ◆ Making informed maintenance and design decisions

Different reliability metrics are

1. MTTF (Mean Time to Failure)

MTTF is the average time a system or software operates before experiencing a failure. It is commonly used for non-repairable systems, such as embedded systems, and a higher MTTF indicates greater reliability.

If a software component runs for 500 hours before failing and another for 700 hours, the MTTF is $(500 + 700) / 2 = 600$ hours.

2. MTTR (Mean Time to Repair)

MTTR is the average time taken to diagnose, fix, and restore a software system after a failure occurs. It indicates how quickly the system can recover, and a shorter MTTR reflects better maintainability and higher system availability.

A software application crashes and takes 2 hours to repair the first time, and 1 hour the next, $MTTR = (2 + 1) / 2 = 1.5$ hours.

3. MTBF (Mean Time Between Failures)

MTBF is the average time between successive failures, including both the normal operating time and the repair time. It is commonly used for repairable systems, and a

higher MTBF indicates that the system is less likely to fail frequently

Formula for calculating MTBF is:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Example:

If MTTF = 600 hours and MTTR = 2 hours, then $\text{MTBF} = 600 + 2 = 602$ hours.

4. Failure Rate

Failure rate is the number of failures per unit of time (e.g., per hour, per day). It helps quantify how frequently a system is expected to fail, and a lower failure rate indicates a more reliable system.

Formula for calculating Failure rate:

$$\text{Failure Rate} = 1 / \text{MTTF} \text{ (for non-repairable systems)}$$

If a system has an MTTF of 1000 hours, then the failure rate = $1/1000 = 0.001$ failures/hour.

4.2.2 Quality Management

Quality Management in software engineering refers to the systematic process of ensuring that a software product meets defined standards, customer expectations, and business goals. It focuses on delivering a high-quality product by planning, controlling, and improving all activities involved in software development and maintenance. It is not just about finding defects but about building quality into the process itself.

An example of software quality can be seen in an online banking application launched by a bank. In this context, correctness is demonstrated by the application's ability to accurately perform transactions such as balance inquiries, fund transfers, and bill payments. Reliability is reflected in its consistent performance without crashes, even during peak usage times. The app exhibits usability by allowing customers to navigate easily, locate features quickly, and complete transactions with minimal assistance. Efficiency is evident in the fast loading of pages and swift processing of transactions. In terms of maintainability, the application's design enables developers to fix bugs or introduce updates without overhauling the entire system. Lastly, portability is achieved by ensuring the application runs smoothly across multiple platforms, including Android, iOS, and web browsers. This comprehensive adherence to software quality attributes makes the banking application robust, user-friendly, and dependable.

A Software Quality Management System (SQMS) include

- ◆ **Managerial Structure:** The quality system plays a central role in managing the overall structure of the organization. Every company establishes a defined managerial hierarchy to oversee and maintain quality processes.

- ◆ **Individual Responsibilities:** Each team member in the organization is assigned specific responsibilities related to quality. These roles should be clearly defined and regularly reviewed by senior management to ensure accountability and commitment to quality goals.
- ◆ **Quality System Activities:** Core activities within a quality management system typically include project audits, systematic reviews, and the development of quality guidelines and methodologies. These practices ensure that quality is embedded throughout the software development lifecycle.

4.2.2.1 Components of Quality Management:

1. **Quality Assurance (QA):** QA is a preventive process. It ensures that proper processes, methods, and standards are followed throughout the software development lifecycle. Its goal is to prevent defects by improving development and testing procedures.
2. **Quality Control (QC):** QC is a corrective process. It involves identifying defects in the actual software product through activities like testing, inspections, and reviews. QC ensures the final product meets the desired quality standards.
3. **Quality Planning:** Quality planning involves setting quality objectives, defining the quality standards, and deciding how these objectives will be achieved. It helps in selecting the right processes and resources.
4. **Continuous Improvement:** This involves ongoing efforts to improve processes, products, and services based on feedback, reviews, and lessons learned from past projects. Consider a healthcare software system used by hospitals to manage patient records and treatments.

If quality management practices are applied:

- ◆ QA ensures that the system is built using standard protocols for data security (e.g., encryption of patient data).
- ◆ QC involves testing the system to ensure there are no bugs in features like patient registration or prescription management.
- ◆ Quality planning sets goals like 99.9% uptime and zero data breaches.
- ◆ Continuous improvement ensures updates based on doctor/nurse feedback, enhancing usability.

Without proper quality management, the software may have bugs, security flaws, or usability issues, leading to errors in patient care.

Quality management is essential in software development to deliver reliable, efficient, and customer-friendly products. It integrates quality into every stage of development, ensuring long-term success and sustainability.

4.2.2.2 Objectives of Quality Management

Different objectives of quality management are :

1. Deliver High-Quality, Defect-Free Software

The primary goal is to ensure that the final software product is free of errors and defects, both in functionality and performance. This leads to a stable, reliable, and accurate system that meets the defined requirements.

2. Ensure Customer Satisfaction

Quality management aims to meet or exceed customer expectations by delivering software that is usable, efficient, and meets business goals. Satisfied customers are more likely to return, recommend, and trust the product or service.

3. Minimize Rework and Reduce Development Costs

By identifying and preventing issues early in the development cycle, quality management helps reduce the cost and time required for bug fixing or redesigning. This leads to efficient use of resources and faster delivery.

4. Improve Process Efficiency

Implementing standardized and optimized development practices ensures that the software development process is streamlined. This increases team productivity, reduces delays, and enhances overall software quality.

5. Build Organizational Trust and Reputation

Consistently delivering high-quality software builds the organization's credibility and reputation in the market. This fosters long-term client relationships and opens doors to more business opportunities.

4.2.2.3 Capability Maturity Model Integration (CMMI) Software Quality Management

CMMI integrates various capabilities into a coherent framework that organizations can use to evaluate and improve their software development processes. It focuses on improving processes and aligning them with business goals to deliver high-quality products efficiently. CMMI is especially valuable in software quality management for its focus on continuous process improvement, defect prevention, and quality assurance.

Key Concepts of CMMI in Software Quality Management:

1. Maturity Levels

CMMI defines five maturity levels that indicate the progression of process improvement within an organization:

- ◆ Level 1 – Initial: Processes are often chaotic, and success depends on individual efforts.

- ◆ Level 2 – Managed: Processes are project-specific and managed, with some consistency.
- ◆ Level 3 – Defined: Processes are well-defined and standardized across the organization.
- ◆ Level 4 – Quantitatively Managed: Performance is measured and controlled using quantitative data.
- ◆ Level 5 – Optimizing: Continuous process improvement is pursued through innovation and refinement.

Characteristics of the Maturity levels

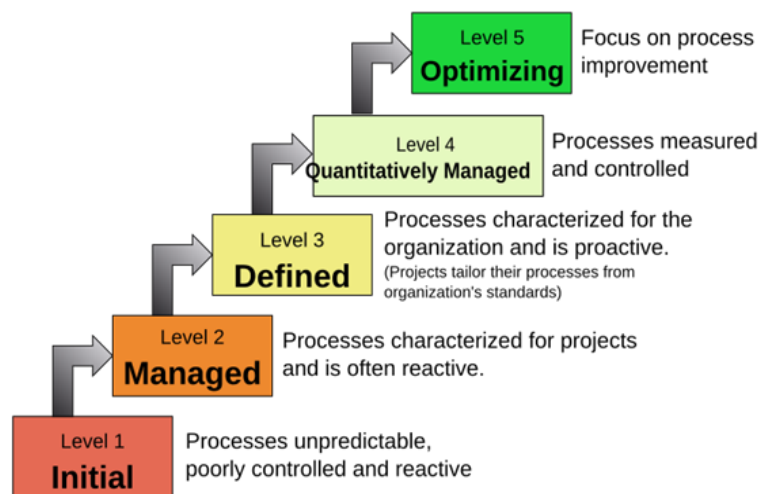


Fig 4.2.1 Characteristics of the maturity levels

2. Process Areas

Each maturity level is supported by process areas that provide detailed practices for process improvement. For software quality management, CMMI emphasizes areas like:

- ◆ Configuration Management: Ensures that software components and changes are controlled and tracked to prevent defects.
- ◆ Project Monitoring and Control: Ensures projects are monitored and risks are managed to avoid quality issues.
- ◆ Requirements Management: Helps define and manage customer requirements, reducing misinterpretations and defects.
- ◆ Measurement and Analysis: Uses data to improve decisions and processes, ensuring the system meets quality standards.

3. Continuous Improvement

CMMI promotes continuous improvement by refining processes over time, thereby preventing defects early in the software development cycle and improving the overall quality. With practices like root cause analysis, teams can pinpoint weaknesses and

correct them, leading to more reliable and efficient systems.

4. Process Focus

CMMI emphasizes process management rather than just outcomes, encouraging organizations to define, measure, and control their processes. This focus ensures that quality is maintained through well-established and repeatable processes, not just individual successes.

5. Defect Prevention

In software quality management, CMMI encourages organizations to implement defect prevention mechanisms by identifying potential problems early in the process. Practices like design reviews, code inspections, and test-driven development help to identify and mitigate defects before they affect the final product.

Impact of CMMI on Software Quality:

By aligning software development practices with CMMI, organizations can achieve:

- ◆ **Improved Product Quality:** Consistent processes lead to predictable outcomes and higher-quality software products.
- ◆ **Reduced Costs:** Early detection and prevention of defects minimize costly rework and late-stage fixes.
- ◆ **Higher Customer Satisfaction:** A well-defined process ensures that products meet customer expectations reliably and on time.
- ◆ **Predictable Outcomes:** With quantifiable and measured processes, the likelihood of software failures is reduced, improving the system's reliability.

4.2.2.4 Common Quality Standards

Common quality standards in software engineering provide frameworks and guidelines to ensure that software products meet high levels of quality and consistency. ISO 9001 is an internationally recognized standard for quality management systems, focusing on meeting customer requirements and improving organizational efficiency through well-defined processes. CMMI (Capability Maturity Model Integration) is a process improvement framework that helps organizations assess and enhance the maturity of their software development processes, moving from ad-hoc practices to optimized and well-managed procedures.

IEEE Standards offer specific guidelines and best practices for various aspects of software engineering, including documentation, testing, and the overall software development lifecycle. Together, these standards help ensure that software is developed systematically, is reliable, and aligns with industry best practices.

4.2.2.5 Tools and Techniques used in Software quality management System

1. **Automated Testing Tools :** Automated testing tools help execute test cases automatically to verify that the software behaves as expected. Tools like Selenium (for web applications) and JUnit (for Java applications) increase testing efficiency and accuracy while reducing manual effort, especially in regression testing.
2. **Static Code Analysis Tools :** Static code analysis tools examine the source code without executing it to detect potential bugs, code smells, security vulnerabilities, and maintainability issues. SonarQube is widely used to enforce coding standards and improve overall code quality early in the development process.
3. **Version Control Systems :** Version control systems like Git help developers manage changes to source code over time. They support collaboration by allowing multiple developers to work on the same codebase, track revisions, and revert to previous versions if needed, thereby preventing code conflicts and loss.
4. **Defect Tracking Tools (e.g., Bugzilla, JIRA) :** Defect tracking tools allow teams to log, manage, and monitor software bugs or issues throughout the development lifecycle. Tools like Bugzilla and JIRA help ensure that defects are addressed efficiently, tracked to resolution, and documented for future reference.
5. **Code Reviews and Pair Programming :** Code reviews involve systematically examining another developer's code to catch mistakes, improve quality, and encourage knowledge sharing. Pair programming, where two developers work together on the same code, enhances collaboration and helps in identifying defects early during development.

Recap

- ◆ **Software reliability :** Software reliability refers to the probability that a software system will function without failure under specified conditions for a certain period of time.
- ◆ **Functional and non-functional requirements :** Functional requirements specify what the system should do, the features, services, and operations the software must support. Non-functional requirements specify how the system performs its functions, focusing on system qualities or constraints.
- ◆ **Failure and Fault in Terms of Software Reliability :** A fault (also known as a bug) is an internal defect in the software's code, design, or logic that has the potential to cause incorrect behavior. A failure, on the other hand, is the observable deviation from the software's expected

behavior when a fault is triggered during runtime.

- ◆ Key Features of Software Reliability are Error-Free Operation, Consistent Performance, Dependability Over Time, Error Detection and Recovery, Minimal Downtime, Scalability and Adaptability, Security and Robustness.
- ◆ Reliability Metrics : Reliability metrics in software engineering are quantitative measures used to assess the reliability of a software system, that is, how consistently and accurately the software performs its intended functions without failure over time.
- ◆ Quality Management in software engineering refers to the systematic process of ensuring that a software product meets defined standards, customer expectations, and business goals.
- ◆ Components of Quality Management are Quality Assurance (QA), Quality Control (QC), Quality Planning, Continuous Improvement.
- ◆ Objectives of Quality Management are : Deliver High-Quality, Ensure Customer Satisfaction, Minimize Rework and Reduce Development Costs, Improve Process Efficiency, Build Organizational Trust and Reputation.
- ◆ Capability Maturity Model Integration (CMMI) Software Quality Management: CMMI integrates various capabilities into a coherent framework that organizations can use to evaluate and improve their software development processes.
- ◆ Common quality standards in software engineering provide frameworks and guidelines to ensure that software products meet high levels of quality and consistency. It includes ISO 9001, CMMI, IEEE Standards.

Objective Type Questions

1. What is the primary objective of quality management in software development?
2. Which of the following is NOT an objective of quality management?
3. What does quality assurance aim to prevent?
4. Which of the following objectives is related to making future maintenance easier?
5. How does quality management contribute to cost efficiency?
6. Ensuring compliance with industry standards (like ISO or CMMI) is an objective of:

7. Which of these is a key objective of quality control
8. What CMMI level emphasizes continuous improvement?
9. Which tool analyzes code without execution?
10. Which standard focuses on quality management systems globally?

Answers to Objective Type Questions

1. To ensure the software product meets customer requirements and expectations with minimal defects.
2. Increasing software defects
3. Quality assurance aims to prevent defects by improving the development and testing processes.
4. Maintainability
5. By identifying and fixing defects early, reducing rework and post-release fixes, thus saving time and cost.
6. Quality management
7. Software
8. Optimizing
9. SonarQube
10. ISO

Assignments

1. Define Quality Management. Explain its importance in software development.
2. List and explain any four key objectives of Quality Management. Provide real-world examples.
3. Differentiate between Quality Assurance (QA) and Quality Control (QC) with respect to quality management objectives.
4. Discuss how customer satisfaction is achieved through effective quality management practices.

5. Explain the role of continuous improvement in Quality Management. Why is it essential for long-term product success?
6. Identify how adherence to industry standards (e.g., ISO, CMMI) supports the objectives of quality management.
7. Describe the relationship between cost efficiency and quality management. How does early defect detection reduce overall project cost?
8. Write a short essay on how quality management ensures software maintainability and scalability in large projects.

Suggested Reading

1. Steve McConnell – *Code Complete: A Practical Handbook of Software Construction* (2nd Edition), Publisher: Microsoft Press, 2004
2. Frederick P. Brooks Jr. – *The Mythical Man-Month: Essays on Software Engineering*, Publisher: Addison-Wesley, 1995
3. Tom DeMarco & Timothy Lister – *Peopleware: Productive Projects and Teams*, Publisher: Addison-Wesley, 2013
4. Gene Kim, Jez Humble, Patrick Debois, and John Willis – *The DevOps Handbook*, Publisher: IT Revolution Press, 2016

Reference

1. Sommerville, Ian. *Software Engineering* (10th Edition), Pearson Education, 2015.
2. Pressman, Roger S., and Maxim, Bruce R., *Software Engineering: A Practitioner's Approach* (8th Edition), McGraw-Hill Education, 2014.
3. Rajib Mall, *Fundamentals of Software Engineering* (5th Edition), PHI Learning Pvt. Ltd., 2018.
4. Pankaj Jalote, *An Integrated Approach to Software Engineering* (3rd Edition), Springer, 2005.
5. Fenton, Norman E., and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous and Practical Approach* (2nd Edition), Brooks/Cole, 1998.



TESTING

Learning Outcomes

The learners will be able to:

- ◆ understand the Purpose of Software Testing
- ◆ identify Various Types of Testing
- ◆ analyze Test Results
- ◆ evaluate the Role of Automation Tools in Testing.

Prerequisites

Imagine you run a popular cake shop. Every day, customers come in expecting delicious, fresh cakes. You have a team of bakers who follow recipes (just like software developers follow design plans). Now, before you sell any cake, you always taste a small piece to ensure it's perfect soft, sweet, and fresh. You check if the ingredients were used correctly, the oven worked properly, and there are no mistakes like too much sugar or undercooked batter.

This act of tasting and checking is similar to software testing.

In software development:

- The developers are like bakers who write code (like baking cakes).
- The testers are like the cake taster, they check if the software is working fine, has no bugs, and meets customer expectations.

If you skip tasting and give the cake to the customer without checking, there's a risk it may not taste good. Similarly, if software is released without testing, users might face bugs, crashes, or security issues, leading to unhappy customers.

Keywords

Verification, Validation, Black Box Testing, White Box Testing, Grey Box Testing

Discussion

4.3.1 Testing in Software Engineering

Software testing is an important process in the Software Development Lifecycle(SDLC). It involves verifying and validating that a Software Application is free of bugs, meets the technical requirements set by its Design and Development, and satisfies user requirements efficiently and effectively. It ensures the reliability, functionality, and quality of the software product. Two main terms used in software testing are verification and validation.

4.3.1.1 Verification and Validation in Software Testing

Verification and Validation are two fundamental aspects of software testing that ensure the quality and correctness of a software product. Both verification and validation are essential to deliver high-quality software. Verification prevents defects, while validation ensures that the software is usable, correct, and fulfills its intended purpose.

Though they are often mentioned together, they serve distinct purposes:

a. Verification

Verification is the process of evaluating the intermediate work products during the software development lifecycle to ensure they meet the specified requirements. It focuses on being process-oriented and answers the critical question:

"Are we building the product right?"

This involves systematic activities such as reviews, inspections, walkthroughs, and desk checks, and is typically performed during the early stages of development. The primary goal is to confirm that the product is being developed in accordance with the design specifications and requirements. For instance, verification may include checking whether design documents conform to established standards or reviewing code for logical errors and adherence to coding style guidelines.

b. Validation

Validation is the process of evaluating the final software product to ensure that it meets the business needs and user requirements. It is product-oriented and answers the key question:

"Are we building the right product?"

Validation involves activities such as unit testing, integration testing, system testing, and acceptance testing, and is typically conducted after development is complete or in

the later stages of the software development lifecycle. The primary goal is to confirm that the software behaves as expected and provides a satisfactory user experience. For example, it includes executing test cases to validate system functionality and performing user acceptance testing to verify usability and business alignment.

Table 4.3.1 Difference between Verification and validation

Aspect	Verification	Validation
Purpose	Are we building the product right?	Are we building the right product?
Focus	Process	Product
Activities	Reviews, inspections, walkthroughs	Testing (functional, non-functional)
Performed by	Developers, QA teams	QA teams, users, stakeholders
Timing	During development	After development

4.3.1.2 Purpose of Testing

We know that everything must be tested before delivering. In case of software development, purpose of testing are:

- ◆ **To verify that the software meets the specified requirements :** Software is developed based on a set of requirements provided by stakeholders or clients. Testing checks whether each function, feature, and behavior works exactly as described in the requirements document. This ensures that the software is doing what it's supposed to do.
- ◆ **To detect and fix defects early in the development process :** Bugs or errors in code can cause serious problems if left unchecked. Testing helps to identify these defects at an early stage, when it's easier and cheaper to fix them. Early detection helps improve the overall quality and reduces the risk of failure later.
- ◆ **To ensure the software performs well under different conditions :** Testing checks how the software behaves under various scenarios—such as different devices, operating systems, network speeds, or user inputs. It helps ensure the application is stable, efficient, and responsive, even under stress or unusual conditions.
- ◆ **To validate that the software is user friendly, secure, and reliable :** Beyond functionality, testing evaluates usability (is it easy to use?), security

(can it protect user data?), and reliability (does it work consistently?). This ensures that users have a safe and satisfactory experience with the software.

4.3.2 Software Testing Life Cycle (STLC)

Software Testing Life Cycle (STLC) is a systematic process that defines a series of phases or stages followed while testing software to ensure its quality and functionality. It outlines how testing should be carried out in a structured and organized way, starting from understanding requirements to the final test closure.

While specific steps can vary, a typical STLC includes:

1. **Requirement Analysis:** Understanding the software requirements from a testing perspective to identify testable aspects.
2. **Test Planning:** Defining the scope, objectives, resources, and schedule for testing activities. Creating a test plan document.
3. **Test Case Design:** Developing test cases based on the requirements, specifying inputs, expected results, and execution conditions.
4. **Test Environment Setup:** Configuring the necessary hardware, software, and network environment to execute the test cases.
5. **Test Execution:** Running the designed test cases on the software build and recording the results (pass/fail).
6. **Defect Reporting:** Identifying and documenting any discrepancies between the expected and actual results (bugs or defects).
7. **Defect Retesting:** Once developers fix the reported defects, testers re-execute the test cases to verify the fixes.
8. **Regression Testing:** Testing the previously working parts of the software to ensure that the fixes haven't introduced new issues.
9. **Test Closure:** Summarizing the testing activities, results, and lessons learned once testing is complete.

4.3.3 Types of Software Testing

4.3.3.1 Based on Software Approach/ Method

Based on approach or method types of Software Testing mainly categorized into the two domains, Manual Testing and Automation Testing.

a. Manual Testing

Manual testing is a technique to test the software that is carried out using the functions and features of an application. In manual software testing, a tester tests the software by following predefined test cases. In this testing, testers make test cases for the codes, test the software, and give the final report about that software. Manual testing is time-consuming because it is done by humans, and there is a chance of human errors.



Every new application must be manually tested before its testing can be automated. Manual Testing requires more effort than automation testing but is necessary to check automation feasibility. There is no requirement for knowledge of any testing tool in manual testing. Unlike automated testing, manual testing relies on predefined test cases and can be time-consuming and prone to human error.

Steps in Manual Testing

Below is the representational diagram of the steps in the manual testing process in detailed manner:

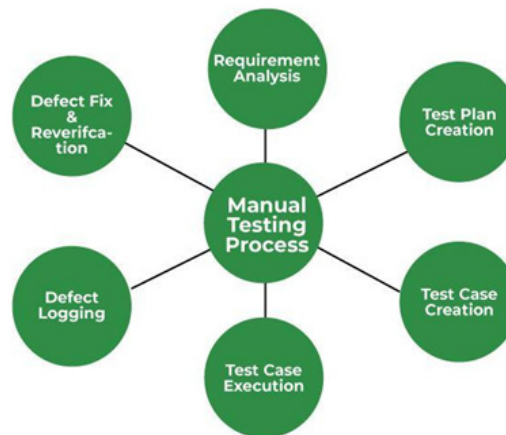


Fig 4.3.1 Steps in Manual testing

1. Requirement Analysis: Study the software project documentation, guides, and Application Under Test (AUT). Analyze the requirements from SRS.
2. Test Plan Creation: Create a test plan covering all the requirements.
3. Test Case Creation: Design the test cases that cover all the requirements described in the documentation.
4. Test Case Execution: Review and baseline the test cases with the team lead and client. Execute the test cases on the application under test.
5. Defect Logging: Detect the bugs, log and report them to the developers.
6. Defect Fix and Re-verification: When bugs are fixed, again execute the failing test cases to verify they pass.

Advantages of Manual Testing

1. Flexibility and Adaptability : Testers can easily adjust to changes in requirements and test cases.
2. Better for UI/UX Testing : Human testers can judge visual elements, layout, and user experience more effectively than automation tools.
3. Cost-Effective for Short-Term Projects : No need to invest in automation tools or frameworks for simple or one-time tests.

4. **Immediate Human Insight** : Testers can quickly catch inconsistencies or unexpected behavior that automated scripts might miss.
5. **Ideal for Exploratory and Ad-hoc Testing** : Supports spontaneous and intuitive testing without predefined test cases.

Disadvantages of Manual Testing

1. **Time-Consuming and Slower Execution** : Especially for large projects or regression testing, manual execution can be slow.
2. **Prone to Human Error** : Repetitive tasks may lead to oversight or inconsistent results.
3. **Less Efficient for Repetitive Tests** : Automation handles repetitive and data-driven tests more effectively.
4. **Not Scalable** : Difficult to manually test large-scale applications with frequent updates.
5. **Lack of Reusability**: Unlike automated scripts, manual test cases need to be executed fresh every time.

b. Automation Testing

Automated Testing means using special software for tasks that people usually do when checking and testing a software product. Nowadays, many software projects use automation testing from start to end, especially in agile and DevOps methods. This means the engineering team runs tests automatically with the help of software tools . It will help to keep the testing team to make the process faster. Continuous delivery (CD) and quickly sends the new code to users.

Automated testing is important for this because it converts the manual steps into automation. Continuous integration (CI) checks the new code changes to prevent issues. CD gets after CI done everything well. Automated testing, CI & CD will together prove that the new code is error-free and ready for deployment quickly for the project purpose.

Types of Automation Testing

1. **Unit Testing** : Tests the smallest part of the code, called a unit, to make sure it works correctly. It is done during development.
2. **Integration Testing**: Checks how different parts of the software work together as a group. It ensures components interact properly and meet functional requirements.
3. **Smoke Testing** : A quick check to see if the basic functions of the software are working. It helps decide if the build is stable enough for further testing.
4. **Performance Testing**: Measures how well the software performs under different loads. It checks for speed, stability, and responsiveness.
5. **Regression Testing** : Makes sure new changes in the code haven't broken anything that was working before.



6. **Security Testing:** Identifies security weaknesses in the application. It helps find risks or vulnerabilities and ensures the app is safe from threats.
7. **Acceptance Testing:** Done at the final stage to confirm the software meets user requirements and is ready for delivery.
8. **API Testing:** Tests the Application Programming Interface (API) to check if it works correctly, is secure, and returns the right results.
9. **UI Testing:** Verifies that all elements on the user interface (like buttons, forms, links, etc) work as expected and look correct.

Advantages of Automation Testing

- ◆ **Easier Test Execution:** Once set up, automated tests can run on their own, and you can check the results later. This saves time and effort.
- ◆ **More Reliable Testing:** Automation makes sure all parts of the application are tested equally, leading to better software quality.
- ◆ **Better Test Coverage:** You can run more tests in less time, which helps find more bugs and test more features, even in complex applications.
- ◆ **Less Human Error:** Since the process is automated, there's little chance of mistakes caused by human oversight.
- ◆ **Saves Time and Money Over Time:** Although it may cost more at the start, automation testing saves time and money in the long run by reducing manual work.
- ◆ **Early Bug Detection:** Automated tests can find problems early, making it easier and cheaper to fix them.

Disadvantages of Automation Testing

- ◆ **High Setup Cost:** Setting up automation takes time, money, and effort, especially when choosing tools and writing custom scripts.
- ◆ **Not Everything Can Be Automated:** Some test cases need human judgment, especially those that involve design, usability, or user experience.
- ◆ **Limited to Certain Test Types:** It's hard to automate things like checking how user-friendly the interface is or testing sound and graphics.
- ◆ **Needs Programming Skills:** Writing automated test scripts usually requires knowledge of programming.
- ◆ **Risk of Incorrect Test Results:** Sometimes, automated tests may report bugs that aren't really there (false positives) or miss real bugs (false negatives).

4.3.3.2 Based on the goal of testing

What aspect of the software are we testing? Based on this question we can divide testing into Functional and Non functional testing.

a. Functional Testing

Functional testing is focused on verifying that the software behaves according to the defined requirements and specifications in other words, checking what the software does. This testing ensures that all features and functionalities work as intended when given the correct input and expected output. Functional testing is primarily black box testing, where the tester does not require knowledge of the internal code.

Techniques Used in Functional Testing: Testers employ various techniques to design effective functional test cases, including:

- ◆ **Boundary Value Analysis (BVA):** Testing at the edges of valid and invalid input ranges.
- ◆ **Equivalence Partitioning:** Dividing the input data into groups (partitions) where the system should behave similarly. Testing one value from each partition is usually sufficient.
- ◆ **Decision Table Testing:** Creating tables to represent complex business rules and conditions, ensuring all combinations are tested.
- ◆ **State Transition Testing:** Testing the system's behavior as it moves between different states.
- ◆ **Use Case Testing:** Designing tests based on how users will interact with the system to achieve specific goals.

Common types of functional testing include:

- ◆ **Unit Testing:** Testing individual components or functions of the code in isolation.
- ◆ **Integration Testing:** Ensuring that different modules or services interact correctly. Different types of integration testing are:
 - ◆ **Component Integration Testing:** Focuses on the interaction between individual components.
 - ◆ **System Integration Testing:** Tests the integration of different subsystems.
- ◆ **Regression testing:** Retesting previously tested parts of the software after changes (bug fixes, new features) to ensure no new issues have been introduced.
- ◆ **System Testing:** Testing the complete system as a whole to verify that it meets the requirements.
- ◆ **Acceptance Testing:** Verifying the system from the end-user's perspective to determine if it's ready for deployment.
- ◆ **User Acceptance Testing (UAT):** User Acceptance Testing (UAT) is the final phase of the software testing process where the end-users or clients verify that the software meets their business requirements and is ready for deployment. The main goal of UAT is to ensure the system performs

correctly in real-world scenarios and is acceptable for use.

- ◆ **Business Acceptance Testing (BAT):** Business Acceptance Testing (BAT) is conducted to assess whether the software aligns with the organization's business goals and supports its core processes. This type of testing is typically carried out by business analysts or domain experts who evaluate the system from a strategic and operational perspective to ensure it delivers the intended business value.
- ◆ **Alpha Testing:** Alpha Testing is conducted in the development environment by internal staff, such as developers or quality assurance teams. It aims to identify bugs and gather early feedback on the software's functionality and usability before it is released to external users for further testing.
- ◆ **Beta Testing:** Beta Testing is carried out by a limited group of external users in a real-world environment. It helps uncover unexpected issues that may not have been identified during earlier testing phases, providing valuable feedback before the software's final release to a broader audience.
- ◆ **Smoke Testing:** A quick initial test to check the basic critical functionalities of the software after a new build.
- ◆ **Sanity Testing:** A focused testing effort after a bug fix or minor change to ensure the specific issue is resolved and no new major problems have been introduced in that area.
- ◆ **Exploratory Testing:** An informal testing approach where testers simultaneously learn, design, and execute tests.
- ◆ **Ad-hoc Testing:** Unstructured and random testing performed without formal test cases.
- ◆ **Mutation Testing:** Introducing small changes (mutations) in the code to check if the test suite can detect these faults.

b. Non Functional Testing

Non functional testing focuses on how well the software performs beyond its functional behavior. It assesses aspects like performance, usability, reliability, and security, which are critical to delivering a high quality user experience. This type of testing helps ensure the software is robust, scalable, and user-friendly.

Key types of non functional testing include:

a. Performance Testing

Evaluates the speed, responsiveness, and stability of the application under varying loads. It ensures the system can handle user demands efficiently.

- ◆ **Load Testing:** Measures system performance under expected user load.
- ◆ **Stress Testing:** Checks how the system behaves under extreme or peak loads.
- ◆ **Endurance Testing (Soak Testing):** Evaluates stability over an extended period of usage.

- ◆ Spike Testing: Tests the system's reaction to sudden and extreme changes in load.

b. Security Testing

Focuses on identifying vulnerabilities and ensuring the confidentiality, integrity, and availability of data and systems. It includes:

- ◆ Penetration Testing: Simulating attacks to find security loopholes.
- ◆ Authentication & Authorization Checks: Ensuring only valid users can access the system.

c. Usability Testing

Assesses how intuitive, user-friendly, and easy to navigate the software is for end-users. It involves observing real users as they interact with the application to detect usability issues.

d. Compatibility Testing

Ensures that the software works consistently across different browsers, operating systems, devices, and network environments. This is crucial for applications that are accessed on multiple platforms.

e. Reliability Testing

Checks the software's ability to function correctly over time without failure. It helps ensure that the system maintains performance and data integrity under normal and abnormal conditions.

f. Accessibility Testing

Verifies that the application can be used by people with disabilities, such as those who rely on screen readers, keyboard navigation, or high-contrast themes. This ensures compliance with accessibility standards like WCAG (Web Content Accessibility Guidelines).

4.3.3.3 Based on Scope/Levels of Software Testing

Software testing can be classified in several ways depending on its purpose, scope, and execution strategy. One of the most widely used classifications is based on the level of testing within the software development lifecycle. These levels help ensure that the software is built correctly at each stage, from individual components to the complete system.

a. Unit Testing

It focuses on testing individual units or components of the software in isolation to ensure they function as intended. It is usually performed by developers and is the first level of testing. Unit tests are often automated and use tools such as JUnit for Java, NUnit for .NET, and PyTest for Python. This type of testing helps catch errors early in the development process, making it easier and cheaper to fix issues.

b. Integration Testing

It examines the interaction between integrated modules or components. Its main objective is to identify interface defects and verify the correct flow of data between different units. Integration testing can follow various approaches, including top-down, bottom-up, and big bang integration. It is generally performed by developers or testers and helps ensure that combined components work together seamlessly.

c. System Testing

It involves testing the entire integrated software system as a whole to validate that it meets the specified functional and non-functional requirements. It is performed by independent testing teams and covers a wide range of scenarios, including end-to-end business workflows, performance, security, and usability. System testing acts as a final check before the software moves to the user acceptance phase.

d. Acceptance Testing

It is the final level of testing, conducted to determine whether the software meets the business requirements and is ready for deployment. It is usually carried out by end users, clients, or product owners. This level includes User Acceptance Testing (UAT), where end users validate functionality from their perspective, and Business Acceptance Testing (BAT), where business stakeholders ensure the system aligns with organizational goals and processes. Acceptance testing ensures that the software is not only technically sound but also valuable and usable in the real-world context.

4.3.3.4 Based on Testing Techniques

Based on techniques used in software testing it is classified as

a. White Box Testing

White Box Testing is a technique where the internal structure of a system like design and code is manually checked. In this method, the development team goes through the code line by line to ensure everything works as intended. If they find any errors or mismatches in the code or design, they fix them. Since this process is done manually by humans, it can be very effective in spotting issues.

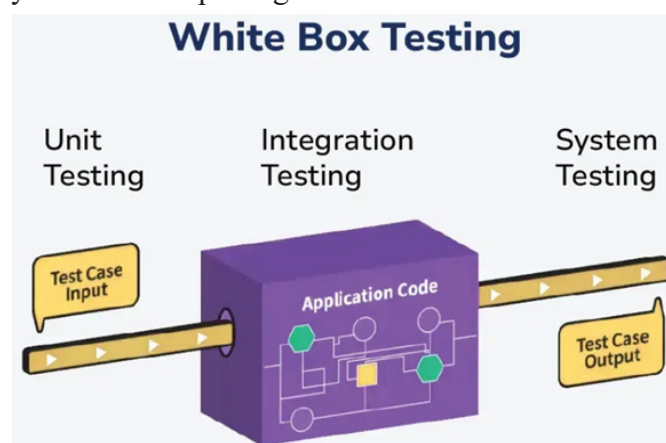


Fig 4.3.2 White Box Testing

Example of white box testing : Suppose we have a simple function in Python that returns the largest of two numbers:

```
def find_max(a, b):
    if a > b:
        return a
    else:
        return b
```

In White Box Testing, we would:

- ◆ Check the logic of the if-else condition.
- ◆ Test different paths:
 - When $a > b \rightarrow$ e.g., `find_max(10, 5)`
 - When $a \leq b \rightarrow$ e.g., `find_max(3, 7)` or `find_max(4, 4)`
- ◆ Check edge cases, like equal numbers.
- ◆ Make sure all lines of code are executed through different test cases (this is called code coverage).

Here are some common types of tests used in white box testing:

- ◆ **Path Testing:** This checks all possible execution paths in the program to make sure each function works as expected. It helps ensure logical conditions are correct and the code avoids unnecessary steps.
- ◆ **Input and Output Validation:** Different inputs are given to functions to check if the outputs are always correct. This helps verify that the software performs well under different conditions.
- ◆ **Security Testing:** This focuses on finding security problems in the code. Tools like static code analyzers are used to identify potential security flaws and ensure secure coding practices are followed.
- ◆ **Loop Testing:** This checks if loops (like for or while) work correctly and efficiently. It ensures that loops handle variables properly and don't cause errors like infinite loops or logic bugs.
- ◆ **Data Flow Testing:** This tracks how variables are used in the program. It checks that variables are declared, initialized, and used correctly to avoid issues related to data handling.

Advantages of White Box Testing

- ◆ **Thorough Testing:** The entire code is tested, making it very detailed and complete.
- ◆ **Code Optimization:** Helps find and remove unnecessary or faulty code, improving performance.
- ◆ **Early Bug Detection:** Can begin early in development, even before the user interface is ready.

- ◆ **Fits Well with Development Process:** Easily integrated into the software development lifecycle.
- ◆ **Finds Hidden Bugs:** Can catch complex issues that other types of testing might miss.
- ◆ **Better Test Coverage:** Allows testers to create strong test cases that cover all possible paths in the code.

Disadvantages of White Box Testing

- ◆ **Requires Coding Skills:** Testers need to understand programming and have access to the code.
- ◆ **May Ignore User Experience:** Focuses on how the code works internally, which can overlook external or usability issues.
- ◆ **Biased Testing:** Testers familiar with the code might miss problems due to assumptions.
- ◆ **Rewriting Tests is Time-Consuming:** Any code changes may require rewriting test cases.
- ◆ **Highly Dependent on Tester Expertise:** The quality of testing depends on the tester's knowledge and skills.
- ◆ **Cannot Find Missing Features:** It only checks what's in the code—not what should be there but is missing.
- ◆ **Risk of Production Errors:** Mistakes can still occur in the final product if issues are missed during testing.

b. Black Box Testing

Black-Box Testing is a type of software testing in which the tester is not concerned with the internal knowledge or implementation details of the software but rather focuses on validating the functionality based on the provided specifications or requirements. Example: Entering a username and password on a login page to check if the system correctly grants or denies access.

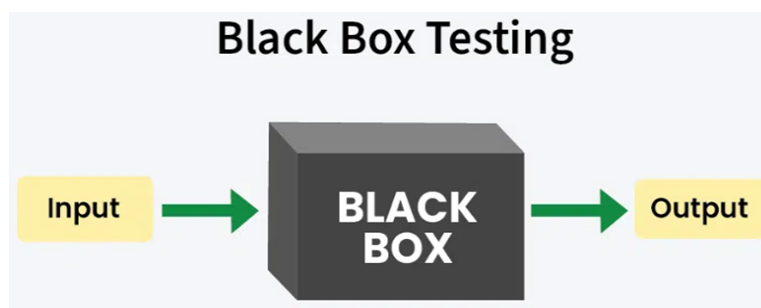


Fig 4.3.3 Black Box testing

Several techniques are used in black box testing to design effective test cases:

1. **Equivalence Partitioning** : It is a black box testing technique that divides input data into distinct partitions or groups typically categorized as valid

and invalid. The main idea is to test just one representative value from each group, under the assumption that all values within a group will be treated similarly by the system. This approach helps reduce the number of test cases while maintaining effective coverage. For example, if the valid input age range is 18 to 60, testers might choose 25 as a representative valid input, and 17 and 65 as representative invalid inputs, instead of testing every possible value within and outside the range.

2. **Boundary Value Analysis (BVA)** : It is a black box testing technique that concentrates on testing the edge values at the boundaries of input ranges, where errors are most likely to occur. Instead of testing values from the middle of an input range, BVA targets the minimum and maximum limits, as well as just outside those limits. For example, if an input range is defined as 1 to 100, BVA would suggest testing the values 0, 1, and 2 at the lower boundary, and 99, 100, and 101 at the upper boundary to verify how the system handles boundary conditions.
3. **Decision Table Testing** : It is a black box testing technique that uses a tabular format to represent various combinations of input conditions and their corresponding expected outputs or actions. It is particularly useful when the system behavior depends on multiple inputs or rules, as it helps ensure that all possible decision logic paths are considered and tested systematically. For example, in an online booking form where pricing is influenced by whether the user is a “member” and whether a “discount code” is applied, a decision table would map all possible combinations of these inputs and their respective outcomes, ensuring comprehensive test coverage.
4. **State Transition Testing** : It is a black box testing technique used to evaluate how software behaves as it transitions between different states in response to specific events or inputs. It is particularly effective for systems where outputs depend on the current state and the event that occurs. The goal is to verify both valid and invalid state transitions to ensure the system behaves correctly in all scenarios. For example, in an ATM system, the machine might move from the “Idle” state to “Card Inserted” and then to “Authenticated.” State transition testing would involve checking these transitions as well as invalid ones, such as attempting authentication without inserting a card.
5. **Error Guessing** : It is a black box testing technique that relies on the tester's intuition, experience, and domain knowledge to anticipate areas where defects are likely to occur. Instead of following formal rules or techniques, the tester makes educated guesses about problematic inputs or scenarios that may not be covered by structured methods. The main purpose is to uncover hidden bugs that standard testing might overlook. For instance, a tester might suspect that a login page could fail when blank inputs are submitted or when special characters are used, and will design tests accordingly to catch such potential issues.

Advantages of Black Box Testing

- ◆ The tester does not need to have more functional knowledge or programming skills to implement the Black Box Testing.



- ◆ It is efficient for implementing the tests in the larger system.
- ◆ Tests are executed from the user's or client's point of view.
- ◆ Test cases are easily reproducible.
- ◆ It is used to find the ambiguity and contradictions in the functional specifications.

Disadvantages of Black Box Testing

- ◆ There is a possibility of repeating the same tests while implementing the testing process.
- ◆ Without clear functional specifications, test cases are difficult to implement.
- ◆ It is difficult to execute the test cases because of complex inputs at different stages of testing.
- ◆ Sometimes, the reason for the test failure cannot be detected.
- ◆ Some programs in the application are not tested.
- ◆ It does not reveal the errors in the control structure.
- ◆ Working with a large sample space of inputs can be exhaustive and consumes a lot of time.

c. Gray Box Testing

Gray Box Testing is a software testing technique that is a combination of the Black Box Testing technique and the White Box Testing technique. In the Black Box Testing technique, the tester is unaware of the internal structure of the item being tested and in White Box Testing the internal structure should be known to the tester.



Fig 4.3.4 Gray Box

4.3.4 Test Case Design

Test Case Design is the process of creating a detailed set of conditions and steps that will be used to verify whether a particular feature or function of software works as expected. A well-designed test case ensures consistency, traceability, and thorough testing of the application. Here's what a typical test case includes:

- ◆ Test Case ID: A unique identifier assigned to each test case for easy tracking and reference.

- ◆ **Description:** A brief explanation of what the test case is intended to validate.
- ◆ **Preconditions:** Any conditions or setup that must be met before executing the test, such as user login or data availability.
- ◆ **Input Data:** The data to be used during the test, including parameters or values that will be entered.
- ◆ **Steps to Execute:** A sequence of actions to perform the test, written clearly and concisely.
- ◆ **Expected Result:** The anticipated outcome if the software functions correctly, based on requirements.
- ◆ **Actual Result:** The outcome observed after executing the test.
- ◆ **Status (Pass/Fail):** Indicates whether the actual result matches the expected result. If they match, the test is marked as “Pass”; otherwise, it's “Fail.”

4.3.5 Test Management Tools

Test Management Tools are software applications that help manage and streamline the testing process in software development. These tools assist in planning, designing, executing, tracking, and reporting testing activities, ensuring that the testing process is organized, efficient, and traceable.

Key Functions of Test Management Tools:

1. **Test Planning:** Define the scope, schedule, objectives, and resources for testing.
2. **Test Case Management:** Create, store, organize, and manage test cases and test suites.
3. **Requirement Mapping:** Link test cases to requirements to ensure complete coverage.
4. **Test Execution:** Run tests manually or automatically and log the outcomes.
5. **Defect Tracking Integration:** Connect with bug tracking tools like JIRA or Bugzilla to report and monitor defects.
6. **Progress Tracking and Reporting:** Generate reports and dashboards on test progress, coverage, and status.
7. **Team Collaboration:** Allow testers, developers, and managers to collaborate and stay updated in real time.

Examples of Popular Test Management Tools:

- ◆ **TestRail** – A web-based tool for managing test cases and test runs.
- ◆ **Zephyr** – A flexible test management tool that integrates well with JIRA.

- ◆ qTest – Provides test case management, automation support, and real-time reporting.
- ◆ PractiTest – Offers test case management and visualization of test data.
- ◆ TestLink – An open-source tool for test case creation, execution, and requirement tracking.

4.3.6 The Defect Life Cycle

The Defect Life Cycle (also known as Bug Life Cycle) represents the journey of a defect or bug from its identification to its resolution and closure. It defines the various stages a defect goes through during its lifetime in the software development and testing process. This life cycle helps ensure that no defect is lost or ignored and that issues are properly tracked and managed until resolution.

Here's a breakdown of each stage:

1. **New:** The tester logs a newly discovered defect. It is recorded with details such as steps to reproduce, severity, screenshots, etc.
2. **Assigned:** The defect is assigned to a developer by the lead or manager for investigation and resolution.
3. **Open:** The developer has started working on fixing the defect. It is under active analysis or correction.
4. **Fixed:** The developer has resolved the issue and marked the defect as fixed. The fix is ready to be verified by the tester.
5. **Retest:** The tester re-executes the relevant test cases to confirm that the defect has been successfully fixed.
6. **Verified:** If the defect no longer appears during retesting, the tester marks it as verified.
7. **Closed:** Once verified, the defect is marked as closed, indicating that it has been resolved and confirmed.
8. **Reopened (if needed):** If the issue reappears or wasn't properly fixed, the tester reopens the defect, and the cycle continues from the assigned/open stage.

4.3.7 Challenges in Software Testing

Software testing, while essential for delivering high-quality applications, comes with its own set of challenges. These challenges can arise from the complexity of software systems, resource limitations, and the rapid pace of development. Here are some common challenges in software testing:

1. **Complexity of Software :** As software becomes more intricate and involves multiple systems, technologies, and user environments, it becomes difficult to test every component thoroughly. Managing dependencies and interactions

between various modules or platforms can lead to incomplete test coverage.

Solution: Using testing frameworks, modular test cases, and automation tools can help manage complexity.

- 2. Time Constraints :** In Agile and DevOps environments, software is developed and delivered at a rapid pace. The constant pressure to meet deadlines can result in insufficient time for thorough testing.

Solution: Implementing automated testing and continuous integration can speed up the testing process, allowing more frequent and efficient testing cycles.

- 3. Changing Requirements :** In agile environments, where requirements can change frequently, it's challenging to maintain consistent and updated test cases. Changes in functionality or design can render test cases obsolete or require constant modifications.

Solution: Continuous communication between developers, testers, and stakeholders ensures alignment, and maintaining test case flexibility can help address changing requirements.

- 4. Inadequate Test Coverage :** Achieving comprehensive test coverage is often difficult, especially when the software has large amounts of data, many possible user interactions, or complex business rules. Missing test scenarios can lead to undiscovered defects.

Solution: Using tools for test case management and employing techniques like boundary value analysis and equivalence partitioning can help maximize coverage.

- 5. Testing for Performance and Scalability :** Ensuring the software's performance under load, stress, and real-world usage can be complex. Inadequate testing for performance issues can lead to crashes, slowdowns, and poor user experience.

Solution: Performance testing tools (e.g., JMeter, LoadRunner) and stress testing should be incorporated early in the testing phase to identify scalability issues.

- 6. Handling Test Data :** Test data can often be difficult to create and maintain, especially for scenarios like database testing, performance testing, or testing edge cases. Sensitive data must be handled properly for privacy and security concerns.

Solution: Test data generation tools, anonymization techniques, and creating reusable test datasets can help solve this issue.

- 7. Integration and Compatibility Testing :** With modern software running on different platforms, browsers, and devices, ensuring compatibility and seamless integration between different modules, systems, and external applications is a major challenge.

Solution: Cross-platform testing tools (like Selenium or Appium) and establishing a continuous integration system can help automate integration and compatibility checks.



- 8. Automation Challenges :** While automation can speed up the testing process, it requires significant investment in terms of time, tools, and skills. Furthermore, not all tests can be automated (e.g., usability or exploratory tests).

Solution: Prioritize automating repetitive and time-consuming tests, while leaving exploratory and usability tests to manual testing.

- 9. Defect Management:** Managing defects, especially in large projects, can be overwhelming. Identifying, reporting, and tracking defects efficiently through the various stages of development and testing can be cumbersome.

Solution: Use defect tracking tools (like JIRA or Bugzilla) to streamline defect reporting, tracking, and resolution, ensuring proper communication between developers and testers.

- 10. Test Environment Issues :** Setting up and maintaining test environments can be time-consuming. Differences between the test and production environments can cause tests to fail unexpectedly or produce inconsistent results.

Solution: Automated environment setups using containerization (e.g., Docker) and cloud services can provide consistency and scalability.

- 11. Human Resource Constraints :** Testers may have limited resources, such as insufficient team size, skill sets, or expertise, which can impact the effectiveness of the testing process.

Solution: Adequate training, resource planning, and leveraging automation can alleviate pressure on testing teams.

- 12. Lack of Communication Between Teams :** In large projects or organizations, poor communication between development, testing, and business teams can lead to misalignment of objectives and priorities.

Solution: Foster a collaborative culture, encourage frequent communication, and use project management tools to ensure all stakeholders are aligned on testing goals.

Recap

- ◆ **Testing :** Software testing is an important process in the Software Development Lifecycle(SDLC). It involves verifying and validating that a Software Application is free of bugs, meets the technical requirements set by its Design and Development, and satisfies user requirements efficiently and effectively.
- ◆ **Verification:** Verification is the process of evaluating the intermediate work products during the software development lifecycle to ensure they meet the specified requirements. It focuses on being process-oriented and answers the critical question: "Are we building the product right?"

- ◆ Validation : Validation is the process of evaluating the final software product to ensure that it meets the business needs and user requirements. It is product-oriented and answers the key question: "Are we building the right product?"
- ◆ Software Testing Life Cycle (STLC) is a systematic process that defines a series of phases or stages followed while testing software to ensure its quality and functionality.
- ◆ Types of Software Testing
- ◆ Based on Software Approach/ Method
- ◆ Manual Testing: Manual testing is a technique to test the software that is carried out using the functions and features of an application.
- ◆ Automation Testing: Automated Testing means using special software for tasks that people usually do when checking and testing a software product.
- ◆ Based on the goal of testing:
- ◆ Functional Testing : Functional testing is focused on verifying that the software behaves according to the defined requirements and specifications
- ◆ Non-Functional Testing: Non-functional testing focuses on how well the software performs beyond its functional behavior.
- ◆ Based on Scope/Levels of Software Testing
- ◆ Unit Testing : focuses on testing individual units or components of the software in isolation to ensure they function as intended
- ◆ Integration Testing : examines the interaction between integrated modules or components. Its main objective is to identify interface defects and verify the correct flow of data between different units.
- ◆ System Testing : Involves testing the entire integrated software system as a whole to validate that it meets the specified functional and non-functional requirements.
- ◆ Acceptance Testing : It is the final level of testing, conducted to determine whether the software meets the business requirements and is ready for deployment.
- ◆ Based on Testing Techniques:
- ◆ White Box Testing : White Box Testing is a technique where the internal structure of a system like design and code is manually checked. In this method, the development team goes through the code line by line to ensure everything works as intended.

- ◆ **Black-Box Testing:** Black-Box Testing is a type of software testing in which the tester is not concerned with the internal knowledge or implementation details of the software but rather focuses on validating the functionality based on the provided specifications or requirements.
- ◆ **Gray Box Testing:** Gray Box Testing is a software testing technique that is a combination of the Black Box Testing technique and the White Box Testing technique.
- ◆ **Test Case Design :** Test Case Design is the process of creating a detailed set of conditions and steps that will be used to verify whether a particular feature or function of software works as expected.
- ◆ **Test Management Tools** are software applications that help manage and streamline the testing process in software development.
- ◆ **The Defect Life Cycle :** The Defect Life Cycle (also known as Bug Life Cycle) represents the journey of a defect or bug from its identification to its resolution and closure.
- ◆ **Challenges in Software Testing :** These challenges can arise from the complexity of software systems, resource limitations, and the rapid pace of development.

Objective Type Questions

1. Which of the following is the main purpose of software testing?
2. Which testing is performed first in the software development process?
3. What does Validation Testing check?
4. Which testing focuses on testing the interfaces and interaction between modules?
5. Which type of testing is performed by the end-user before the product release?
6. System Testing is done after
7. What is the goal of Regression Testing?
8. Which of the following is NOT a type of software testing?
9. Which testing technique focuses only on inputs and outputs?
10. Which testing technique combines Whitebox and Blackbox?

Answers to Objective Type Questions

1. To find errors
2. Unit testing
3. Customer requirements
4. Integration testing
5. User Acceptance Testing (UAT)
6. A and B
7. To ensure changes don't break existing functionality
8. Compilation testing
9. Blackbox
10. Graybox

Assignments

1. Define Software Testing. Explain its importance in the software development life cycle (SDLC).
2. Differentiate between Verification and Validation with suitable examples.
3. Explain the differences between Unit Testing, Integration Testing, System Testing, and Acceptance Testing.
4. What is Regression Testing? Why is it important when maintaining software applications?
5. Describe Black Box Testing and White Box Testing. Provide real-world scenarios where each type would be used.
6. What are the key steps involved in writing effective test cases? Illustrate with an example of an online shopping cart.
7. Discuss the role of Automation Testing in modern software development. Mention tools commonly used for automation.
8. Explain the significance of Performance Testing. What parameters are checked during performance testing of a web application?
9. What is Security Testing? List and explain at least three techniques used to identify security vulnerabilities.
10. Prepare a brief test plan for a mobile banking app, covering test objectives, testing types, and key test cases.

Suggested Reading

1. Steve McConnell – *Code Complete: A Practical Handbook of Software Construction* (2nd Edition), Publisher: Microsoft Press, 2004
2. Frederick P. Brooks Jr. – *The Mythical Man-Month: Essays on Software Engineering*, Publisher: Addison-Wesley, 1995
3. Tom DeMarco & Timothy Lister – *Peopleware: Productive Projects and Teams*, Publisher: Addison-Wesley, 2013
4. Gene Kim, Jez Humble, Patrick Debois, and John Willis – *The DevOps Handbook*, Publisher: IT Revolution Press, 2016

Reference

1. Sommerville, Ian. *Software Engineering* (10th Edition), Pearson Education, 2015.
2. Pressman, Roger S., and Maxim, Bruce R., *Software Engineering: A Practitioner's Approach* (8th Edition), McGraw-Hill Education, 2014.
3. Rajib Mall, *Fundamentals of Software Engineering* (5th Edition), PHI Learning Pvt. Ltd., 2018.
4. Pankaj Jalote, *An Integrated Approach to Software Engineering* (3rd Edition), Springer, 2005.
5. Fenton, Norman E., and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous and Practical Approach* (2nd Edition), Brooks/Cole, 1998.



Deployment

Learning Outcomes

The learners will be able to:

- ◆ understand the Deployment Process
- ◆ recognize Different Deployment Environments
- ◆ apply Deployment Strategies
- ◆ evaluate Post-Deployment Activities

Prerequisites

Consider a school management system, once the software is developed and thoroughly tested, it is deployed in the school's IT infrastructure or on a secure cloud server. This deployment includes creating user accounts for teachers, students, parents, and administrators, as well as configuring features like attendance tracking, timetable scheduling, fee management, and report generation. Data such as student records, class lists, and exam schedules are uploaded. Training may be provided to teachers and administrative staff to help them navigate the system efficiently. Once deployed, the system automates daily school operations, making administration more efficient and transparent.

Keywords

Configuration, Deployment Strategy, Consistency, Traceability, Canary Deployment

Discussion

4.4.1 Deployment in Software Engineering

Deployment in Software Engineering refers to the process of delivering a developed and tested software application to the end-users or production environment. It involves several key activities such as releasing, installing, configuring, and making the software operational for its intended users. Deployment is a crucial phase in the Software Development Life Cycle (SDLC), ensuring that the software product is accessible, functional, and reliable in real-world conditions.

4.4.2 Key Steps in Deployment Process

4.4.2.1 Release Management

This step involves planning when and how the school management software will be made available to the school. For instance, the software provider schedules a release before the academic year starts to ensure it's ready for use by teachers, students, and administrators. Proper version control ensures that the correct and stable version of the software is released at the right time, avoiding disruption during school operations.

4.4.2.2 Build and Packaging

Once the software is developed, the source code is compiled, and necessary files are bundled into a deployable format. For example, the school management software may be packaged into a web application with an installer or deployed as a container (if cloud-based). This ensures that the complete, working application is ready to be installed on the school's server or cloud environment.

4.4.2.3 Environment Setup

Before deploying, different environments must be prepared. For a school, this might include:

- ◆ A development environment for ongoing improvements
- ◆ A testing environment where updates can be checked
- ◆ A production environment where the real school data is used

The deployment team sets up servers, databases (e.g., student records), network configurations, and user access levels for different roles like teachers, admins, and students.

4.4.2.4 Installation and Configuration

The software is now installed on the school's infrastructure (on-premises or in the cloud). During this phase, critical configurations are applied. For example, linking the system to the school's database, customizing the login for various user types, setting up the academic year, or enabling SMS/email alerts for parents.

4.4.2.5 Testing in Production Environment

After installation, a quick smoke test is performed to ensure everything works. For instance, testers may try logging in as a teacher, entering attendance, or generating a report card to confirm the system behaves correctly in the live environment.

4.4.2.6 Monitoring and Maintenance

Once live, the system is monitored using performance and error-tracking tools. If a teacher reports an error in the timetable or if the fee module crashes, the IT team can identify and fix it quickly. Routine maintenance includes updating features (e.g., adding new exam formats), fixing bugs, and applying security patches to protect sensitive student data.

4.4.3 Importance of Software Deployment

Software deployment in software engineering is the process of taking a fully developed and tested software application and making it operational in the real-world environment where users can access and interact with it. It involves several key steps, including preparing the infrastructure, installing the software components, configuring settings, performing post-deployment testing, activating the system for users, and monitoring its performance. A rollback plan is also kept ready in case issues arise after deployment. For example, when a school management system is deployed, the software is installed on the school's server or cloud platform, configured to match the school's academic structure, and made accessible to teachers, students, and administrators. Teachers can begin marking attendance, students can check schedules, and administrators can manage fees thus, the software becomes a valuable tool that delivers real benefits. Deployment not only makes the system usable but also opens the door for feedback and regular updates, ensuring the software continues to meet users' evolving needs.

4.4.4 Deployment strategies

4.4.4.1 Manual Deployment

Manual deployment involves a human-driven process to copy, install, and configure software on the target systems. In the context of a school management system, this might mean the IT staff manually transferring files to the school server, setting up databases, and configuring access for users. Although simple for small-scale deployments, this method is time-consuming and prone to mistakes, such as missing files or incorrect settings. It's often used in smaller schools with limited technical infrastructure, but it's not ideal for frequent updates or complex setups.

4.4.4.2 Automated Deployment

Automated deployment uses tools like Jenkins, GitLab CI/CD, or Docker to streamline the deployment process. For a school, this means that when the software team pushes a new update (like adding a parent communication feature), the system automatically builds, tests, and deploys the update to the live environment without manual effort. This ensures faster, more reliable releases, reduces human errors, and allows the school to benefit from new features more efficiently.

4.4.4.3 Blue-Green Deployment

This strategy involves maintaining two identical environments blue (current live system) and green (new version). In a school scenario, the new version of the school management software perhaps with a new report card module is first deployed to the green environment. Once it's fully tested and verified, the traffic is switched from the blue to the green environment with minimal downtime. This ensures a safe and smooth transition, with the option to revert back quickly if issues arise.

4.4.4.4 Canary Deployment

Canary deployment rolls out new software features gradually to a small group of users. For example, a school might deploy a new online assignment submission feature only to the teachers of grade 10. Their feedback and usage data are monitored to identify any bugs or usability issues. If everything goes well, the feature is gradually expanded to more grades and finally to the whole school. This strategy minimizes risk by detecting problems early.

4.4.4.5 Rolling Deployment

In a rolling deployment, the software is updated on a few servers or systems at a time instead of all at once. For a large school network or a group of schools using the same management system, the update might begin with a few schools first. If the deployment is successful, it continues gradually across the rest. This method ensures continuous service availability and reduces the chance of widespread failure if something goes wrong.

4.4.5 Popular Deployment Tools

Table 4.4.1 Different deployment tools

Tool	Description
Jenkins	Open-source automation server used for continuous integration and continuous deployment (CI/CD). Supports various plugins.
GitLab CI/CD	Integrated tool within GitLab for automating builds, testing, and deployments, tightly integrated with Git repositories.
Docker	Containerization platform to package applications and their dependencies, ensuring consistent deployment across systems.
Kubernetes	Orchestration tool to automate deployment, scaling, and management of containerized applications (works well with Docker).

Ansible	Automation tool used for configuration management, application deployment, and task automation. Uses simple YAML syntax.
Chef	Configuration management tool to automate infrastructure and application deployment, using Ruby-based scripts.
Puppet	Automates the delivery and operation of software, commonly used for infrastructure management and deployment.
AWS CodeDeploy	A deployment service offered by Amazon Web Services for automated code deployment to servers, Lambda, and EC2 instances.
Octopus Deploy	Specifically designed for deployment automation, it supports .NET, Java, and other environments.
Azure DevOps	Microsoft's CI/CD and deployment solution integrated with Azure Cloud and Git repositories.

4.4.7 Software Deployment vs Software Release:

Software deployment involves the entire process of making the software available for use, including installation and configuration. Software release, on the other hand, specifically refers to the distribution of a new version or update to users. Deployment is the broader process that includes release activities.

Table 4.4.2 Comparison of Software Deployment vs Software Release

Parameters	Software Deployment	Software Release
Definition	The process of making software or updates usable in a particular environment is referred to as deployment.	A software release is when a particular version is made accessible to users outside of the development team.
Timing	The software may not always be available to end users even after deployment, which is a possible step in the release process. It can be carried either as part of standard maintenance procedures or in advance of a release.	When software is released, it becomes accessible to end users. Choosing when to release the programme to the public requires considerations that are frequently in line with marketing plans or user expectations.

Scope	The technical side of delivering and installing software is the main emphasis of deployment.	Release covers more ground than just deployment. It also includes documentation, marketing, and user communication
Goals	Ensuring that the programme is deployed accurately and effectively in the target environment is the main objective of deployment.	Delivering a software version that satisfies quality criteria and offers users value is the main objective of a release.

4.4.5 Configuration Management in Software Engineering

Configuration Management (CM) refers to the systematic process of handling changes in software, hardware, and documentation throughout the software development life cycle. It ensures that the software product maintains its integrity, consistency, and traceability during development, deployment, and maintenance.

4.4.6 Key Objectives of Configuration Management

Objective of configuration management are:

4.4.6.1 Version Control

Version control is the process of tracking and managing changes to software components such as source code, documentation, and configuration files. It allows multiple developers to work collaboratively without overwriting each other's work. In case of issues or regressions, developers can easily revert to a previous version of the software. Tools like Git help manage this process efficiently, ensuring a clear history of edits and facilitating teamwork in software projects.

4.4.6.2 Change Management

Change management involves a structured approach to handling modifications in a software system. It ensures that all changes whether minor bug fixes or major feature additions are properly evaluated, approved, and documented before implementation. This prevents untested or unauthorized updates from disrupting the software's stability. A good change management process minimizes risks and ensures only high-quality, verified changes reach the production environment.

4.4.6.3 Consistency

Consistency in configuration management means maintaining a uniform software environment across different stages of the software development life cycle development, testing, staging, and production. This ensures that the software behaves identically across all these environments, avoiding environment-specific bugs. For example, if a

school management system works well during testing but fails in production due to configuration differences, consistency practices can prevent such issues.

4.4.6.4 Traceability

Traceability refers to the ability to track every change made to the software who made it, why it was made, and when it was applied. This helps in understanding the history and context of changes, making debugging and audits more efficient. For instance, if a new update in a hospital management system causes billing errors, traceability allows the development team to quickly identify the change responsible and fix it based on documented reasons and timestamps.

4.4.7 Key Components of Configuration Management:

Key components of Configuration managements are

4.4.7.1 Configuration Items (CIs)

Configuration Items are the essential components that are tracked and managed within the configuration management process. These can include source code files, test scripts, system documentation, libraries, database schemas, hardware specifications, and more. Each CI is uniquely identified so that changes can be monitored and controlled over the software development lifecycle. For example, in a school management system, the login module's code, related user manuals, and associated test scripts would all be considered configuration items.

4.4.7.2 Version Control Systems (VCS)

Version Control Systems are tools used to track and manage changes made to configuration items over time. Tools like Git, Subversion (SVN), and Mercurial allow multiple developers to work on the same project while keeping a record of all modifications. These systems make it easy to revert to earlier versions, resolve conflicts, and understand the evolution of the software. For instance, if a teacher dashboard feature in a school software breaks after an update, Git can help identify when and how the code was changed.

4.4.7.3 Change Control Board (CCB)

The Change Control Board is a group of stakeholders (often including developers, testers, project managers, and clients) responsible for evaluating change requests. Their job is to review proposed changes, assess their impact, and approve or reject them based on criteria like risk, cost, and benefit. In a hospital management system, if a new feature is proposed for automating lab test scheduling, the CCB would decide whether it's appropriate to implement based on the system's current priorities and stability.

4.4.7.4 Configuration Baseline

A Configuration Baseline is a formally approved version of configuration items at a specific point in time, which serves as the foundation for further development. It acts as a reference point and ensures that everyone is working from a consistent and tested

version of the software. For example, before launching the exam module in a school ERP, the team may create a baseline version that has passed all required testing and reviews.

4.4.7.5 Audit and Status Reporting

Audits in configuration management are conducted to ensure that all configuration activities are being followed as planned, and that configuration items match the defined specifications. Status reporting keeps the team informed about the current state of configuration items, such as which version is deployed or what changes are pending approval. This transparency supports better decision-making and accountability. For instance, in a software upgrade cycle, audit logs and status reports help verify whether all modules of the student management system were updated successfully.

Recap

- ◆ **Deployment :** Deployment in Software Engineering refers to the process of delivering a developed and tested software application to the end-users or production environment.
- ◆ **Key Steps in Deployment Process:**
 - **Release Management :** This step involves planning when and how the school management software will be made available to the school.
 - **Build and Packaging :** Once the software is developed, the source code is compiled, and necessary files are bundled into a deployable format.
 - **Environment Setup :** Before deploying, different environments must be prepared. For a school, this might include:
 - **Installation and Configuration :** The software is now installed on the school's infrastructure (on-premises or in the cloud).
 - **Testing in Production Environment :** After installation, a quick smoke test is performed to ensure everything works.
 - **Monitoring and Maintenance :** Once live, the system is monitored using performance and error tracking tools.
- ◆ **Deployment strategies**
 - **Manual Deployment :** Manual deployment involves a human-driven process to copy, install, and configure software on the target systems.
 - **Automated Deployment :** Automated deployment uses tools like Jenkins, GitLab CI/CD, or Docker to streamline the deployment process.
 - **Blue-Green Deployment :** This strategy involves maintaining two identical environments: blue (current live system) and green (new version).

- Canary Deployment: Canary deployment rolls out new software features gradually to a small group of users.
 - Rolling Deployment : In a rolling deployment, the software is updated on a few servers or systems at a time instead of all at once.
 - ◆ Configuration Management : Configuration Management (CM) refers to the systematic process of handling changes in software, hardware, and documentation throughout the software development life cycle.
 - ◆ Key Objectives of Configuration Management
- Objective of configuration management are:
- Version Control: Version control is the process of tracking and managing changes to software components such as source code, documentation, and configuration files.
 - Change Management : Change management involves a structured approach to handling modifications in a software system.
 - Consistency : Consistency in configuration management means maintaining a uniform software environment across different stages of the software development life cycle development, testing, staging, and production.
 - Traceability : Traceability refers to the ability to track every change made to the software who made it, why it was made, and when it was applied.
- ◆ Key Components of Configuration Management:Key components of Configuration managements are
- Configuration Items (CIs) : Configuration Items are the essential components that are tracked and managed within the configuration management process.
 - Version Control Systems (VCS) : Version Control Systems are tools used to track and manage changes made to configuration items over time.
 - Change Control Board (CCB) : The Change Control Board is a group of stakeholders (often including developers, testers, project managers, and clients) responsible for evaluating change requests.
 - Configuration Baseline : A Configuration Baseline is a formally approved version of configuration items at a specific point in time, which serves as the foundation for further development.
 - Audit and Status Reporting : Audits in configuration management are conducted to ensure that all configuration activities are being followed as planned, and that configuration items match the defined specifications.

Objective Type Questions

1. Which phase of the Software Development Life Cycle (SDLC) involves delivering the product to the customer and obtaining feedback?
2. What is the primary goal of Continuous Deployment in DevOps?
3. Which of the following is NOT a common deployment strategy?
4. In a Blue-Green Deployment strategy, what is the primary purpose of maintaining two identical environments?
5. Which tool is commonly used for automating the deployment process in a DevOps pipeline?
6. What is the first basic test done after installation?
7. What is applied regularly to improve performance and security?
8. What term describes making software operational for users?
9. What is the alternative environment kept ready in blue-green deployment?
10. Which deployment strategy uses partial user rollout?

Answers to Objective Type Questions

1. Deployment
2. To automate the deployment process without manual intervention
3. Waterfall Deployment
4. To ensure zero downtime during deployment
5. Jenkins
6. Smoke
7. Maintenance
8. Deployment
9. Green
10. Canary

Assignments

1. Explain the role of deployment in the Software Development Life Cycle (SDLC). Why is it crucial for the success of a software product?
2. Describe different software deployment strategies such as Blue-Green Deployment, Rolling Deployment, and Canary Deployment. Compare their advantages and disadvantages.
3. What is Continuous Deployment? How does it differ from Continuous Integration and Continuous Delivery? Explain with examples.

4. Discuss the significance of deployment automation tools (like Jenkins, Ansible, Docker, Kubernetes) in modern software deployment processes. Provide real-world use cases for each.
5. Write a short essay on the challenges faced during the deployment of large-scale applications and suggest possible solutions to overcome them.
6. Explain the concept of Infrastructure as Code (IaC) and its role in software deployment. How does it improve deployment consistency and scalability?
7. Consider a scenario where a new e-commerce website is being deployed. Draft a detailed deployment plan covering pre-deployment checks, deployment steps, rollback strategies, and post-deployment testing.
8. Discuss the impact of poor deployment practices on software reliability and user satisfaction. Give examples of real-world cases where improper deployment led to failure or downtime.

Suggested Reading

1. Steve McConnell – *Code Complete: A Practical Handbook of Software Construction* (2nd Edition), Publisher: Microsoft Press, 2004
2. Frederick P. Brooks Jr. – *The Mythical Man-Month: Essays on Software Engineering*, Publisher: Addison-Wesley, 1995
3. Tom DeMarco & Timothy Lister – *Peopleware: Productive Projects and Teams*, Publisher: Addison-Wesley, 2013
4. Gene Kim, Jez Humble, Patrick Debois, and John Willis – *The DevOps Handbook*, Publisher: IT Revolution Press, 2016

Reference

1. Sommerville, Ian. *Software Engineering* (10th Edition), Pearson Education, 2015.
2. Pressman, Roger S., and Maxim, Bruce R., *Software Engineering: A Practitioner's Approach* (8th Edition), McGraw-Hill Education, 2014.
3. Rajib Mall, *Fundamentals of Software Engineering* (5th Edition), PHI Learning Pvt. Ltd., 2018.
4. Pankaj Jalote, *An Integrated Approach to Software Engineering* (3rd Edition), Springer, 2005.
5. Fenton, Norman E., and Pfleeger, Shari Lawrence, *Software Metrics: A Rigorous and Practical Approach* (2nd Edition), Brooks/Cole, 1998.

BLOCK 5

Object - Oriented Design Concepts





Unified Modeling Language

Learning Outcomes

The learner will be able to:

- ◆ familiarize the role of UML in software development life cycle.
- ◆ identify the main differences between structural and behavioral UML diagrams.
- ◆ summarize the main concepts of object-oriented design using UML.
- ◆ list the steps involved in creating a UML diagram.
- ◆ name the key contributors to the development of UML.
- ◆ understand the UML notations to represent the structure and behavior of a system through appropriate diagrams.

Prerequisites

UML (Unified Modeling Language) is a standardized visual language used to model, design, and document the structure and behavior of software systems. Through the use of UML, complex systems can be broken down into simpler, more understandable parts using a variety of visual diagrams. For example, in a banking application, class diagrams can depict elements such as accounts, customers, and transactions. Use case diagrams highlight how users interact with the system, while sequence diagrams demonstrate the flow of operations and data across components. This visual approach helps clarify how different parts of a system function and interact, making it easier to identify design flaws early and ensure efficient system behavior.

UML also plays a key role in improving communication and collaboration across development teams. Since it is a widely accepted and standardized language, UML allows developers, project managers, testers, and clients to share a common understanding of system architecture and functionality. For instance, when designing a shopping cart for an e-commerce platform, UML diagrams can help all stakeholders visualize how the feature should work, reducing the risk

of miscommunication. Furthermore, it supports system analysis and continuous improvement by allowing teams to explore and refine design alternatives visually. By using UML, software development becomes more organized, transparent, and aligned with user needs. UML helps connect programming concepts with real-world system design, making abstract ideas more concrete and practical. Analyzing UML diagrams helps learners break down complex problems into manageable parts, encouraging logical and analytical thinking

Keywords

Visual Modeling, Object-Oriented Programming (OOP), Object-Oriented Design, Inheritance, Polymorphism, Encapsulation, Abstraction, UML Tools. Aggregation, Generalization.

Discussion

Unified Modeling Language (UML) is a general-purpose visual modeling language used to represent the design of a system. The main purpose of UML is to provide a clear, standardized way to visualize the components of a software system. For example, if a team is designing an online shopping application, they might use UML diagrams to show how users interact with the system, how the system processes payments, and how inventory is updated after a purchase. These diagrams help developers, testers, project managers, and even clients. While UML is not a programming language, it serves as a powerful visual tool for understanding and documenting software systems. Visualizing saves a lot of time and helps to understand how a system structure, user interactions, processes etc.

UML is a standardized general-purpose visual modeling language in the field of Software Engineering. It is used for specifying, visualizing, constructing, and documenting the primary artifacts of the software system. It describes the working of both the software and hardware systems. It was developed by Grady Booch, Ivar Jacobson, and James Rumbaugh.

5.1.1 Aspects of UML

- ◆ **Structural Aspect-** Structural aspect focuses on the static parts of a system, such as classes, objects, and their relationships. It describes the arrangement of different parts within the system
- ◆ **Behavioral Aspect-** The behavioral aspect deals with how the system behaves over time. It includes the flow of activities, interactions between objects, and system responses.
- ◆ **Architectural Aspect-** The architectural aspect gives a high-level view of the structure of the system. It helps in organizing different parts of the system

and shows how everything fits together. These three aspects work together to give a complete understanding of the system.

5.1.2 Object-Oriented Concepts Used in UML Diagrams

The object-oriented concepts in UML diagrams include classes, objects, inheritance, abstraction, encapsulation, and polymorphism. These concepts help to represent real-world entities and their relationships in a structured way. OOPS concepts improve the modularity and clarity by providing an organized approach to model the complex systems.

- ◆ **Class:** A class in UML is a visual representation of a software entity that groups together related data (attributes) and functions (operations) into a single unit.
- ◆ **Objects:** An object in UML is a specific, identifiable instance of a class that exists during program execution. It contains real values for the attributes defined in the class and can perform the operations described by the class.
- ◆ **Inheritance:** Inheritance in UML represents a mechanism where one class (called a subclass or child class) inherits attributes and operations from another class (called a superclass or parent class). It models the "is-a" relationship and is a core concept of object-oriented design.
- ◆ **Abstraction:** Abstraction in UML refers to the process of hiding internal implementation details and showing only the essential features of a system component.
- ◆ **Encapsulation:** Encapsulation is an important idea in object-oriented programming and is shown in UML through classes. It means grouping related data and functions together inside a class. The class hides its internal data and only allows access or changes through public methods.
- ◆ **Polymorphism:** Polymorphism is a core principle of object-oriented programming (OOP), and it is effectively modeled in UML (Unified Modeling Language). The term "polymorphism" means "many forms." In software, it refers to the ability of different classes to respond to the same operation or message in different ways. This allows a program to be more flexible and easier to extend or modify.

5.1.3 Steps to create UML Diagrams

Unified Modeling Language diagrams are used to visually represent the structure, behavior, and architecture of software systems. The process of creating Unified Modeling Language diagrams generally involves the following steps:

Step 1: Identify the Purpose

UML diagrams can be used for various purposes. These include collecting system requirements, designing system architecture, and documenting class relationships. Therefore, it is important to clearly understand the specific goal before creating a UML diagram.



Step 2: Identify elements and relationships

Identify the key elements such as class, object, use case or component based on the purpose of the diagram. Then, determine how these elements are related to each other.

Step 3: Gather requirements and information

Collect the necessary information about the system that include system specifications, functional requirements, or design constraints.

Step 4: Create a rough sketch

Starting with a quick sketch on paper or a whiteboard can be very useful. This initial draft provides a clear overview before moving to a UML modeling tool. It helps to understand how the different elements are positioned and connected.

Step 5: Choose a UML modeling tool

Choose a UML modeling tool that meets your specific requirements. There are many options available, both online and offline. These tools come with various features to design and adjust UML diagrams effectively.

Step 6: Create the diagram

Create a new project or diagram using the UML modeling tool of your choice. Start by adding components to the diagram and connect these elements using appropriate relationships.

Step 7: Assign properties to elements

Provide each element in the diagram with the necessary properties and details. Make sure each component accurately reflects its role in the system. For example, in class diagrams, define key attributes and methods for each class.

Step 8: Add Annotations and Comments

Annotations and comments are valuable tools that can significantly enhance the clarity and maintainability of UML diagrams. Annotations are text labels or notes added to explain specific elements or relationships in the diagram. Comments are usually attached to elements to describe logic, purpose, or special conditions.

Step 9: Validate and review

Review the diagram to ensure that it is complete and accurate. Verify that the elements, constraints, and relationships correctly represent the intended system or process.

Step 10: Refine and Iterate

Make improvements to the diagram based on feedback and further insights. UML diagrams are typically created in multiple iterations, gradually becoming more precise and complete.

5.1.4 Tools for creating UML Diagrams

There are several tools available for creating Unified Modeling Language diagrams. These diagrams are widely used in software development to visually represent system architecture, design, and implementation. These are some of the popular tools used for creating UML diagrams.

1. Lucidchart: Lucidchart is an online tool used to make diagrams. It helps to create UML diagrams easily. It is good for teamwork because many people can work on the same diagram at the same time..
2. Diagrams.net: Access this UML tool through any web browser. It allows for diagram creation even without an internet connection. It includes a complete set of UML symbols.
3. SmartDraw: SmartDraw is an online tool for creating diagrams. It provides a broad selection of templates for UML diagrams.
4. Visual Paradigm: Supporting a wide variety of UML diagrams, Visual Paradigm offers a complete toolset for software development, accessible via its online and desktop platforms.
5. StarUML: StarUML is an open-source UML modeling tool with a user-friendly interface.

5.1.5 Building blocks of UML

UML has three main building blocks such as things, relationships, and diagrams. These are used to model and design a software system.

1. Things

Things are the most important building blocks in UML. They represent the fundamental elements used to model a system. Things can be classified into four types: structural, behavioral, grouping, and annotational elements.

a. Structural things- Structural things in UML represent the static parts of a system. The main types of structural things in UML are:

- ◆ **Class:** class represents a group of objects that share the same responsibilities, attributes, and behaviors.

Class
Attributes
Operations

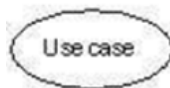
- ◆ **Interface:** An interface is a set of operations that a class must follow or implement. It shows what a class should be able to do.



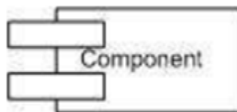
- ◆ **Collaboration:** It shows how different elements work together through interaction to achieve a specific task or behavior.



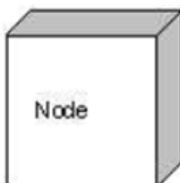
- ◆ **Use case:** A use case shows how a system performs a group of actions to achieve a specific purpose.



- ◆ **Component:** A component represents a physical part of the system that performs a specific function.

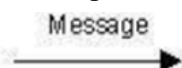


- ◆ **Node:** A node is a physical element that exists during runtime.

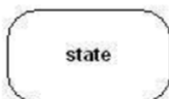


- b. Behavioral things:** It captures the dynamic aspects of UML models.
The key behavioral things include:

- ◆ **Interaction:** It is a behavior where elements exchange messages to complete a specific task.



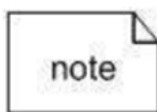
- ◆ **State machine:** It shows how an object moves through different states triggered by external events.



- c. **Grouping Things** - It is used to organize elements in a UML model. Package is the only grouping element that combines both structural and behavioral things.



- d. **Annotational things** - It is used to capture remarks, descriptions, and comments about UML model elements. The only annotational thing available is a Note, which is used to add comments, constraints, and other details to UML elements.



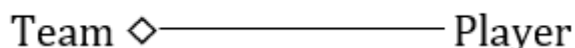
2. Relationship

UML defines several types of relationships to show how elements in a model are connected. These relationships help to describe both the structure and behavior of a system. The main types of UML relationships are:

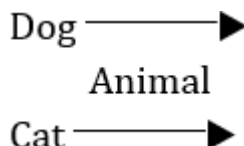
- ◆ **Association**- It represents a basic link between two classes or objects.. It also shows how instances of one class are connected to instances of another. Example: A "Student" is associated with a "Course".



- ◆ **Aggregation**: A "whole-part" relationship where the part can exist independently of the whole. Example: A team has players, but players can exist without the team.



- ◆ **Generalization**: Generalization: Shows a parent-child relationship between classes. Example: A dog and cat are types of animals.



- ◆ **Dependency**: A weaker relationship showing that one element uses or depends on another.

Report - - - - - ➔ DataSource

- ◆ Composition: A stronger "whole-part" relationship where the part cannot exist independently.

House ◆ ————— Room

3. Diagrams

UML diagrams are the ultimate output of the entire process. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

5.1.6 Advantages of UML

1. It is a general-purpose modeling language.
2. It shows how a system works using diagrams and pictures.
3. It is used mainly in object-oriented design and analysis.
4. It helps to plan and design software systems clearly.
5. It can also be used to visualize outside of software, like in factories or business workflows.
6. It makes it easier for developers, designers, and clients to understand the system.
7. There are many tools available to draw UML diagrams.
8. It helps to bridge the gap between technical and non-technical users.

5.1.7 Limitations of using UML in Software Engineering

1. UML can become large and complex in big projects.
2. Managing diagrams takes time and effort.
3. Sometimes developers may ignore models and focus only on code.
4. It does not generate complete working code.
5. Some tools are expensive or difficult to use.

Recap

- ◆ UML stands for Unified Modeling Language.
- ◆ It is a general-purpose visual language used to model software systems.

- ◆ UML helps in designing, visualizing, and documenting system components.
- ◆ It uses diagrams to show how a system works and how users interact with it.
- ◆ UML is not a programming language but supports object-oriented design.
- ◆ It was developed by Grady Booch, Ivar Jacobson, and James Rumbaugh.
- ◆ UML is useful for developers, testers, project managers, and clients.
- ◆ It saves time by providing a clear visual understanding of system structure and behavior.
- ◆ UML includes three key aspects: structural, behavioral, and architectural.
- ◆ Structural aspects deal with static parts like classes and objects.
- ◆ Behavioral aspects show how the system acts and interacts over time.
- ◆ Architectural aspects provide a high-level view of the entire system design.
- ◆ UML supports object-oriented concepts such as classes, objects, inheritance, encapsulation and polymorphism.
- ◆ UML tools like Lucidchart, Diagrams.net, and StarUML help to create accurate diagrams.
- ◆ Diagrams are refined by assigning properties, adding comments, and reviewing them.
- ◆ UML diagrams are improved over multiple iterations to ensure accuracy.
- ◆ UML helps in organizing and planning software projects effectively.
- ◆ It improves team communication and system understanding.
- ◆ But UML also has limitations—it can be complex and time-consuming.
- ◆ UML has three main building blocks: things, relationships, and diagrams, which are used to model and design software systems.
- ◆ Things include structural, behavioral, grouping, and annotational elements that help represent the system's parts and behavior.
- ◆ Relationships in UML show how elements are connected, using types like association, aggregation, composition, generalization, and dependency.
- ◆ UML diagrams combine all elements and relationships to visually represent the system's design and functionality.

Objective Type Questions

1. What does UML stand for?
2. Which modeling language is used to visualize software systems?
3. UML is mostly used in which type of programming?
4. What type of diagram shows classes and relationships?
5. Which aspect of UML deals with system behavior over time?
6. What object-oriented concept allows one class to inherit from another?
7. What do we call a real instance of a class?
8. What UML concept hides internal details and shows only important features?
9. What term means grouping data and methods in a single unit?
10. What principle allows one operation to behave differently in different classes?
11. Which UML tool is free and open-source?
12. Which UML tool supports both desktop and cloud versions?
13. Which modeling tool works offline and online?
14. Who are the primary creators of UML?
15. Which relationship shows a "whole-part" where the part can exist independently?
16. Which UML relationship indicates inheritance?
17. Which element in UML is used to group other elements?

Answers to Objective Type Questions

1. Unified modeling language
2. UML
3. Object-oriented
4. Class
5. Behavioral
6. Inheritance
7. Object
8. Abstraction
9. Encapsulation
10. Polymorphism
11. StarUML
12. VisualParadigm
13. Diagrams.net
14. Grady Booch, Ivar Jacobson, James Rumbaugh
15. Aggregation
16. Generalization
17. Package

Assignments

1. Discuss the significance of UML in object-oriented programming.
2. Discuss the steps involved in creating UML diagrams.
3. What are the key advantages and disadvantages of using UML?
4. Describe the three main aspects of UML. How do these aspects help in creating a complete system design?
5. Explain the three main building blocks of UML with examples.

Suggested Reading

1. Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.
2. Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley.
3. Ambler, S. W. (2005). *The Object Primer: Agile Model-Driven Development with UML 2.0* (3rd ed.). Cambridge University Press.
4. Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The Unified Modeling Language Reference Manual* (2nd ed.). Addison-Wesley.

Reference

1. Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.
2. Cattell, R. (2001). *UML for Java Programmers*. Addison-Wesley.
3. Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley.
4. Larman, C. (2004). *UML and Object-Oriented Design Foundations* (2nd ed.). Prentice Hall.
5. McDonald, J., & Viswanathan, S. (2005). *UML 2.0 in a Nutshell*. O'Reilly Media.
6. Arlow, J., & Neustadt, I. (2005). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley.



Static and dynamic diagrams

Learning Outcomes

The learner will be able to :

- ◆ identify the difference between static and dynamic UML diagrams.
- ◆ understand the symbols used in different UML diagrams.
- ◆ familiarize how different UML diagrams help in designing and showing how a software system works.
- ◆ recognize the primary purpose of the key static UML diagrams (Class, Component, Deployment) in modeling a system's architecture.
- ◆ recognize the primary purpose of the key dynamic UML diagrams (Sequence, Activity, State Machine) in modeling a system's functionality and interactions.

Prerequisites

UML diagrams offer significant real-world benefits in software development. For instance, when designing an e-commerce website, a Use Case Diagram can visually represent all the ways a customer can interact with the site (browsing, adding to cart, checkout),. The learning process should begin by grasping the fundamental distinction between static diagrams, which model the system's structure. The dynamic diagrams, which illustrate its behavior over time. Subsequently, each diagram type should be studied individually, focusing on its specific purpose, the notation used to represent elements and their relationships. Exploring UML modeling tools can also enhance practical application. Studying numerous examples of each diagram type will also be highly beneficial for real-world application. The benefits of mastering UML diagrams in real-time software development are significant, including improved communication among stakeholders through a common visual language. Ultimately, understanding UML empowers developers and stakeholders to collaborate more effectively and build robust, well-documented software systems. It also provides a powerful visual

language that enhances communication, improves design, facilitates documentation, and ultimately leads to the development of more robust and understandable software systems.

Keywords

Structural Aspects, Behavioral Aspects, Classes, Attributes, Objects (Instances), Nodes (Hardware), Artifacts (Software), States, Events, Transitions, Use Cases, Actors.

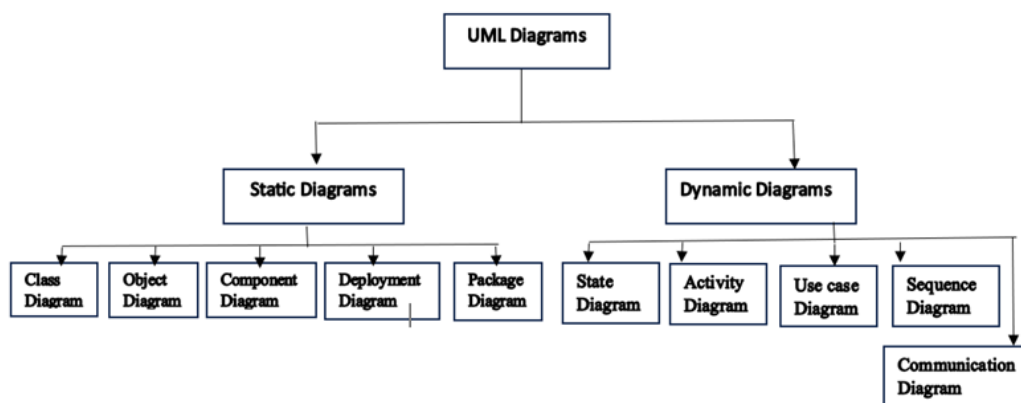
Discussion

UML diagrams are the output of the Unified Modeling Language. It is a pictorial representation of classes, objects, and relationships between them. It is used to define the functionality or the design of a system. UML diagrams are broadly classified into two categories: static and dynamic diagrams.

- ◆ Static Diagrams – represent the structural aspects of the system.
- ◆ Dynamic Diagrams – represent the behavioral aspects of the system.

5.2.1 Types of UML diagrams

UML diagrams are mainly classified as:



5.2.1.1 Static Diagrams

Static diagrams in UML depict the structural elements of a software system, showing what components exist and how they are interconnected. These diagrams do not depict behavior or time-based interactions but specify the organization and hierarchy of system components. It helps to understand the overall system architecture, planning object-oriented software and defining how components are arranged and interact statically. Static UML diagrams include the following types:

1. Class Diagrams

Class diagrams are one of the most important types of diagrams in UML. It is widely used in software engineering, particularly in the design of object-oriented systems. These diagrams provide a static view of the system by modeling its classes, attributes, operations (methods), and the relationships between different classes. In a class diagram, each class is represented as a rectangular box divided into three sections:

- ◆ The top section displays the class name.
- ◆ The middle section lists the attributes or properties that define the class.
- ◆ The bottom section contains the methods or operations that the class can perform.

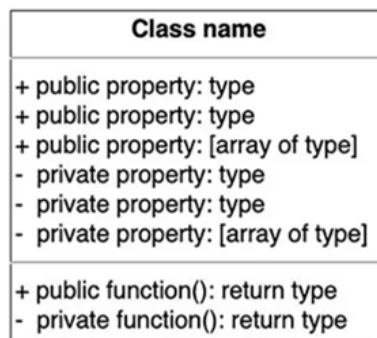


Fig:5.2.1 Class diagram symbol

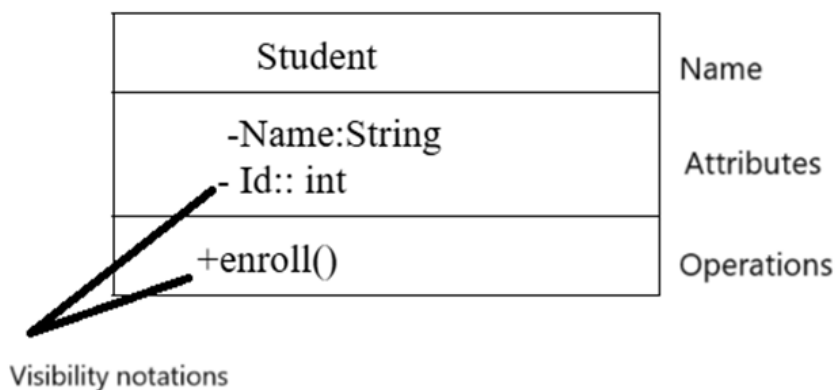


Fig: 5:2:2 Class diagram example

Example: School System with Aggregation between the School and Student classes.

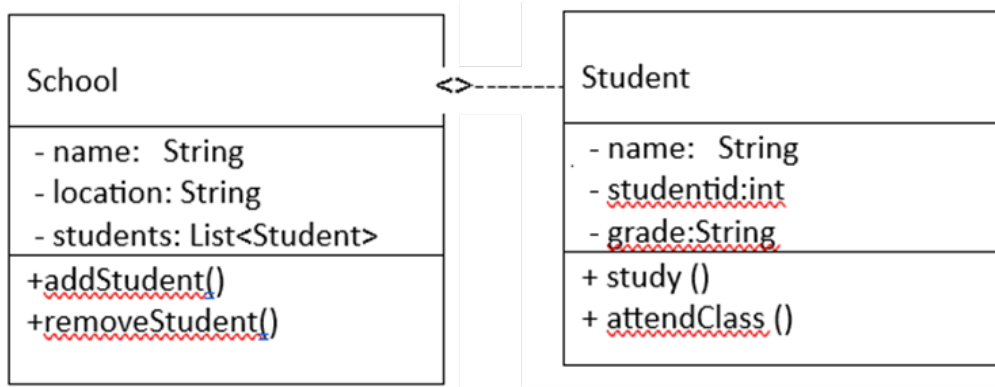


Fig: 5:2:3 Class diagram with relationship

1. **Students:** A list of Student objects that belong to the school. A School can have many Students (aggregation). If the School was removed, the Students would still exist independently. That means the students are not destroyed when the school is removed.

In this case, the relationship signifies that a School is composed of many Students, but the Student can exist outside the scope of the School.

2. Object Diagram

Object diagram focuses on the actual instances of classes at a specific point in time. The class diagrams illustrate the structure and relationships of classes in a general way. The object diagrams depict specific objects (instances of classes) and the concrete relationships between them during a particular moment in the system's execution. It provides a snapshot of how the system behaves in real time by showing how various instances are linked and interact, based on the structure defined in the class diagram.

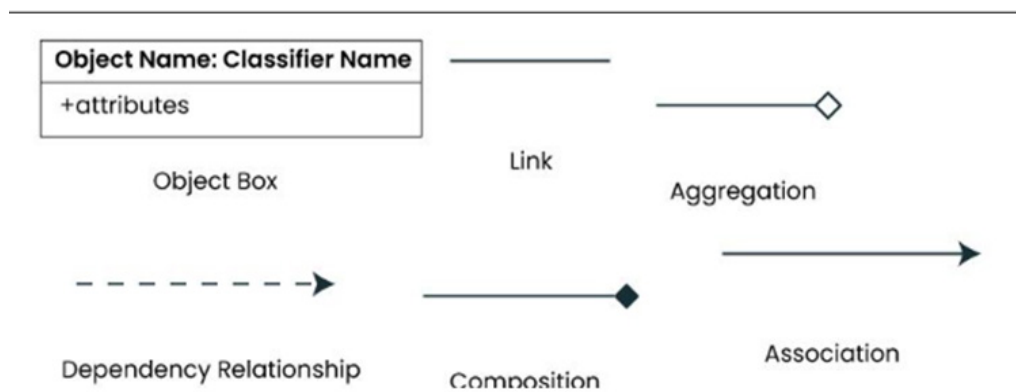


Fig: 5.2.4 Object diagram symbol

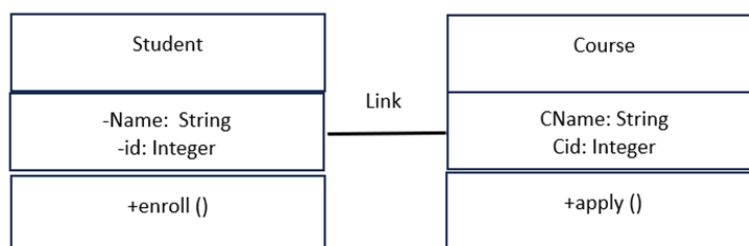


Fig: 5:2:5 Object diagram example (An object of class student is linked to an object of class course)

3. Component diagrams

A component diagram illustrates the physical components of a system and how they interact with each other. It is used to model the high-level structure of a system by showing its software components such as classes, interfaces, packages, or modules and the dependencies between them. Components can be software classes, collections of classes, or subsystems. The component diagram symbols are:

Component symbol: A rectangle with the «component» keyword and a small folded rectangle in the top right corner.

Interfaces: A circle (lollipop) symbol for provided interfaces, and a half-circle (socket) for required interfaces, if needed. Interfaces describe what a component can do or what it needs from others. Circles are for provided interfaces and half-circles for required interfaces.

Ports: Ports are specific points on a component's boundary where interactions take place. They define where interfaces are either provided by the component or required from other components.

Dependency arrows: Dotted arrows with open arrowheads indicating usage or communication between components.

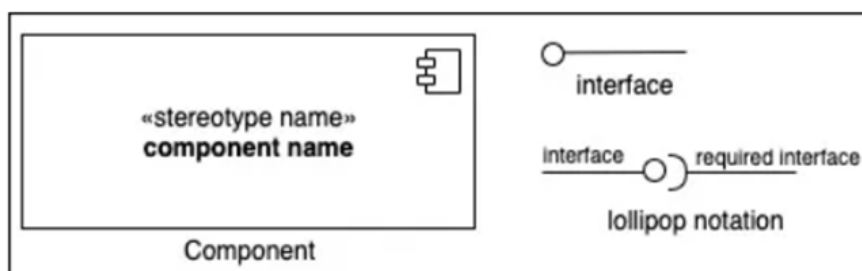


Fig: 5:2:6 Component diagram symbols

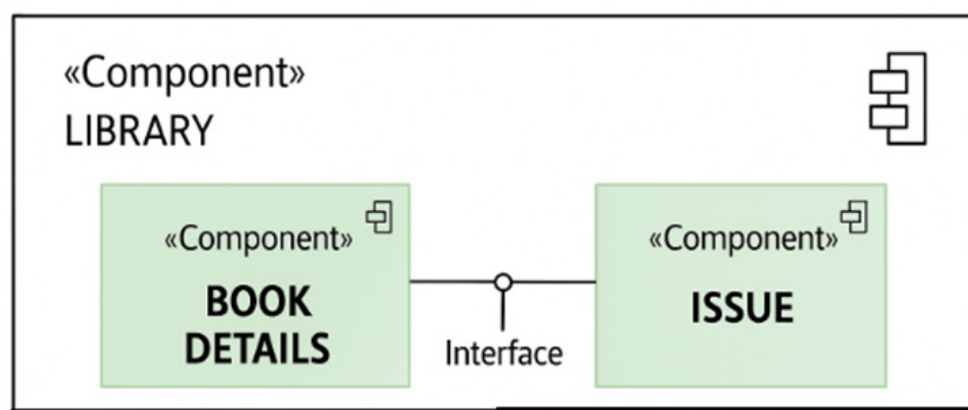


Fig: 5:2:7 Component diagram example

4. Deployment Diagrams

A Deployment Diagram provides a detailed representation of the physical architecture of a system. It highlights the relationship between the software and the hardware, showing how the software is physically deployed and how different components communicate with one another. This diagram is particularly useful for understanding the operational environment of a system, as it depicts the mapping of software artifacts (such as components, executables, libraries) to physical nodes (servers, workstations, cloud instances, etc.). Key elements of a deployment diagram include:

Nodes: Physical hardware devices where software components are deployed, like servers, workstations, or routers.



Fig: 5:2:8 Node

Components: A rectangle with two tabs that indicate a software part is called a component.



Fig: 5:2:9 Component

Artifacts: Physical files placed on nodes that implement software components, like executables or scripts.



Fig: 5:2:10 Artifacts

Dependencies: A dashed line that ends in an arrow denotes the dependency of one node or component on another.



Fig: 5.2.11 Dependency

Associations: A message or other sort of communication between nodes is indicated by a line.

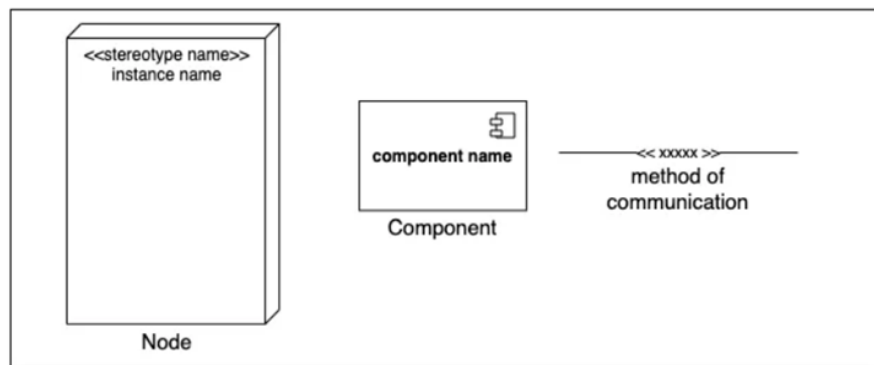


Fig: 5:2:12 Deployment symbols

Steps to Create a Deployment Diagram

- ◆ List all software and hardware parts. (e.g., servers, databases, web apps)
- ◆ Find out how they are connected. (e.g., how data moves between them)
- ◆ Collect extra details. (e.g., network information, special rules)
- ◆ Draw the hardware (nodes) = Use cube symbols for devices like servers.
- ◆ Add the software (artifacts).- Put them inside the correct hardware.
- ◆ Connect them with lines or arrows. - Show how the parts communicate.
- ◆ Label everything clearly. - Add names, protocols, or specs if needed.
- ◆ Write down important notes. - Include any decisions or special setups.

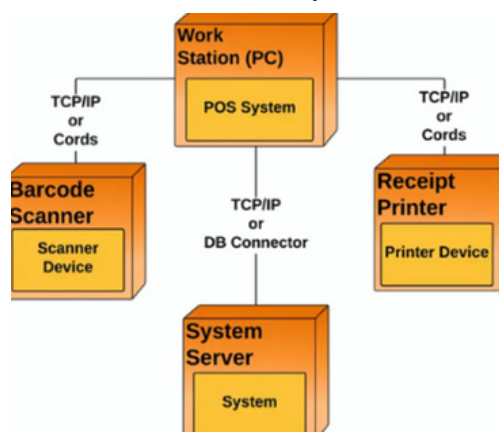


Fig: 5:2:13 Deployment diagram illustrates the physical components and their connections within a Point of Sale (POS) system,

This deployment diagram depicts a common setup for a POS system:

1. A Work Station (PC) is the central hub where the cashier interacts with the POS software.
2. A Barcode Scanner is used to quickly input product information into the POS system on the workstation.
3. A Receipt Printer is used to provide customers with a record of their purchase, controlled by the POS system.
4. A System Server acts as the backend, storing the central database and handling more complex operations for the entire POS system.
5. Peripheral devices like the barcode scanner and receipt printer can connect to the workstation either through network cables (TCP/IP) or physical cords. The POS software on the workstation can communicate with the server over a network (TCP/IP) or directly access its database.

6. Package Diagrams

A package diagram in UML (Unified Modeling Language) is a structural diagram used to organize elements of a system into logical groups called packages. Packages are like folders that contain related classes, components, or other UML elements. This diagram is especially helpful in organizing large systems by grouping similar functionalities.

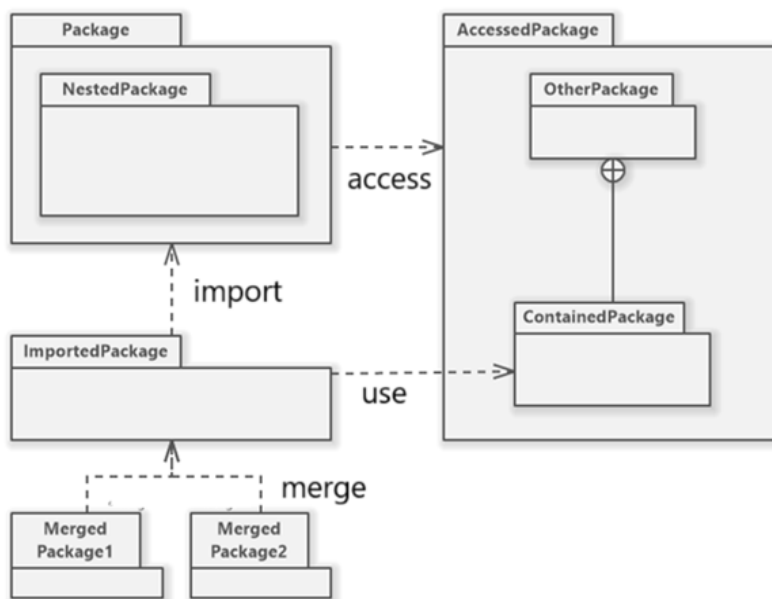


Fig: 5:2:14 Package diagram symbols

Key Elements of a UML Package Diagram:

1. Packages: Represented as a folder icon.
2. Dependencies: Shown with dashed arrows indicating relationships like:
`<<use>>`: One package uses elements from another.

- <<import>>: One package makes the elements of another package directly accessible.
- <<access>>: One package accesses elements (but not necessarily imports them).
- <<merge>>: Combines two packages into one, inheriting elements.

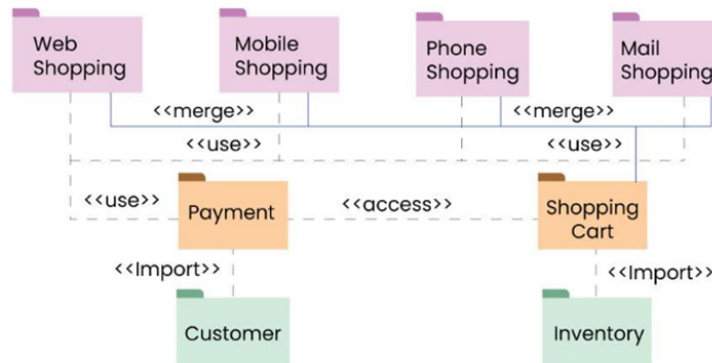


Fig: 5:2:15 Package diagram example

This diagram shows how a shopping system works behind the scenes, whether someone shops through a website, a mobile app, by phone, or by mail. All these shopping methods are connected and work together as part of a single system. No matter which way a customer shops, the system always uses the same important parts. First, there is a shopping cart, where items are added before buying. Then there is a payment system that handles the money part of the purchase. The shopping cart checks the inventory to see if the items are available, and the payment system uses the customer's details to complete the transaction. This shared setup helps keep everything organized, so all shopping methods use the same tools to make the process smooth and efficient for both the customer and the business.

5.2.1.2 Dynamic Diagrams

Dynamic UML diagrams describe the behavior of systems. A dynamic diagram describes the operations, actions, and changes that occur in a system over time.

1. State machine diagram

A state machine diagram describes the characteristics of a system as it changes in response to an event. As a system changes, it moves from one state to another.

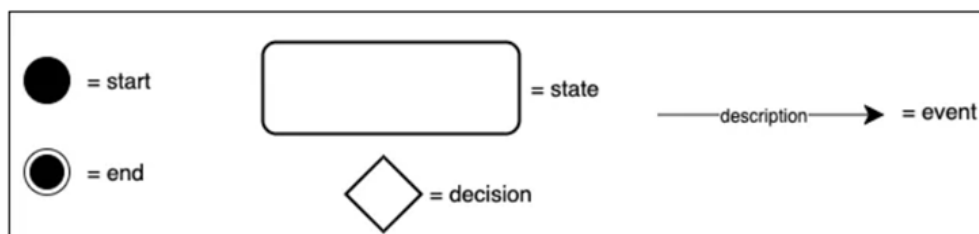


Fig:5:2:16 State machine symbols

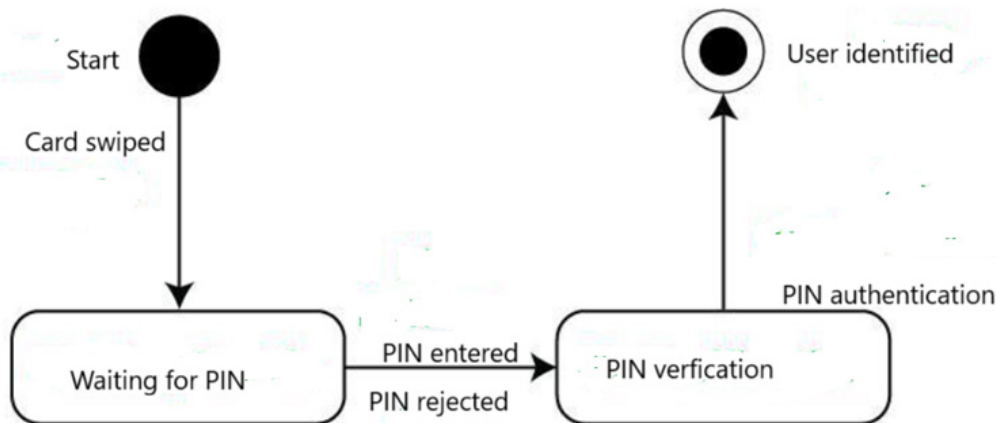


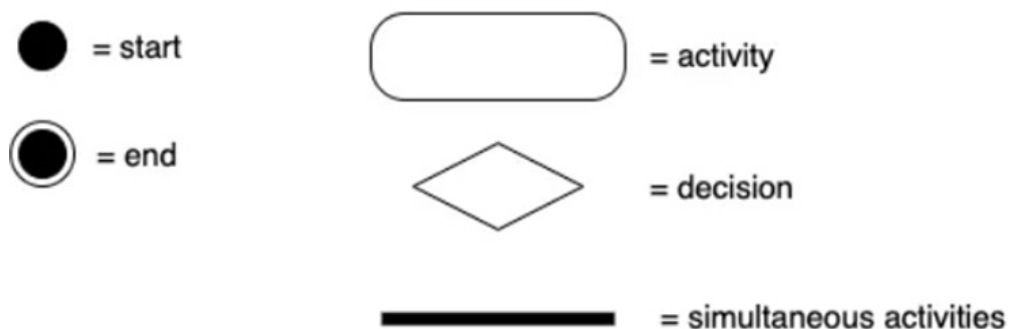
Fig: 5.2.17 An example of a state machine diagram for user verification

5.2.16 An example of a state machine diagram for user verification.

This state machine diagram illustrates a slightly more complete process of user identification, likely in an ATM or similar system. The user identification process begins at the Start state. Upon the event of a card being swiped, the system transitions to the Waiting for PIN state, prompting the user for input. Once the user enters their PIN, the system proceeds to the PIN verification state. If the entered PIN is rejected during this verification, indicating an incorrect PIN, the system reverts back to the Waiting for PIN state, allowing the user to attempt PIN entry again. Conversely, if the PIN is successfully authenticated, the system moves from the PIN verification state to the User identified final state, signifying successful user recognition.

2. Activity Diagrams

Activity diagrams serve as visual tools to model both sequential and concurrent activities, essentially depicting workflows within a system or process. It also describes the flow of actions in a system or subsystem.



5.2.18 Activity diagram symbols

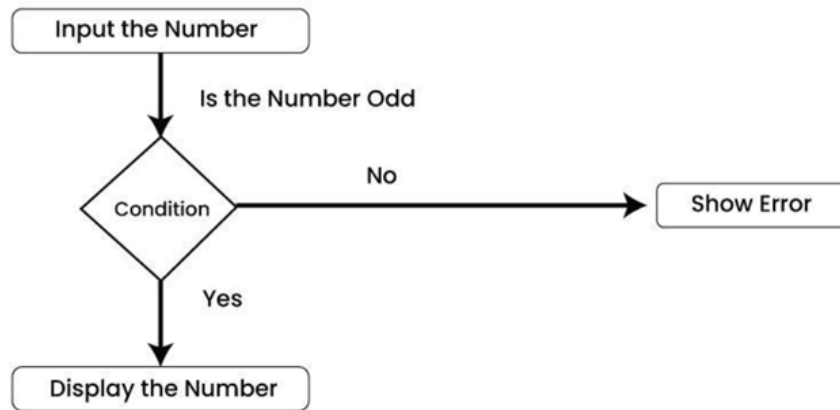
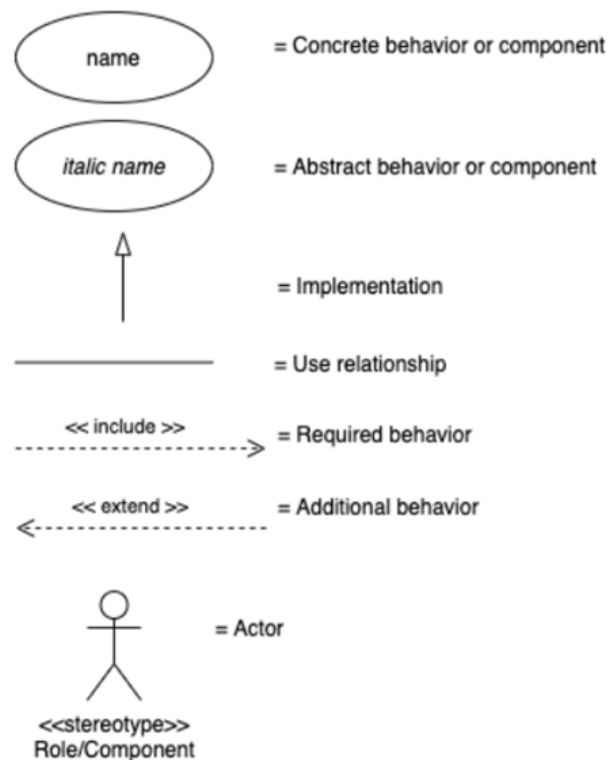


Fig: 5.2.19 A activity diagram example using decision node

The activity diagram illustrates a process that starts by receiving a number as input. A conditional check then determines if this number is odd. If the number is odd, the subsequent action to display it. However, if the number is even, the process branches to display an error message. This straightforward example effectively demonstrates the use of a decision node within an activity diagram to represent conditional behavior.

3. Use Case Diagrams

A use case diagram describes various behaviors in a system and who or what controls that behavior.



5.2.20 Use case symbols

The main function of a use case diagram is to depict the different ways actors interact with the system to achieve specific goals. It also provides a clear and concise visual summary of how users will interact with the system to fulfill their needs.

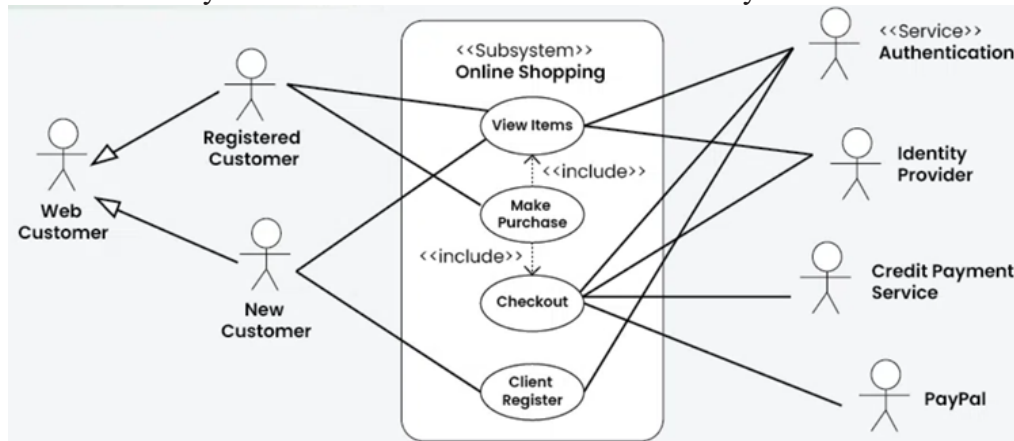


Fig: 5.2.21 Use case diagram example Online shopping

This use case diagram illustrates the interactions within the "Online Shopping" subsystem. It shows that Web Customers, categorized as Registered Customers and New Customers, can View Items. Registered Customers can further Make Purchases (which includes viewing items) and proceed to Checkout (which includes making a purchase). New Customers have the option to Client Register to become registered customers. The diagram also highlights the involvement of external services: Authentication and Identity Provider are used for user verification, while Credit Payment Service and PayPal are utilized for processing payments during the Checkout process.

4. Sequence diagram

Sequence diagrams are visual blueprints that clearly illustrate how different objects within a system collaborate with each other to achieve a specific task. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Sequence symbols

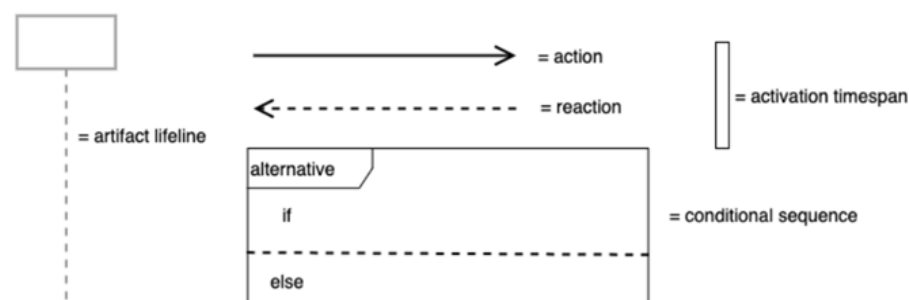


Fig: 5.2.22 Sequence diagram symbols

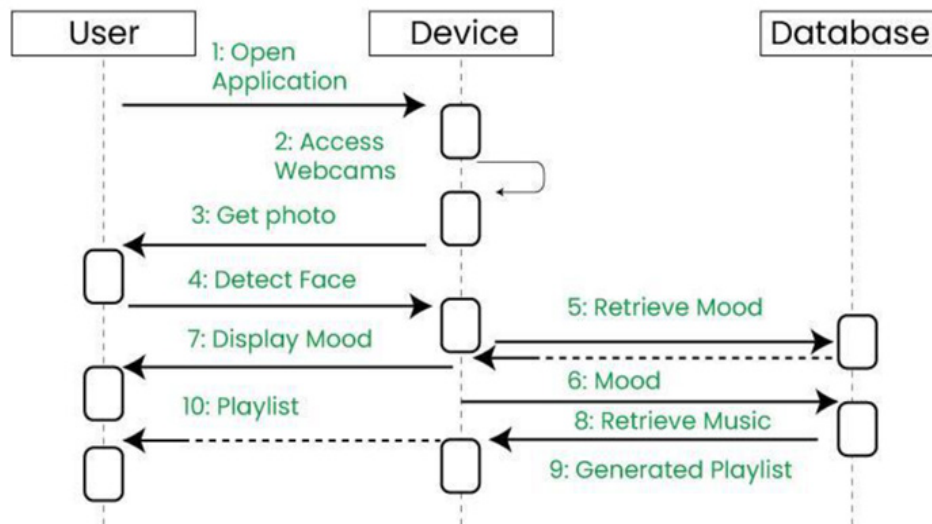


Fig: 5.2.23 Example of sequence diagram Mood-Based Music Playlist Generation

The sequence diagram titled "Mood-Based Music Playlist Generation" depicts the interaction flow between a User, a Device, and a Database to create a mood-based music playlist. The process begins with the User opening an application on the Device, which then accesses the device's webcams to capture a photo of the User and subsequently detects their face. The Device then queries the Database to retrieve the mood associated with the detected facial expression, receives the mood information, and displays it to the User. Following this, the Device requests the Database to retrieve a music playlist corresponding to the identified mood, receives the generated playlist, and finally presents it to the User.

5. Communication Diagram

The communication diagram describes how messages get exchanged between components in a system. It also describes how one component is associated with another component. Messages are described using lines with arrowheads. An association between two components is described using a line without any arrowheads.

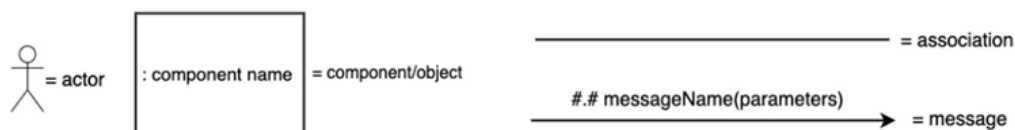


Fig: 5.2.24 Communication symbols

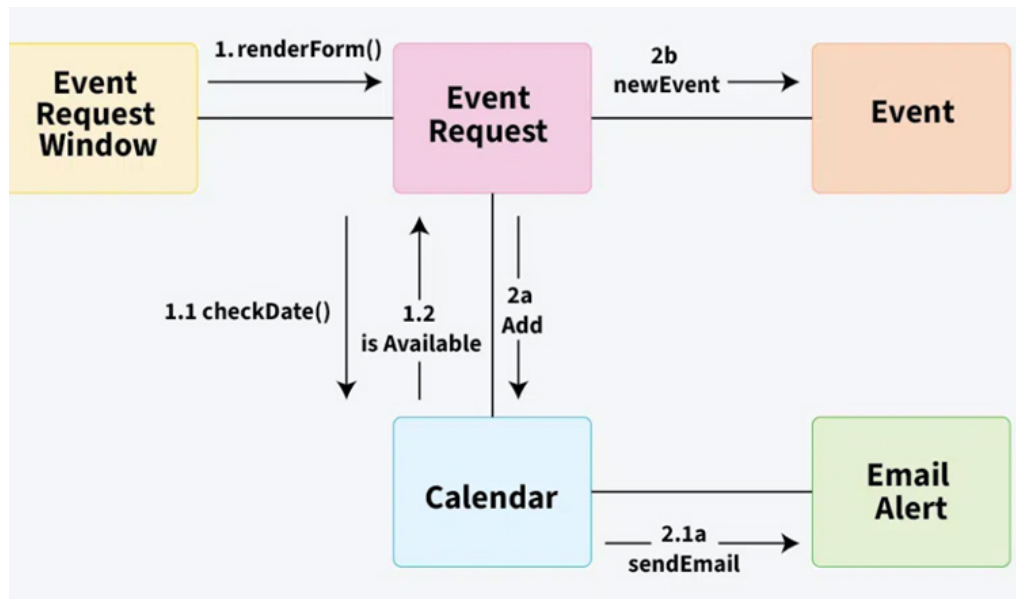


Fig: 5.2.25 Communication diagram example

5.2.2 Compare static and dynamic diagrams

Table 5.2.1 Comparison of static and dynamic diagrams

Aspect	Static diagrams	Dynamic diagrams
Purpose	Show the structure of a system.	Show the behavior of a system over time.
Time	Do not represent time or change	Represent sequences of operations or events over time
Focus	Objects, classes, relationships, architecture	Interactions, state changes, workflows.
Main Elements	Classes, attributes, methods, objects, packages	Messages, states, transitions, control flow
System Aspect	"What the system is" – its components and static relationships	"What the system does" – how it behaves and reacts
Examples	Class diagrams, Object diagrams, component diagrams	Sequence diagrams, state machine diagrams, activity diagrams

Recap

- ◆ UML diagrams are visual tools used to design and understand software systems.
- ◆ They show how different parts of a system work and interact.
- ◆ UML diagrams are divided into two types: static and dynamic.
- ◆ Static diagrams show the structure of the system.
- ◆ Dynamic diagrams show how the system behaves over time.
- ◆ Static diagrams include class, object, component, deployment, and package diagrams.
- ◆ Class diagrams show classes, their attributes, methods, and relationships.
- ◆ Each class box has three parts: name, attributes, and methods.
- ◆ Object diagrams show specific examples (instances) of classes at a certain moment.
- ◆ Object diagrams show specific examples (instances) of classes at a certain moment.
- ◆ Object diagrams give a real-time snapshot of the system's behavior.
- ◆ Component diagrams show the software pieces (like modules) and how they depend on each other.
- ◆ Components may include classes, interfaces, or subsystems.
- ◆ Deployment diagrams show where software runs, like on servers or devices.
- ◆ They include nodes (hardware), components (software), and artifacts (files).
- ◆ Deployment diagrams help understand the physical setup of a system.
- ◆ Package diagrams organize system elements into folders called packages.
- ◆ They help manage large systems by grouping related items together.
- ◆ State machine diagrams show how a system changes from one state to another.
- ◆ Activity diagrams show workflows and how tasks are performed step by step.
- ◆ These diagrams can show both sequential and parallel actions.
- ◆ Use case diagrams show how users interact with the system to achieve goals.

- ◆ Sequence diagrams show how different parts of the system communicate in order.
- ◆ Communication diagrams show how components are connected and exchange messages.

Objective Type Questions

1. Which type of UML diagram shows system structure?
2. Which type of UML diagram shows system behavior?
3. What type of diagram represents classes and their relationships?
4. Which diagram shows object instances at runtime?
5. Which diagram shows software components and their dependencies?
6. Which diagram displays hardware and software deployment?
7. Which diagram organizes elements into logical groups?
8. Which diagram models user interactions with a system?
9. What diagram shows changes in system states?
10. What diagram models control flow or workflows?
11. Which diagram shows object interaction over time?
12. Which diagram emphasizes communication between objects?
13. What keyword is used to represent a package in UML?
14. Which diagram is commonly used in object-oriented design?
15. What diagram type uses nodes and artifacts?
16. Which element in a class diagram holds methods?
17. Which element in a class diagram holds variables?
18. What symbol represents a class in UML?
19. What symbol represents a package in UML?
20. What line type shows dependency?
21. What keyword represents a reusable software part?
22. What shape is used for a decision in activity diagrams?
23. What does a lollipop symbol represent?
24. Which diagram is best for visualizing real-time object links?
25. Which diagram is used to plan modules or systems logically?

Answers to Objective Type Questions

1. Static
2. Dynamic
3. Class
4. Object
5. Component
6. Deployment
7. Package
8. Use case
9. State
10. Activity
11. Sequence
12. Communication
13. package
14. Class
15. Deployment
16. Operation
17. Attribute
18. Rectangle
19. Folder
20. Dashed
21. Component
22. Diamond
23. Interface
24. Object
25. Package

Assignments

1. Explain the difference between static and dynamic UML diagrams with suitable examples.
2. Draw a UML class diagram for an online library management system and explain its components.
3. Explain the symbols used in component diagrams and draw a sample diagram.
4. Draw and explain an activity diagram for an online food ordering process.
5. Differentiate between class diagrams and object diagrams using suitable examples.

Suggested Reading

1. Larman, C. (2015). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Pearson Education.
2. Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The unified modeling language user guide* (2nd ed.). Addison-Wesley Professional.
3. Fowler, M. (2004). *UML distilled: A brief guide to the standard object modeling language* (3rd ed.). Addison-Wesley Professional.
4. Alhir, S. S. (2003). *Learning UML*. O'Reilly Media.

Reference

1. Larman, C. (2015). *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development* (3rd ed.). Pearson Education.
2. Stevens, P., & Pooley, R. (2006). *Using UML: Software engineering with objects and components* (Updated ed.). Addison-Wesley.
3. Alhir, S. S. (2003). *Learning UML*. O'Reilly Media.



Design Walkthroughs

Learning Outcomes

After completion of this unit, the learner will be able to:

- ♦ define the term design walkthrough and its role in the software development process.
- ♦ list the key steps involved in conducting a critical design review.
- ♦ recall the different types of consistency checks used in software design.
- ♦ identify the roles and responsibilities of participants in a design walkthrough session.
- ♦ state the purpose and benefits of using consistency checkers in software engineering.

Prerequisites

To effectively understand the topic "Design Walkthroughs, Critical Design Review, and Consistency Checkers," learners must first be familiar with the Software Development Life Cycle (SDLC) and the key activities involved in the design phase. These techniques are used to validate and verify design decisions before moving into implementation. For example, in a logistics management system, a design walkthrough might be conducted to ensure that the module handling shipment tracking aligns with the user requirements and architectural constraints defined earlier. Without this foundational knowledge, it becomes difficult to appreciate the timing and purpose of design validation activities within a software project.

Additionally, a solid grasp of software design artifacts such as UML diagrams, data flow diagrams, and interface specifications is crucial. Design walkthroughs and critical reviews often focus on examining these artifacts to identify issues early.



For example, in an online ticket booking system, a consistency checker might highlight a mismatch between the sequence diagram and the class diagram for seat reservation. Understanding the role of different team members like developers, testers, and business analysts also helps, as these reviews are collaborative processes aimed at improving design quality and ensuring consistency across documentation and implementation.

Keywords

Process, Roles, Critical Design Review, Tools, Consistency Checker, Manual, Automated.

Discussion

5.3.1 Introduction to Software Design Verification

Software design verification is a crucial step in the software development life cycle, ensuring that the design correctly implements the requirements and lays a strong foundation for successful implementation. Verification activities aim to detect errors early, improve the quality of the final product, and reduce costly rework later in the development process. Unlike testing, which verifies the finished product, design verification focuses on reviewing and analyzing design artifacts like system models, diagrams, and documents to confirm their accuracy, completeness, and consistency before any code is written.

Among the key techniques used in design verification are design walkthroughs, critical design reviews, and consistency checkers. A design walkthrough is an informal group activity where the author of the design presents it to a peer group to gather feedback and identify defects. A critical design review is a more formal and structured evaluation, typically conducted at major project milestones, with the goal of determining whether the design is mature enough to proceed to development. Both walkthroughs and reviews

encourage collaboration among team members, including designers, developers, testers, and stakeholders, to enhance the overall quality and robustness of the software.

In addition to manual reviews, consistency checkers play an important role by automatically analyzing design documents to identify inconsistencies, omissions, or contradictions. These tools help ensure that all parts of the design are aligned, minimizing the risk of integration problems during development. Together, walkthroughs, critical reviews, and consistency checkers form a comprehensive approach to validating software design, ultimately leading to more reliable, maintainable, and high-quality software systems.

5.3.2 The Concept of Design Walkthroughs

Design walkthroughs are a structured, yet informal technique used in software engineering to review and validate a system's design before development proceeds. In a walkthrough, the designer presents the design documents such as system models, architecture diagrams, or interface specifications to a group of peers, including developers, testers, and project stakeholders. The primary goal is to uncover errors, inconsistencies, or areas of improvement early in the development cycle. Unlike formal inspections, walkthroughs are more flexible and focus on understanding the design rather than strictly evaluating it for approval.

By promoting collaborative review, design walkthroughs help ensure that the software design meets the specified requirements, follows best practices, and is understandable to all team members. For example, in designing a mobile banking application, a walkthrough might highlight missing security features in the transaction flow. Early feedback gathered through walkthroughs not only improves design quality but also reduces the likelihood of costly changes during later stages of development, making it a vital part of effective software engineering practices.

5.3.2.1 Definition and Purpose

A design walkthrough is an informal peer review process in software engineering where the author of a design presents their work to a group of colleagues to examine the logic, accuracy, and completeness of the design. The primary focus is on understanding the design, identifying potential errors, inconsistencies, or gaps, and gathering feedback to improve it before proceeding to the implementation phase. Walkthroughs are typically less formal than inspections and emphasize collaborative learning and problem-solving rather than strict evaluation or approval.

The main purpose of a design walkthrough is to detect design flaws early and improve the quality of the software product. It helps ensure that the design meets the specified requirements, is technically feasible, and aligns with project goals. Walkthroughs encourage open discussion, knowledge sharing, and early validation, allowing teams to uncover misunderstandings or hidden issues before coding begins. By doing so, they reduce the risk of expensive rework later in the project and promote a deeper shared understanding among team members about the system being built.

5.3.2.2 Walkthrough Process and Roles

The walkthrough process is an informal peer review technique used in software engineering to examine and validate design or requirement documents. It involves the author of the document presenting their work to a group of peers including developers, testers, and stakeholders to gain feedback, detect errors, and improve the overall quality of the design. Unlike formal inspections, walkthroughs emphasize understanding the content, clarifying doubts, and encouraging collaborative discussion. This process helps identify inconsistencies, missing information, or logical flaws early in the development cycle, reducing rework and ensuring the design aligns with project goals and requirements. A design walkthrough process (Fig: 5.3.1) typically follows these structured steps:

1. **Preparation Stage:** The walkthrough begins with the preparation stage, where the walkthrough leader defines the objectives of the session and selects appropriate participants, including designers, developers, testers, and stakeholders. The relevant design documents, such as architecture diagrams, interface specifications, and flowcharts, are distributed in advance to give participants time to review the material.
2. **Presentation Stage:** During the presentation stage, the author of the design takes the lead by walking the participants through the design components in a logical and structured manner. The focus is on explaining the functionality, logic, and rationale behind each design element, while encouraging questions and clarifications from the participants to ensure shared understanding.
3. **Discussion and Evaluation Stage:** In this stage, the participants actively evaluate the design by identifying inconsistencies, omissions, ambiguities, and potential defects. Constructive criticism is encouraged, and the discussion is kept focused on improving the quality of the design rather than assigning blame or fault.
4. **Documentation of Findings:** Throughout the walkthrough, a recorder or designated participant is responsible for documenting the issues identified, along with any suggestions or recommendations for improvement. These records serve as an essential reference for post-walkthrough revisions.
5. **Follow-up and Revision:** After the walkthrough session, the author reviews the documented feedback and revises the design accordingly. If the changes are substantial or unresolved issues remain, a follow-up walkthrough may be scheduled to re-evaluate the modified design.

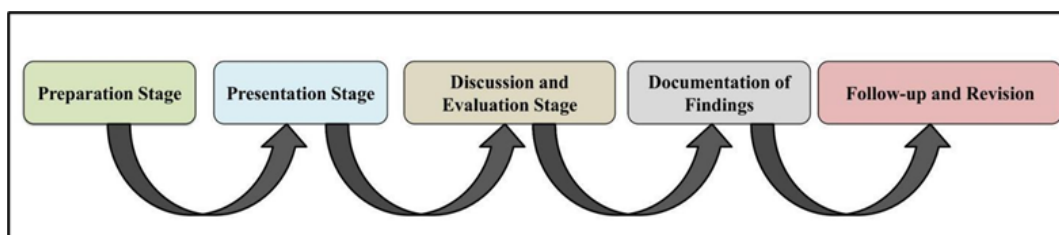


Fig 5.3.1: Design walkthrough process

The walkthrough roles are clearly defined responsibilities assigned to participants in a design walkthrough to ensure the process is effective, focused, and productive. Each role contributes uniquely to the session's objectives. Several key roles (Fig: 5.3.2) contribute to the effectiveness of a walkthrough:

1. **Author (Designer):** The author is responsible for creating the design and presenting it during the walkthrough session. They explain the design's logic, respond to questions, and accept constructive feedback. The author also incorporates necessary changes after the walkthrough based on the feedback provided.
2. **Walkthrough Leader (Coordinator):** The walkthrough leader organizes the session, defines its purpose, invites appropriate participants, and ensures that the session proceeds in a structured and time-bound manner. They moderate the discussion and keep it aligned with the session's goals.
3. **Reviewers (Participants):** Reviewers include developers, testers, analysts, and sometimes clients, who evaluate the design from different perspectives. They actively participate by identifying issues, suggesting improvements, and ensuring that the design meets technical, functional, and user requirements.
4. **Recorder (Scribe):** The recorder is responsible for capturing all key points discussed during the walkthrough, including identified defects, suggestions, and action items. Their notes ensure that feedback is preserved and followed up appropriately.

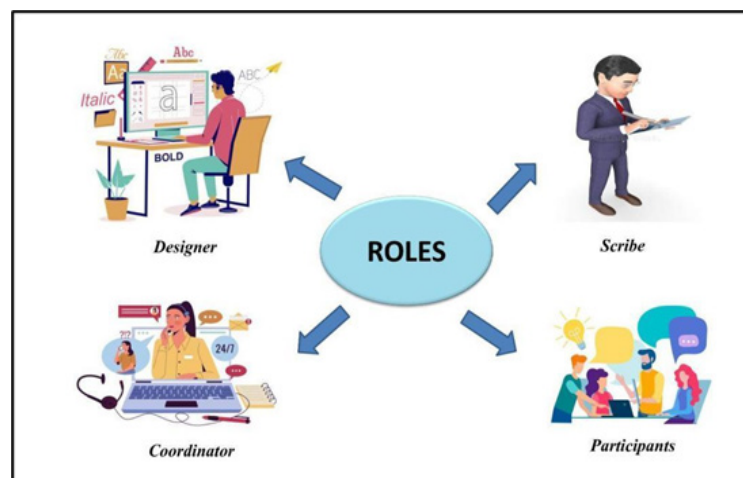


Fig 5.3.2: Roles in design walkthrough

5.3.2.3 Benefits and Limitations

Some of the benefits of design walkthrough include:

1. **Early Error Detection:** Helps identify design flaws, inconsistencies, and missing elements before development begins, reducing costly fixes later.
2. **Improved Design Quality:** Peer feedback leads to refinement of the design, ensuring better structure, logic, and alignment with requirements.

3. **Knowledge Sharing:** Team members gain a better understanding of the design, promoting collaboration and reducing knowledge complexity.
4. **Increased Team Communication:** Encourages open discussion among cross-functional team members, leading to better coordination and shared understanding.
5. **Reduced Development Time and Cost:** Catching issues early prevents rework during coding and testing phases, ultimately saving time and resources.
6. **Supports Training and Onboarding:** Junior team members benefit from walkthroughs by learning from experienced colleagues and understanding the project better.

Some of the limitations of design walkthrough include:

1. **Time-Consuming:** Organizing and conducting walkthroughs, especially for complex designs, can take considerable time.
2. **Dependent on Participant Engagement:** The effectiveness of a walkthrough relies heavily on the active participation and expertise of the reviewers.
3. **Lack of Formal Documentation:** Being an informal process, important decisions or discussions may not always be well-documented unless explicitly recorded.
4. **Risk of Groupthink:** Participants may hesitate to challenge the author's design, especially in hierarchical teams, leading to overlooked issues.
5. **Not a Substitute for Formal Review:** Walkthroughs are useful but may miss deeper technical or structural issues that a formal design review might catch.
6. **Effectiveness:** Walkthroughs are generally more effective for functional aspects of the software rather than non-functional requirements like performance, scalability, or security.

5.3.2.4 Example Scenarios

Design walkthroughs are versatile review activities that can be applied at various stages of software development. Below are detailed example scenarios illustrating how and when walkthroughs can be effectively used to ensure software quality, consistency, and clarity. Example scenarios are described below:

A.Reviewing the Design of a New System Module

Scenario: A software engineer has completed the preliminary design of a new payment processing module in an e-commerce platform. The module is expected to handle transactions, integrate with third-party payment gateways, and maintain transaction logs.

Purpose of Walkthrough: To validate the design's correctness, identify any missing functionalities, and ensure that the module integrates smoothly with the existing system.



Participants: The author (designer), senior developers, system architect, and QA engineers.

Expected Outcomes: Detection of potential design flaws (e.g., lack of failover for payment errors), suggestions for performance improvements, and alignment with security standards (e.g., encryption of sensitive data).

B. Reviewing a Complex User Interface (UI) Design

Scenario: The UI/UX team has developed a prototype and design documentation for a customer onboarding form in a banking application.

Purpose of Walkthrough: To evaluate the usability, accessibility, and consistency of the UI design with existing application interfaces.

Participants: UI/UX designer (author), frontend developers, accessibility expert, business analyst, and product owner.

Expected Outcomes: Improved form structure, better navigation flow, enhanced accessibility features (such as keyboard navigation and screen reader support), and more intuitive user experience.

5.3.3 Critical Design Reviews

A Critical Design Review (CDR) in software engineering is a formal evaluation process conducted to assess the technical quality, completeness, and feasibility of a software system's detailed design before development begins. It typically involves a panel of experts including software engineers, architects, and stakeholders who critically analyze design documents, architecture, interfaces, and performance considerations to ensure the solution meets all functional and non-functional requirements. CDRs help identify design weaknesses, integration risks, or gaps in requirement coverage early in the development lifecycle. As a key milestone in structured software projects, especially in safety- or mission-critical domains, a CDR ensures that the design is mature, well-documented, and ready for implementation with minimal risk.

5.3.3.1 Concept and Objectives

The Critical Design Review (CDR) is a formal evaluation process in software engineering used to assess the detailed design of a system before the development phase begins. It is conducted after the preliminary design has been completed and is intended to ensure that the system's architecture, components, interfaces, and data flows are fully developed and technically sound. The review typically involves a panel of experienced stakeholders including software engineers, architects, testers, and project managers who examine whether the design meets all functional and non-functional requirements, complies with applicable standards, and aligns with project goals.

CDRs are essential in large-scale or high-risk software projects, such as those in aerospace, defense, or healthcare, where design errors can have critical consequences. The process aims to uncover flaws, gaps, or technical risks in the design before any code is written, thus reducing the likelihood of costly changes later. A successful CDR demonstrates that the design is complete, feasible, and ready for implementation,

serving as a key decision point for proceeding with software development.

The objectives of Critical Design Reviews (CDRs) in software engineering, clearly outlined as:

1. **Ensure Design Completeness:** Verify that the detailed design fully addresses all system and user requirements, with traceability to the original specifications.
2. **Validate Technical Feasibility:** Assess whether the proposed design is technically sound, practical, and implementable using the chosen technologies and tools.
3. **Identify and Mitigate Risks:** Detect potential design flaws, performance issues, integration challenges, or security vulnerabilities before development begins.
4. **Confirm Standards Compliance:** Ensure the design adheres to organizational, industry, and regulatory standards (e.g., coding conventions, architectural guidelines, safety requirements).
5. **Promote Team and Stakeholder Alignment:** Facilitate communication and consensus among developers, architects, project managers, and clients to align expectations and clarify ambiguities.
6. **Support Project Decision-Making:** Provide a formal checkpoint for management to approve, revise, or halt the project based on the quality and readiness of the design.
7. **Improve Design Quality and Maintainability:** Encourage feedback from diverse technical perspectives to enhance the design's robustness, efficiency, scalability, and future maintainability.

5.3.3.2 Review Planning and Preparation

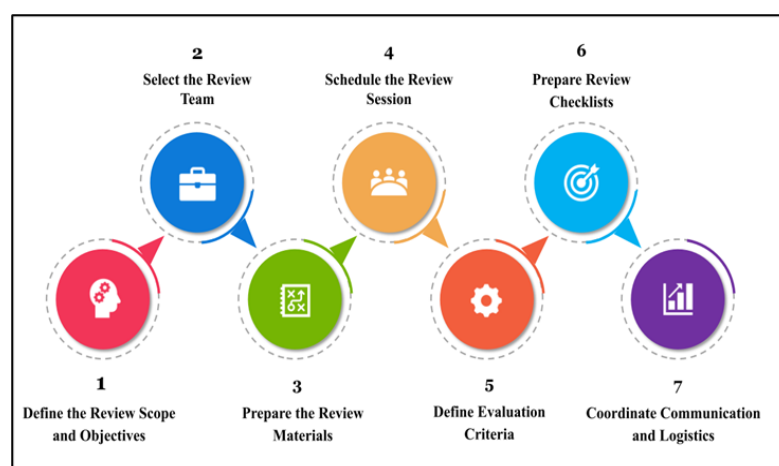


Fig 5.3.3: Planning phases of CDR

A successful Critical Design Review (CDR) begins with thorough planning and preparation to ensure that the review is effective, objective, and aligned with project

goals. This phase sets the foundation for a structured and productive evaluation process (Fig: 5.3.3).

1. **Define the Review Scope and Objectives:** Before initiating the CDR, the scope of the review must be clearly defined. This includes identifying the components, subsystems, or documents to be reviewed (e.g., architecture diagrams, interface specifications, database designs). Clear objectives—such as validating completeness, technical soundness, and risk mitigation—should be established and communicated to all participants.
2. **Select the Review Team:** An effective CDR requires a diverse and qualified review team. The team typically includes the design author, software engineers, system architects, QA leads, security specialists, and project stakeholders. Reviewers should have expertise relevant to the design under review and be independent enough to provide unbiased feedback.
3. **Prepare the Review Materials:** The design team must prepare and distribute comprehensive materials in advance of a review meeting. These may include:
 - ◆ Detailed design documents
 - ◆ System architecture diagrams
 - ◆ Interface control documents
 - ◆ Requirements traceability matrix
 - ◆ Risk assessment and mitigation plans
 - ◆ Relevant standards or guidelines
4. **Schedule the Review Session:** A structured schedule should be set for the review session. The agenda must allocate time for each component, allow for discussion, and include time to summarize findings and action items. The session should be moderated by a review leader or facilitator to maintain focus and ensure all viewpoints are heard.
5. **Define Evaluation Criteria:** Establish clear criteria for assessing the design, such as:
 - ◆ Completeness and correctness
 - ◆ Compliance with requirements and standards
 - ◆ Feasibility and maintainability
 - ◆ Risk identification and mitigation
6. **Prepare Review Checklists:** Reviewers should use structured checklists to ensure that no critical aspects are overlooked. Checklists may cover design principles, coding standards, security best practices, performance considerations, and interface compatibility.

7. **Coordinate Communication and Logistics:** Finally, ensure that logistics such as meeting invitations, access to documents, and collaboration tools (e.g., shared drives, conferencing platforms) are arranged well in advance. This avoids delays and supports efficient participation, especially in distributed teams.

5.3.3.3 Review Meeting Procedures

The Critical Design Review meeting is a structured and formal session aimed at thoroughly evaluating the detailed design of a software system. It follows a well-defined process to ensure the design is technically sound, requirement-compliant, and ready for implementation.

1. **Opening the Review:** The Review Chair or Moderator begins by welcoming attendees and confirming the agenda. The objectives of the CDR are clearly stated (e.g., verifying design completeness, identifying risks). All participants briefly introduce themselves and confirm their roles. Review logistics (duration, discussion format, tools) are reviewed.
2. **Design Presentation by the Author:** The design author or responsible engineer presents the key aspects of the detailed design. The presenter explains design choices, justifications, and how risks are addressed. This includes:
 - ◆ Architecture diagrams
 - ◆ Data flow and control flow
 - ◆ Interfaces and API specifications
 - ◆ Security and performance considerations
 - ◆ Traceability to requirements and constraints
3. **Reviewer Discussions and Q&A:** Reviewers engage in interactive discussion, raising questions, concerns, and suggestions. Discussions focus on:
 - ◆ Completeness and correctness of the design
 - ◆ Adherence to standards and best practices
 - ◆ Scalability, maintainability, and integration issues
 - ◆ Risk identification and mitigation strategies

The moderator ensures balanced participation, keeps the discussion on track, and prevents domination by any one participant.

4. **Documentation of Comments and Findings:** All comments, issues, and action items are recorded in a structured format (e.g., review log or issue tracker). Findings are categorized as:

- ◆ Major issues (must be fixed before implementation)
- ◆ Minor suggestions (optional improvements)
- ◆ Informational comments

5. Summary and Conclusion: The moderator summarizes the review outcomes, including:

- ◆ Major findings
- ◆ Agreed decisions
- ◆ Follow-up actions and responsibilities

A go/no-go decision is made:

- ◆ Go: Design is approved (possibly with minor revisions)
- ◆ Conditional Go: Design is approved pending specific changes
- ◆ No-Go: Major redesign is required before approval

6. Post-Review Actions: A formal review report is generated and shared with stakeholders. Assigned team members resolve open issues within agreed timelines. A follow-up review (if required) may be scheduled to verify critical corrections.

5.3.3.4 Comparison with Walkthroughs

CDRs are formal, high-stakes evaluations with decision authority, used primarily in critical systems or large-scale projects. Walkthroughs are informal reviews that focus on peer learning and early detection of issues, without formal approval outcomes. The comparison of CDR with design walkthroughs depicted as in Table 5.3.1.

Table 5.3.1: Comparison of CDR with Design walkthrough

Aspect	Critical Design Review (CDR)	Design Walkthrough
Formality	Highly formal and structured	Informal or semi-formal
Objective	Evaluate technical correctness, completeness, and feasibility	Understand the design and gather early feedback
Participants	Experts, stakeholders, architects, QA, and project managers	Peers, designers, and sometimes junior developers
Timing	After detailed design is completed, before implementation begins	Early in the design phase

Output	Formal review report, decision (Go/No-Go), action items	Notes, suggestions, and informal action points
Decision-making	Yes – review may result in approval or request for major changes	No – primarily used for improvement, not approval
Focus	Risk mitigation, requirement coverage, standards compliance	Design understanding, logic checking, and peer input
Leadership	Led by a review chair or moderator	Often led by the designer or team lead
Documentation Required	Detailed design documents and supporting artifacts	Design drafts, diagrams, and initial concepts

5.3.4 Consistency Checkers

Consistency is crucial in large and complex systems where multiple models, documents, codebases, or configurations interact. Inconsistencies can lead to system failures, bugs, miscommunication among teams, or delays in development. Consistency checkers in software engineering are tools or processes used to ensure that various software artifacts such as requirements, design models, code, and documentation remain aligned and logically coherent throughout the development lifecycle. As software systems grow in complexity and involve multiple teams or tools, inconsistencies can easily arise, leading to miscommunication, bugs, or even system failures. Consistency checkers help detect and resolve such discrepancies early, improving the reliability and quality of the final product.

These tools can operate on different levels of abstraction, including syntactic consistency (ensuring correct structure and formatting), semantic consistency (ensuring correct meaning and behavior across models), and version consistency (ensuring compatibility across evolving components). Common examples include UML model checkers, code static analysis tools, and schema validators. By integrating consistency checkers into development workflows, teams can maintain a more robust and maintainable software system, reduce errors, and streamline collaboration across various roles and phases of development.

5.3.4.1 Importance of Consistency Checking

Consistency checking is critically important in software engineering because it ensures that all parts of a software system work together harmoniously. As software projects often involve multiple components such as requirements documents, design models, source code, and test cases any mismatch or inconsistency between these can lead to defects, integration failures, or incorrect system behavior. Consistency checking helps detect such issues early in the development process, reducing the cost and effort of fixing problems later.

Moreover, consistency checking enhances collaboration among team members by maintaining a shared understanding of the system across different roles and tools. It supports traceability, improves maintainability, and is especially vital in safety-critical domains like healthcare, aviation, and automotive systems where regulatory compliance demands a high level of rigor. Overall, consistent software artifacts contribute to building reliable, high-quality, and easily evolvable software systems.

5.3.4.2 Types of Consistency Checkers

In software engineering, consistency checkers are essential tools that help maintain the coherence and correctness of software artifacts throughout the development process. As software projects often involve multiple models, documents, and components created by different teams, ensuring consistency between these elements is critical to prevent errors, misunderstandings, and integration failures. Different types of consistency checkers are designed to handle various aspects of consistency, ranging from syntax validation to semantic alignment.

Each type of consistency checker addresses a specific level or scope of potential inconsistency. For example, syntactic checkers focus on the correct structure and formatting of artifacts, while semantic checkers ensure that the meaning and behavior across models and code are logically aligned. Other types include intra-model and inter-model checkers, which verify internal consistency within a model or between different models, respectively. Additionally, traceability checkers maintain logical connections between requirements, design, and implementation, and version or configuration checkers help avoid discrepancies across versions and environments. Understanding these types helps teams choose the right tools to ensure software integrity and quality. Types of consistency checkers (Table 5.3.2) are summarized below:

1. **Syntactic Consistency Checkers:** These checkers ensure that software artifacts follow the correct structure and syntax rules. For example, a UML model checker may verify that all associations and elements conform to UML standards, or a code linter may ensure correct use of programming language syntax.
2. **Semantic Consistency Checkers:** These tools verify that the meaning and behavior of related artifacts are aligned. For instance, a semantic checker may confirm that the behavior described in a UML state diagram matches the logic implemented in the source code.
3. **Intra-model Consistency Checkers:** These tools verify consistency within a single model. For example, they might check whether all components in a class diagram are properly linked, or if all methods used are defined within the same UML model.
4. **Inter-model Consistency Checkers:** These focus on ensuring consistency between different models or views (e.g., checking that an element defined in a class diagram also exists in the sequence diagram).

5. **Traceability Checkers:** These ensure that there is a consistent and traceable relationship between different development artifacts, such as linking requirements to design elements and corresponding code modules.
6. **Version Consistency Checkers:** These tools help maintain consistency across different versions of software artifacts, ensuring that changes in one version are properly propagated to others and that all team members are working on compatible versions.
7. **Configuration Consistency Checkers:** These ensure that configuration files, dependencies, and environment settings are consistent across development, testing, and production environments.

Table 5.3.2: Types of Consistency Checkers

Type Of Consistency Checker	Description	Example Use Case
Syntactic Consistency	Ensures correct structure and format according to syntax rules.	Verifying proper UML notation or code formatting with a linter.
Semantic Consistency	Ensures that the meaning and behavior of related artifacts align.	Matching UML state diagrams with corresponding implementation logic.
Intra-model Consistency	Checks consistency within a single model or artifact.	Ensuring all classes in a class diagram have defined relationships.
Inter-model Consistency	Ensures consistency across different models or views.	Ensuring a class in the class diagram is also used in sequence diagrams.
Traceability Checkers	Ensures traceable links between artifacts like requirements, design, and code.	Linking user requirements to test cases and design components.
Version Consistency	Ensures compatible versions of components or documents are used.	Preventing mismatches between different versions of shared modules.
Configuration Consistency	Validates environment settings and configurations across systems.	Checking that database credentials match across staging and production.

5.3.4.3 Tools and Techniques for Consistency Checking

Consistency checking in software engineering is supported by a variety of tools and techniques designed to detect and manage inconsistencies across software artifacts such as models, code, requirements, and configurations. These tools (Table 5.3.3) automate the process of verifying that different components of a system remain aligned, helping teams maintain high-quality, reliable software.

Table 5.3.3: Common Tools for Consistency Checking

Tools	Purpose	Example Use Cases
SonarQube	Static code analysis	Checks code quality, syntax, and structural consistency across source files.
Eclipse Modeling Framework (EMF)	Model consistency checking	Verifies intra- and inter-model consistency in UML or domain-specific models.
Enterprise Architect	Model-driven development	Supports consistency checking across UML diagrams and other system artifacts
Alloy Analyzer	Formal specification and model checking	Analyzes logical consistency in software specifications.
Simulink Model Advisor (MATLAB)	Model verification and validation	Checks for structural and behavioral consistency in control system models.
ReqIF Studio	Requirements consistency	Verifies traceability and consistency of requirements documents.
Git / Version Control Tools	Version consistency	Helps manage version conflicts and ensure consistency across team codebases.

In software engineering, consistency checking techniques are essential for identifying and resolving mismatches across various software artifacts such as models, code, requirements, and configurations. As software systems evolve and involve multiple contributors, the likelihood of inconsistencies increases, potentially leading to system errors, misinterpretations, and failures. Consistency checking techniques provide

structured methods to detect such discrepancies early, ensuring all components of the system remain aligned and function as intended.

These techniques range from simple static analysis to more advanced formal verification methods. Common approaches include static analysis for examining code structure, model checking for validating logical behavior, and traceability analysis for ensuring proper linkage between requirements and implementation. Other techniques like schema validation, rule-based constraint checking, and integration with CI/CD (Continuous Integration and Continuous Delivery / Continuous Deployment) pipelines help automate and streamline consistency verification throughout the development process. By applying these techniques, software teams can enhance the reliability, quality, and maintainability of their systems. Each techniques (Table 5.3.4) are clearly explained below:

1. Static Analysis

Static analysis is a technique that examines software artifacts such as source code or models without executing them. It is commonly used to identify syntactic and structural inconsistencies, potential bugs, or violations of coding standards. By analyzing the internal structure of code, static analysis helps developers detect issues early in the development cycle, improving code quality and reducing debugging time. Tools like SonarQube, ESLint, and PMD automate this process and are often integrated into development environments or CI/CD pipelines.

2. Model Checking

Model checking is a formal verification technique used to systematically examine the logical behavior of system models against predefined properties or specifications. It explores all possible states of a system model to ensure that it behaves as expected under all scenarios. This technique is especially useful in safety-critical domains such as aerospace, automotive, and medical devices, where correctness is paramount. Tools like SPIN, NuSMV, and Alloy Analyzer are commonly used to perform model checking.

3. Traceability Analysis

Traceability analysis focuses on ensuring that there are clear, traceable links between software artifacts, such as requirements, design models, code, and test cases. This technique ensures that every requirement is properly implemented and tested, and it helps track the impact of changes across the system. It is particularly important in large and regulated projects, where accountability and compliance are required. Tools supporting traceability include ReqIF Studio and IBM Engineering Requirements Management (DOORS).

4. Schema and Configuration Validation

Schema and configuration validation ensures that data formats and configuration files conform to a defined structure or schema. It is widely used to maintain consistency in configuration files such as XML or JSON, especially across different environments like development, testing, and production. Validating these files prevents runtime errors and misconfigurations. Tools such as XML Schema Validator or JSON Schema Validator are used to automate this process.

5. Constraint-Based Checking

Constraint-based checking involves using formal rules or logical constraints to verify consistency within or between models. These rules are often written in languages like OCL (Object Constraint Language) and are applied to ensure that software models, such as UML diagrams, comply with defined structural or semantic rules. This technique is useful in model-driven development environments, where ensuring the correctness of abstract models is critical before generating code or proceeding to implementation.

6. Change Impact Analysis

Change impact analysis assesses how a change in one part of the system affects other components or artifacts. It helps identify all elements that may be impacted by a modification, ensuring that updates do not introduce inconsistencies elsewhere. This technique is crucial in complex and evolving systems where changes can ripple across multiple layers. It supports maintainability and helps prevent regression issues.

7. Automated Testing in CI/CD Pipelines

In CI/CD pipelines, consistency checking is integrated through automated tests that run whenever new code is committed or deployed. These tests can include unit tests, integration tests, and system tests, all aimed at ensuring that recent changes do not break existing functionality or introduce inconsistencies. By embedding these checks in the pipeline, development teams gain fast, continuous feedback, enabling quicker and safer delivery of software updates.

Table 5.3.4: Types of consistency checking techniques

Techniques	Description	Usage	Tools	Example
Static Analysis	Analyzes code or models without executing them to detect syntax errors, code smells, and violations.	Early error detection, code quality improvement	SonarQube, ESLint, PMD	Detecting unused variables or unreachable code in a Java project.
Model Checking	Uses formal methods to verify that a model satisfies specific logical or behavioral properties.	Ensuring correctness in safety-critical systems.	Alloy Analyzer, SPIN, NuSMV	Verifying that a model of a traffic light system never allows both lights to be green.

Traceability Analysis	Ensures that all requirements are properly linked to design, implementation, and test cases.	Requirement tracking, audit compliance, impact analysis	ReqIF Studio, IBM DOORS	Verifying that every user requirement has a corresponding test case
Schema & Config Validation	Validates structure of config/data files against predefined schemas (e.g., XML, JSON).	Preventing configuration errors in different environments	JSONLint, XML Validator, Yamllint	Ensuring config.yaml in production matches required structure.
Constraint-Based Checking	Applies formal rules (e.g., OCL) to verify internal or cross-model consistency.	Enforcing modeling standards in UML or domain-specific models.	Eclipse EMF, Enterprise Architect	Checking that all states in a UML diagram have at least one outgoing transition.
Change Impact Analysis	Assesses the effects of a change in one artifact on other related components.	Managing updates in large, interdependent systems.	Rational Software Architect	Identifying which test cases to rerun after a requirement change.
Automated Testing in CI/CD	Integrates testing into pipelines to continuously verify consistency during development.	Maintaining consistency after every code change or deployment	Jenkins, GitHub Actions, GitLab CI	Running unit and integration tests on every code commit.

5.3.4.4 Automated vs Manual Consistency Checks

In software engineering, consistency checks are crucial to ensuring that different components of a system such as requirements, code, models, and configurations are aligned and function correctly together. These checks can be performed either manually by human reviewers or automatically using software tools. The choice between automated and manual consistency checking depends on factors such as the complexity of the system, the frequency of changes, and the resources available.

Automated consistency checks rely on tools and scripts to detect inconsistencies quickly and consistently, making them suitable for large-scale or rapidly changing systems. In contrast, manual consistency checks depend on human inspection and reasoning, offering flexibility and deeper insight in complex or ambiguous scenarios. While automation excels in speed and repeatability, manual methods remain valuable for nuanced evaluations where human judgment is necessary. Together, both approaches play a complementary role in maintaining software quality and reliability. Below table 5.3.5 shows a clear comparison of Automated vs Manual Consistency Checking.

Table 5.3.5: Comparison of Automated vs Manual Consistency Checking

Aspect	Automated Consistency Checks	Manual Consistency Checks
Definition	Performed using tools or scripts to automatically detect inconsistencies across artifacts.	Conducted by developers, testers, or analysts by manually reviewing artifacts.
Speed	Fast and repeatable; runs in seconds to minutes depending on scope.	Time-consuming, especially for large systems.
Accuracy	High for defined rules; can miss context-specific issues.	Can identify subtle or complex issues if the reviewer has expertise.
Scalability	Easily scalable to large systems and frequent code changes.	Not scalable for frequent or large-scale consistency checks.
Tool Dependency	Requires setup and maintenance of tools (e.g., SonarQube, Jenkins, EMF).	No tools required, but relies heavily on individual effort and domain knowledge.
Human Error Risk	Low for repetitive checks; rules are consistently applied.	High risk due to oversight, fatigue, or misunderstanding.
Adaptability	Limited to what the tool is programmed to check.	Can adapt flexibly to complex, evolving, or domain-specific scenarios.
Examples	Code linting, model validation, schema checking, CI/CD pipeline tests.	Manually verifying UML diagrams, reviewing requirements traceability, inspecting documents.

5.3.5 Comparative Analysis of Software verifying methods

In software engineering, ensuring the quality and correctness of system design is a crucial step before implementation. Among the key techniques used for validating and verifying designs are design walkthroughs, critical design reviews (CDRs), and consistency checkers. While all aim to detect errors and improve the reliability of software artifacts, they differ significantly in their methods, formality, and scope. Design walkthroughs are informal peer-based evaluations, CDRs are structured and often conducted by external stakeholders, and consistency checkers rely on automated or semi-automated tools to detect mismatches across artifacts. This comparative analysis (Table 5.3.6) explores the unique characteristics, strengths, and limitations of each technique, highlighting their complementary roles in achieving robust and maintainable software designs.

Table 5.3.6: Comparative Analysis of Design Walkthroughs, Critical Design Reviews, and Consistency Checkers

Aspect	Design Walkthrough	Critical Design Review (Cdr)	Consistency Checkers
Definition	Informal peer review of design documents and logic.	Formal review of the system design by stakeholders before implementation.	Automated or semi-automated tools that detect mismatches across artifacts.
Formality	Informal and flexible	Highly formal with predefined criteria	Tool-based, formal or semi-formal
Participants	Designers, developers, and possibly testers	Project managers, developers, clients, QA teams	Mostly automated; may involve developers for interpretation
Focus Area	Logic flow, structure, and clarity of design	Compliance with requirements, feasibility, risks, and standards	Internal and cross-artifact consistency
Timing	During early or mid design phases	After major design milestones, before implementation	Continuously or before builds/deployments
Method	Manual walkthrough, often line-by-line	Structured meeting with documentation and decision logs	Analysis using static tools, constraints, schema validators, etc.
Output	Informal feedback, design improvements	Formal approval/rejection and action plans	Reports of inconsistencies or rule violations

Strengths	Encourages collaboration, easy to conduct	Thorough review with broad visibility	Fast, scalable, repeatable, good for large systems
Limitations	May miss formal issues; depends on team expertise	Time-consuming, requires scheduling and preparation	Limited to what the tool can check; lacks contextual judgment
Example Tools	N/A (whiteboards, docs, discussion-based)	N/A (meetings, presentations, checklists)	SonarQube, Alloy, Jenkins, EMF, JSON/XML validators
Best Use Case	Early-stage feedback and collaborative design refinement	Approving final design for critical or complex systems	Ensuring model/code/data consistency throughout the development cycle

Recap

- ◆ Design Walkthroughs are informal reviews where the author explains the design to peers for early feedback and error detection.
- ◆ Key roles in a walkthrough include the designer (presenter), developers, testers, and sometimes a moderator to guide the discussion.
- ◆ The main benefit of walkthroughs is fostering collaboration and catching design flaws early before they become costly to fix.
- ◆ One major drawback is that they rely heavily on the participants' expertise and may overlook formal or hidden issues.
- ◆ Important scenarios where design walkthrough can be used are Reviewing the Design of a New System Module, Reviewing a Complex User Interface (UI) Design.
- ◆ Critical Design Reviews (CDRs) are formal, structured meetings used to evaluate whether a system design meets all requirements before development.
- ◆ Planning a CDR involves scheduling the review at a key project milestone and selecting stakeholders such as project managers, QA, and clients.
- ◆ Preparation for CDRs includes compiling complete design documents, risk assessments, checklists, and a presentation to justify the design.

- ◆ CDRs help ensure that the design is technically sound, feasible, and aligned with business goals. However, they require significant time and resources and may introduce delays if documentation is incomplete or feedback is extensive.
- ◆ Consistency Checkers are tools and techniques used to detect mismatches within or between software artifacts like models, code, or requirements.
- ◆ Tools like SonarQube, JSONLint, Alloy Analyzer, and Jenkins are commonly used for automated consistency checks.
- ◆ Techniques include static analysis, model checking, constraint-based validation, and schema or configuration validation.
- ◆ Automated consistency checking is efficient, repeatable, and often integrated into CI/CD pipelines to support ongoing verification.
- ◆ Manual consistency checking, though slower, is still essential for interpreting complex cases that require human judgment or domain knowledge.

Objective Type Questions

1. What is the review type where the designer presents the design informally to peers?
2. Who typically conducts a design walkthrough?
3. What is a formal milestone-based review conducted before implementation?
4. What type of checking uses automated tools for validation?
5. What role guides and records feedback during a walkthrough?
6. Which tool is commonly used for code consistency checking?
7. What type of consistency check relies on human judgment?
8. What model-checking tool is used for formal consistency validation?
9. What kind of analysis is used in consistency checkers?
10. What stage is a CDR typically conducted in?
11. Design walkthroughs are generally _____ reviews where feedback is gathered from peers.

12. A Critical Design Review (CDR) is conducted to verify if the design meets _____ and technical expectations.
13. Manual consistency checks are less scalable but provide valuable _____-based insight.
14. Tools like Jenkins and SonarQube help in performing _____ consistency checks.
15. During a CDR, preparation includes risk analysis and compiling all relevant _____ documentation.

Answers to Objective Type Questions

1. Walkthrough
2. Designer
3. CDR (Critical Design Review)
4. Consistency
5. Moderator
6. SonarQube
7. Manual
8. Alloy
9. Static
10. Design
11. Informal
12. Requirements
13. Context
14. Automated
15. Design

Assignments

1. Describe the process of a Design Walkthrough. Who are the key participants, and what roles do they play in ensuring the success of the walkthrough?
2. Identify and explain at least three benefits and two drawbacks of using Design Walkthroughs in software development. Support your answer with an example scenario.
3. Explain the importance of planning and preparation in conducting a successful Critical Design Review (CDR). Compare Critical Design Reviews (CDRs) and Design Walkthroughs in terms of formality, objectives, participants, and outcomes.
4. List and explain the types of consistency checking techniques. Evaluate the role of automated and manual consistency checkers in modern software engineering and list the examples of tools that support this method.

Suggested Reading

1. TutorialsPoint–SoftwareEngineering-https://www.tutorialspoint.com/software_engineering/
2. Software Engineering Institute (SEI), Carnegie Mellon University-<https://sei.cmu.edu>
3. GeeksforGeeks–SoftwareEngineering-<https://www.geeksforgeeks.org/software-engineering/>
4. Coursera / edX (Software Engineering Courses)-<https://www.coursera.org> / <https://www.edx.org>

Reference

1. Ian Sommerville (2015), “*Software Engineering*” (10th Edition.), published by Pearson Education.
2. Roger S. Pressman and Bruce R. Maxim (2015), “*Software Engineering: A Practitioner’s Approach*” (8th Edition), published by McGraw-Hill Education.
3. Daniel Galin (2017), “*Software Quality Assurance: From Theory to Implementation*” (2nd Edition), published by Pearson Education.



Other UML Diagrams

Learning Outcomes

After completion of this unit, the learner will be able to:

- ◆ explore the concept of non-standard UML diagrams
- ◆ define the concept, components of a Profile Diagram in UML
- ◆ recall the key elements, purpose of a Composite Structure Diagram
- ◆ identify the definition, key concepts of an Interaction Overview Diagram
- ◆ list the components of a Timing Diagram
- ◆ recognize the examples, benefits and drawbacks of non-standard UML diagrams
- ◆ comparison of non-standard UML diagrams

Prerequisites

To effectively work with non-standard UML diagrams like Profile Diagrams, Composite Structure Diagrams, Interaction Overview Diagrams, and Timing Diagrams, a strong understanding of core UML concepts is essential. Knowledge of standard UML diagrams such as class, sequence, and activity diagrams is foundational, as these are frequently extended or referenced in non-standard diagrams. For example, in a smart home automation system, a Profile Diagram might be used to customize the system's UML model with domain-specific stereotypes such as «Sensor», «Thermostat», and «ControlUnit». Understanding how to extend UML with profiles is crucial for such domain-specific applications.



Additionally, more advanced concepts such as component-based design and real-time system modeling are necessary for working with Composite Structure Diagrams and Timing Diagrams. For example, in a telecommunication network, a Composite Structure Diagram could be used to model how different network components (like routers, switches, and transmission lines) interact within a larger network architecture. Similarly, a Timing Diagram might be applied to a medical device system, like a pacemaker, to model time-sensitive events such as sensor readings, response times, and state changes over time. These diagrams are used to represent real-time system behavior, where precise timing of interactions and events is critical.

Keywords

Profile Diagram, Composite Structure Diagram, Interaction Overview Diagram, Timing Diagram.

Discussion

5.4.1 Introduction to Other UML Diagrams

A standardized method for designing, documenting, and visualizing software systems is offered by the Unified Modeling Language (UML). Despite being widely used for generic system modeling, core diagrams like class, use case and sequence diagrams might not adequately represent the complexity of specialized or domain-specific systems.

Non-standard (or non-core) UML diagrams are useful in this situation. Additional modeling capabilities tailored to specific aspects of a system's architecture or behavior are provided by these diagrams, which include profile diagrams, composite structure diagrams, interaction overview diagrams, and timing diagrams. They are especially helpful in situations where more accuracy, structural specifics or unique additions beyond the general-purpose diagrams are needed.

By leveraging these non-standard diagrams, software architects and system designers can model intricate internal behaviors, real-time constraints, and domain-specific features more effectively, ensuring that complex systems are both well-documented and accurately represented.

5.4.1.1 Purpose of Non-Standard UML Diagrams

Non-standard UML diagrams also referred to as non-core or specialized UML diagrams extend the modeling capabilities of UML beyond the commonly used diagrams like class, use case, or sequence diagrams. Their primary purpose is to provide more precise, domain-specific, or structurally detailed representations in complex or specialized systems. The key purposes of Non-standard UML diagrams are:

1. **Modeling Specialized Concepts:** Diagrams like Profile Diagrams allow for customization of UML to represent domain-specific elements using stereotypes, tagged values, and constraints.
2. **Detailed Internal Structures:** Composite Structure Diagrams help model the internal architecture of a classifier, revealing how its parts collaborate internally useful in embedded systems and complex software.
3. **Precise Interaction Behavior:** Interaction and Timing Diagrams capture exact communication sequences and time-sensitive behavior, essential in real-time and performance-critical applications.
4. **Enhancing Clarity in Complex Systems:** These diagrams break down complex systems into comprehensible components or timelines, aiding in system understanding, design, and documentation.
5. **Extending UML for Specific Domains:** With tools like profiles and stereotypes, organizations can adapt UML to specific industries (e.g., aerospace, healthcare, automotive) without altering the core language.
6. **Facilitating Code Generation and Validation:** By using detailed, structured models, some tools can auto-generate code, configurations, or perform static validation against rules and constraints.

5.4.1.2 When to use these Specialized Diagrams?

Specialized UML diagrams are especially helpful when regular, widely used UML diagrams are insufficient to convey the complexities of a system's behavior, architecture, or domain-specific needs. In complicated systems, real-time applications, and situations where the UML language itself needs to be customized, these diagrams are especially helpful.

Use Profile diagrams when you need to enhance UML for a specific domain. In order to develop unique modeling features suited to particular industries, like healthcare, automotive, or aerospace, they enable the construction of stereotypes, tagged values, and constraints. Additionally, they are crucial in model-driven engineering, which calls for specialized modeling frameworks.

Composite Structure diagrams are useful for demonstrating how a class or component's internal structure works and how its many elements work together. These diagrams are perfect for systems like embedded systems or component-based architectures that are composed of several interdependent subcomponents. They shed light on the connections and information flow between components.

Interaction Overview diagrams are perfect for modeling intricate processes or use cases with numerous steps and interactions because they offer a high-level perspective of control flow across various interactions. When coordinating and visualizing the sequence and logic such as decisions, loops, or parallel paths between various interaction diagrams, such as sequence or communication diagrams, they are very helpful. By arranging complex behavior in an organized manner, these diagrams assist stakeholders and system designers in comprehending how different system components

interact over time as part of a larger process. They are frequently used to illustrate the overall structure of dynamic behavior during system analysis or design.

Timing diagrams are made especially for embedded and real-time systems where it's important to time events and state transitions precisely. They are crucial when working with time limitations, sensor/actuator coordination, or applications that require high performance because they enable you to represent how objects or systems behave over time.

5.4.2 Profile Diagrams

Profile Diagrams are a specialized type of UML (Unified Modeling Language) diagram used to customize UML models for particular domains or platforms without altering the underlying UML metamodel. They allow developers to define stereotypes, tagged values, and constraints, effectively extending the semantics of standard UML elements to suit specific needs. Unlike structural or behavioral diagrams, Profile Diagrams are not used to model system architecture or behavior directly but rather to define reusable extensions that can be applied across other UML diagrams like Class, Component, or Sequence Diagrams. This makes them especially useful in domain-specific modeling, such as software for embedded systems or enterprise applications.

5.4.2.1 Definition, Purpose and Importance

A Profile Diagram is a kind of structural UML diagram that allows the UML standard to be extended by adding restrictions, tagged values, and custom stereotypes. These components are used to customize pre-existing UML metaclasses (such as Class, Component, or Activity) for a particular development environment or domain. Profile Diagrams offer a simple and adaptable way to adjust UML to unique modeling requirements without changing the basic UML metamodel. This makes UML perfect for developing domain-specific modeling languages (DSMLs).

The main purpose of a Profile Diagram is to allow developers and architects to represent domain-specific concepts directly in UML models by applying custom stereotypes and rules. This makes models more expressive, relevant, and easier to understand for a particular application area such as embedded systems, business processes, or cloud architectures. Additionally, profiles promote standardization within teams or organizations by formalizing best practices and ensuring consistent application of domain rules across multiple projects. Profile diagrams in UML are important because this will:

- ◆ **Customize UML:** Allow customization for specific domains or industries.
- ◆ **Clarify Semantics:** Define precise meanings and behaviors of model elements.
- ◆ **Ensure Consistency:** Establish standards and enforce best practices.
- ◆ **Support Automation:** Enable tooling support for validation and code generation.

- ◆ **Promote Reusability:** Facilitate reuse across projects and scalability.
- ◆ **Enhance Communication:** Serve as clear documentation for stakeholders.

In software development, profile diagrams often enhance modeling accuracy, consistency, and communication.

5.4.2.2 Components of a Profile Diagram

In the context of UML (Unified Modeling Language), a profile diagram usually depicts constraints, tagged values, and stereotypes that extend UML metamodel elements. The essential elements of a profile diagram are listed below:

1. Profile: This is the primary container in the diagram that shows a group of restrictions, tagged values, and stereotypes. It acts as an extension mechanism that allows UML to be customized for particular fields or uses (Fig 5.4.1).



Fig 5.4.1: Profile in Profile Diagram

2. Stereotypes: Stereotypes can modify pre-existing model elements or define new ones. Their names are indicated by guillemets (<<>>). For example, a stereotype might be <<Entity>> or <<Service>>, which can be applied to classes or components to give them specific meanings or behaviors beyond standard UML (5.4.2).

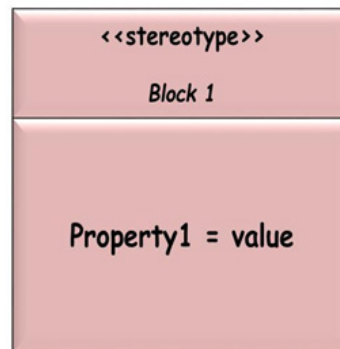


Fig 5.4.2: Stereotypes in Profile Diagram

3. Metaclass: Stereotypes can be applied to standard UML elements like classes, interfaces, components, etc., which are known as metaclasses. A profile diagram shows these standard UML elements to illustrate how stereotypes are applied to them (Fig 5.4.3).

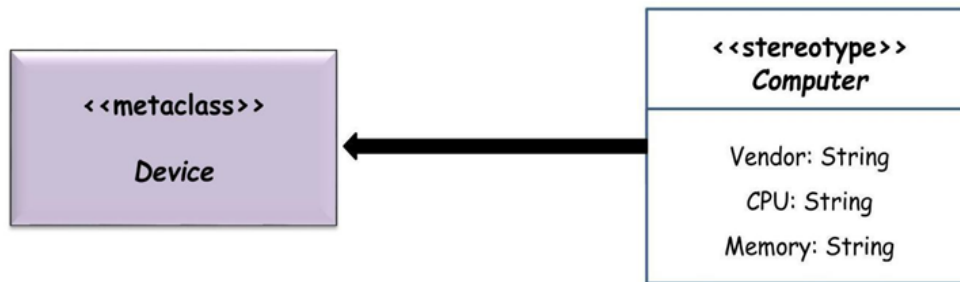


Fig 5.4.3: Metaclass in Profile Diagram

4. Tagged Values: Stereotypes may be coupled with tagged values that offer extra details or characteristics specific to the stereotype. As an example, a `<>` stereotype may have tagged values such as `persistent` or `tableName` (Fig 5.4.4).

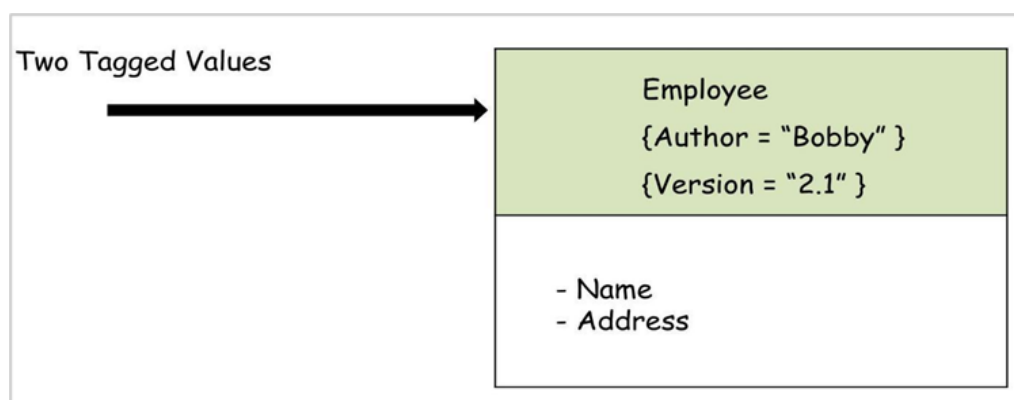


Fig 5.4.4: Tagged Values in Profile Diagram

5. Constraints: Constraints define rules or conditions that apply to elements annotated with a particular stereotype. These are expressed in a formal language like OCL (Object Constraint Language) and ensure that the model adheres to specific rules defined by the profile (Fig 5.4.5).

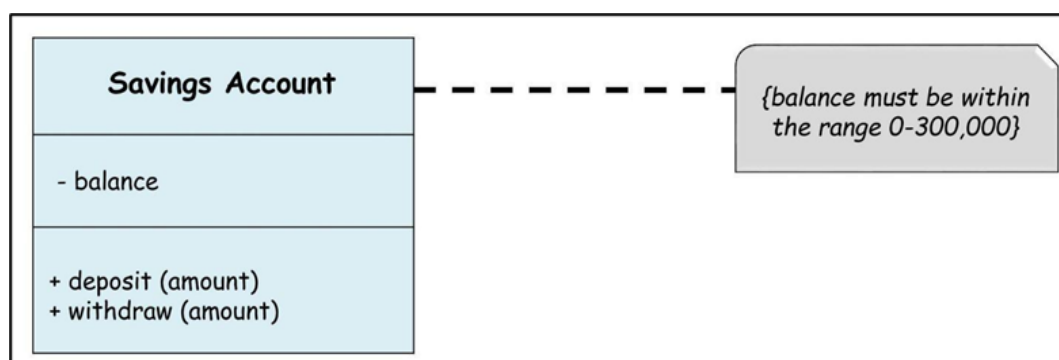


Fig 5.4.5: Constraints in Profile Diagram

6. Extensions: Extensions show how stereotypes extend the standard UML metamodel elements. This can be depicted as a relationship between the stereotype and the metaclass it extends. For example, a stereotype `<<Entity>>` might extend the `Class` metaclass (Fig: 5.4.6).

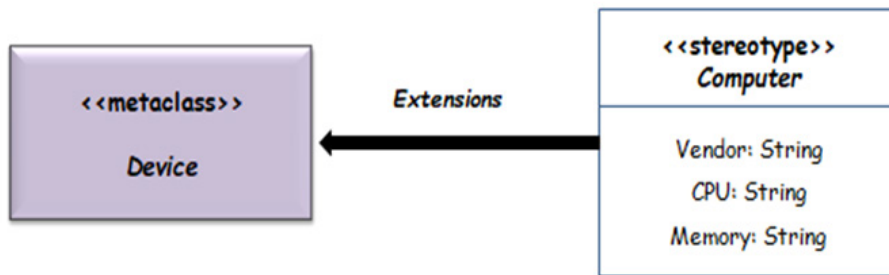


Fig 5.4.6: Extensions in Profile Diagram

5.4.2.3 Example of a UML Profile Diagram

In IT management, strategic planning and operational effectiveness depend on the proper representation and management of various network components, including workstations, servers, and switches/routers (Fig:5.4.7).

To accurately define these components and make sure they comply with organizational IT management policies, the task is to customize UML using profiles, such as the Network profile. The goal of this profile diagram customisation is to improve IT infrastructure management decision-making, standardization, and clarity. Detailed explanation of the example is depicted below:

1. Packages and Profiles:

- ◆ **ITManagement Package:** This package represents the domain of IT Management, encompassing various IT assets and management entities.
- ◆ **Profiles (Network, Telecomms, Software):** Profiles are used to extend UML and define domain-specific concepts, stereotypes, and constraints.
- ◆ **Network Profile:** This profile focuses on defining stereotypes related to network components within IT management.
- ◆ **Stereotypes:**
 - A. Workstation: Represents a workstation device within the network. It inherits from NetworkNode.
 - B. Server: Represents a server within the network infrastructure. It inherits from NetworkNode.
 - C. Switch/Router: Represents a network switch or router. It inherits from NetworkNode.
 - D. User (class): Represents a user entity within the IT system. This stereotype is defined directly within the Network profile, indicating it relates specifically to network-related users or roles.

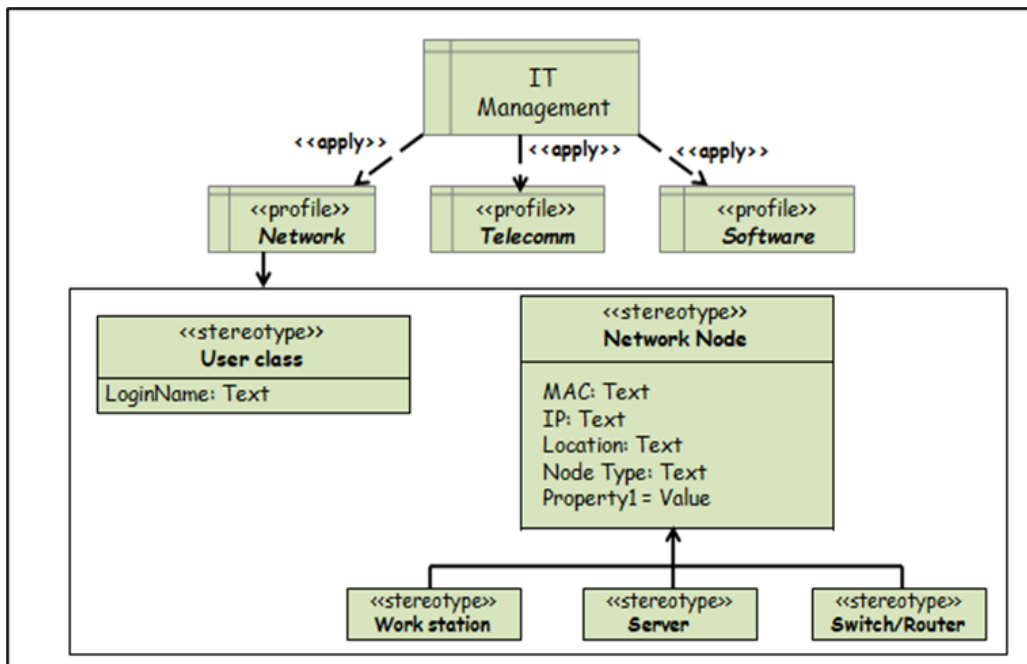


Fig 5.4.7:Example of an IT management Profile Diagram

2. Relationships and Application:

- ◆ **Application of Profiles:** The diagram visually shows how the Network, Telecomms, and Software profiles are applied to the ITManagement package. This application is typically represented by a dependency or applied stereotype relationship between the profiles and the ITManagement package.
- ◆ **Extension of Stereotypes:** Stereotypes like Workstation, Server, and Switch/Router extend from the NetworkNode stereotype. This inheritance relationship (extension) ensures that the specific attributes, behaviors, and roles defined for NetworkNode are inherited by these more specific stereotypes.

3. Purpose and Benefits:

- ◆ **Domain-Specific Modeling:** By using profiles, the UML language is customized to better represent and manage IT assets specific to IT management. This enhances clarity and communication among stakeholders by aligning the modeling language directly with domain-specific concepts and requirements.
- ◆ **Standardization and Reusability:** Profiles allow for the standardization of modeling practices across different projects within IT management. They also promote reusability of defined stereotypes and constraints, reducing redundancy and improving consistency in modeling.

In conclusion, the IT Management profile diagram shows how profiles (Network, Telecoms, Software) enhance the UML language to specify certain stereotypes (such as Workstation, Server, Switch/Router) and relationships relevant to the ITManagement package. In the framework of IT management procedures, this customisation improves

the modeling capabilities to precisely depict and manage IT infrastructure, components, and roles.

5.4.2.4 Benefits and Drawbacks

Profile Diagrams play a crucial role in extending UML to better represent specific domains and application needs. By allowing the definition of custom stereotypes, tagged values, and constraints, they provide a flexible and standardized way to tailor UML models without altering the core UML metamodel. This adaptability makes Profile Diagrams especially valuable in complex or specialized systems, where standard UML elements alone may not be sufficient to capture all relevant concepts and rules. Below are the benefits of using profile diagram:

1. **Enhanced Expressiveness:** Profile diagrams allow UML to be extended with domain-specific concepts, making the modeling language more expressive and relevant to real-world scenarios.
2. **Improved Communication:** By customizing UML with a profile, stakeholders can communicate using a modeling language that directly reflects their domain knowledge and requirements, reducing misunderstandings.
3. **Tool Support and Automation:** Profiles enable modeling tools to understand and support domain-specific constructs, enhancing tool capabilities such as validation, code generation, and simulation.
4. **Consistency and Reusability:** Profiles promote consistency in modeling practices across projects or organizations by standardizing extensions and customizations, which can be reused in future projects.
5. **Flexibility and Adaptability:** Organizations can adapt UML to changing project needs or evolving industry standards by modifying or extending profiles, ensuring that the modeling language remains relevant over time.

While Profile Diagrams offer powerful capabilities for extending UML to suit domain-specific needs, they are not without limitations. Their flexibility and customization potential can introduce challenges in modeling, collaboration, and tool compatibility. Understanding the potential drawbacks of using Profile Diagrams is essential to ensure they are applied effectively and do not compromise the clarity, maintainability, or interoperability of the UML models.

1. **Complexity and Learning Curve:** Profile Diagrams introduce additional layers of abstraction and customization, which can make models more complex and harder to understand—especially for teams unfamiliar with UML extension mechanisms.
2. **Tool Support Limitations:** Not all UML modeling tools offer full or consistent support for profile diagrams, especially when it comes to enforcing constraints or generating code from customized elements. This can limit the practical utility of profiles and lead to inconsistencies across tools.
3. **Overhead in Maintenance:** Custom profiles must be maintained and

updated as domain requirements evolve. This can create additional work for teams, particularly in ensuring that all diagrams using the profile remain compatible with updated versions.

4. **Reduced Portability:** Models that rely heavily on domain-specific profiles may not be easily portable across different organizations or tools, making collaboration and model reuse more difficult in heterogeneous environments.
5. **Risk of Misuse:** Without clear guidelines or proper governance, teams may overuse or incorrectly apply stereotypes and tagged values, leading to bloated or semantically incorrect models that undermine the benefits of standardization.

5.4.2.5 Tools for Designing Profile Diagrams

Designing Profile Diagrams requires specialized UML modeling tools that support the creation and application of custom stereotypes, tagged values, and constraints. These tools enable modelers to extend standard UML elements effectively and ensure that custom profiles are integrated seamlessly into broader system models. By offering visual interfaces, validation features, and sometimes code generation capabilities, these tools streamline the process of domain-specific modeling and enhance productivity. Choosing the right tool is essential for ensuring compatibility, usability, and adherence to UML standards. Several tools can help you create UML Profile Diagrams, including:

- ◆ **Enterprise Architect:** A comprehensive UML modeling tool with extensive support for profiles.
- ◆ **MagicDraw:** Offers powerful modeling capabilities and support for UML profiles.
- ◆ **Visual Paradigm:** A versatile tool that supports UML and custom profile creation.
- ◆ **UMLet:** A lightweight tool for creating UML diagrams, including profiles.
- ◆ **Sparx Systems UML:** Another robust tool for UML modeling with profile support.

5.4.3 Composite Structure Diagrams

In the Unified Modeling Language (UML), a sort of diagram called a composite structure diagram is used to show how a class is organized internally and how its components interact with one another. Composite structure diagrams go deeper into the makeup of a system, showing how different sections (or components) are connected together and work together to accomplish the class's function, in contrast to standard class diagrams that concentrate on exterior interfaces and interactions. These diagrams are particularly helpful for simulating complicated systems where it is crucial to comprehend internal dynamics like data flow and control.

The parts, ports and connectors are important components of composite structure diagrams. Internal roles or components are represented by parts, communication interaction points are ports, and the connections or communication channels between

the parts are defined by connectors. In order to promote a more modular and adaptable design, these diagrams assist developers and architects in seeing how a system is organized within. Composite structure diagrams help improve design choices by providing a clear picture of how different components within a class work together, particularly in systems that rely significantly on dynamic configurations and reusable components.

5.4.3.1 Parts, Ports, and Connectors

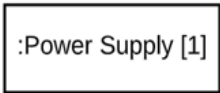
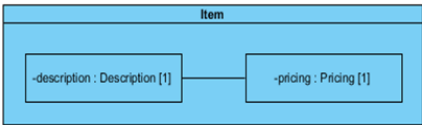
Parts, Ports, and Connectors are the core building blocks (Table 5.4.1) of Composite Structure Diagrams in UML used to model the internal structure and collaborative behavior within a system.

Parts represent internal components or roles within a classifier. Each part is an instance specification that defines a specific function or responsibility the classifier must perform. Parts are shown as rectangles inside a class or component, and they can be typed by other classifiers. They allow a system to be broken into smaller, manageable elements, each handling a portion of the overall behavior.

Ports are interaction points on the boundary or inside a classifier, specifying how a part can communicate with the environment or other parts. Ports define the interface(s) a part provides or requires and are typically shown as small squares on the edge of a classifier or part. They promote encapsulation by hiding internal details and exposing only the necessary features.

Connectors link parts and ports together to represent communication paths. A connector indicates that instances of the connected elements will interact at runtime, either through message passing, data flow, or control signals. Connectors can also define the contract or protocol that governs the interaction, making them essential for modeling structured collaboration.

Table 5.4.1: Parts, Ports and Connectors

Symbol	Name	Description
	Part	Acts as a runtime instance of classes or interfaces
	Port	Acts as an interaction point between a classifier instance (or its behavior) and its environment

Connector	Illustrates communication between parts

Together, these elements form a detailed and organized view of how a system's internal components are structured and how they interact, supporting better modularity and design clarity. The following composite structure diagram explained the concept of data types (Fig: 5.4.8).

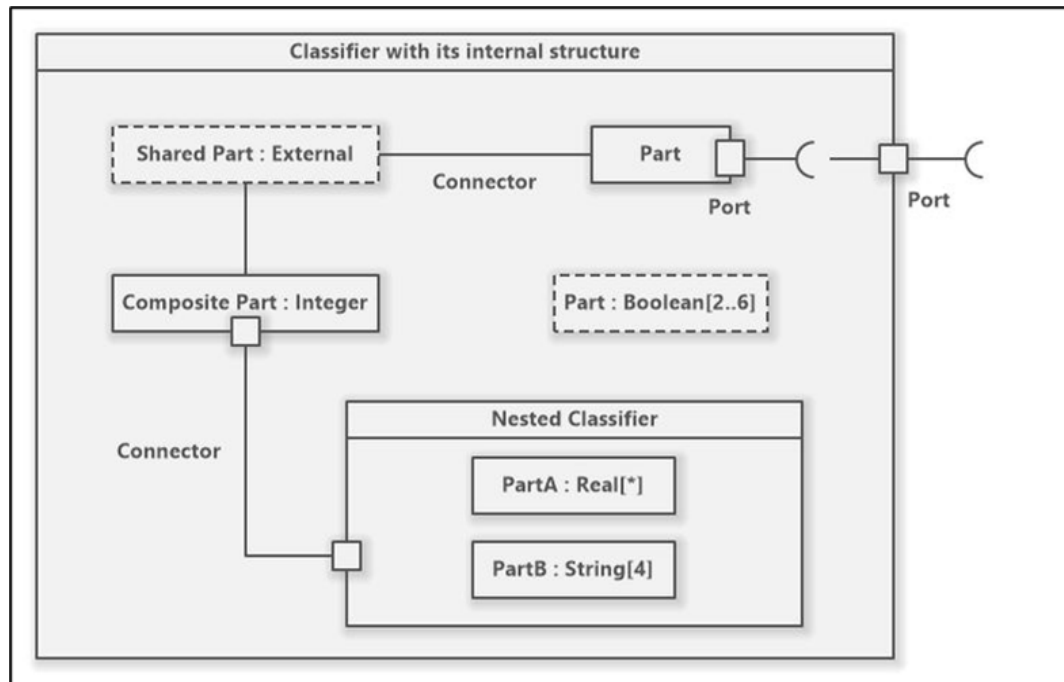


Fig 5.4.8: Sample Composite Structure Diagram

5.4.3.2 Modeling Internal Structure of Classifiers

In UML, Composite Structure Diagrams are used to model the internal structure of classifiers, particularly focusing on the configuration of parts and their relationships, including how they interact with the outside world through ports. This is useful for visualizing the implementation details of classes, components, or deployment nodes, showcasing the objects and roles involved in their behavior.

How to Model Internal Structure:

- ◆ **Open a Composite Structure Diagram:** Create a new diagram or display the structure compartment of a classifier.
- ◆ **Model Parts:** Represent the instances owned by the classifier.
- ◆ **Define Ports:** Specify the interaction points for external communication or communication between internal parts.

- ◆ **Add Connectors:** Establish associations or dependencies between parts, ports, or other elements.
- ◆ **Refine and Iteratively Refine:** Continue to refine the diagram as the internal structure of the classifier is understood and evolves.

Example: Imagine a "Computer" class. A Composite Structure Diagram for it might show:

- ◆ **Parts:** CPU, RAM, Hard Drive, Motherboard, etc.
- ◆ **Ports:** Power Input, USB, Ethernet, etc.
- ◆ **Connectors:** Relationships between the parts (e.g., RAM connecting to the CPU via the motherboard) and ports (e.g., USB port on the motherboard).

By using these elements, Composite Structure Diagrams provide a detailed and visual representation of the internal workings of complex classifiers, aiding in understanding and communicating their structure and behavior.

5.4.3.3 Sample Composite Structure Diagram

A Composite Structure Diagram in UML visually represents the internal structure of a class or collaboration, focusing on its parts and how they interact. It's like a zoomed-in view of a class, showing its internal composition and the relationships between those components.

Imagine a class called "Car" with internal parts like "Wheel", "Engine", and "SteeringWheel". A Composite Structure Diagram for "Car" would visually represent this:

Explanation:

1. **Car (Classifier):** The main element of the diagram, representing the class "Car".
2. **Parts:** "Wheel", "Engine", and "SteeringWheel" are depicted as internal parts within the "Car" classifier.
3. **Connectors:** Lines or symbols could connect the parts to represent how they interact. For example, a connector between "Wheel" parts might represent the axle.
4. **Ports:** If the "Car" class has interfaces for interaction with other parts of the system (like a "drive" port for the "Engine"), these could be shown as ports on the boundary of the "Car" classifier.

The diagram (Fig. 5.4.9) shows what's inside a "Car" and how its components (like wheels, engine, and steering wheel) are arranged and connected. It's a way to visualize the internal design of the class and how its parts work together to achieve a specific behavior.

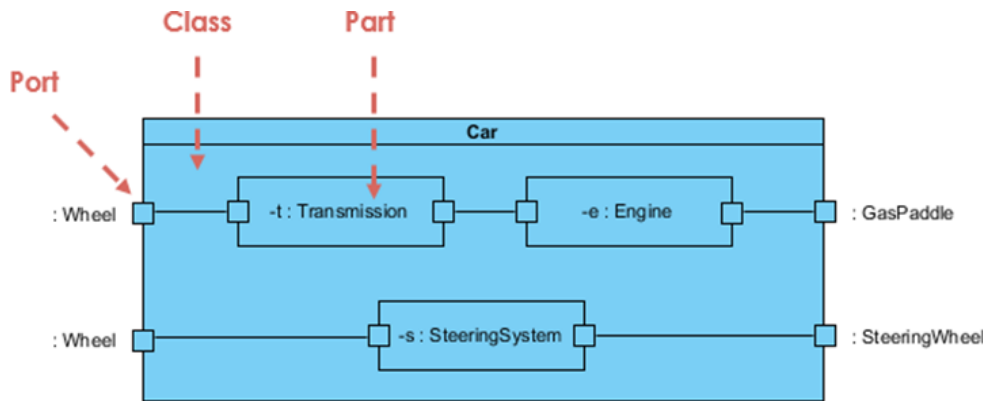


Fig 5.4.9: Sample Composite Structure Diagram: A Car

5.4.3.4 Benefits and Drawbacks

Composite Structure UML Diagrams offer significant advantages in system design by providing a detailed and organized view of a system's internal structure and interactions. They help architects and developers visualize how individual components, or parts, collaborate within a classifier using defined interfaces and communication paths. This internal perspective enhances modularity, supports encapsulation, and encourages component reusability. Especially in complex or component-based systems, these diagrams improve clarity and communication across development teams, enabling better design decisions and a more maintainable architecture. Benefits of Composite Structure UML Diagrams are:

1. **Detailed Internal View:** They provide a clear and structured visualization of the internal workings of a system or component, making it easier to understand complex behaviors and collaborations.
2. **Improved Modularity:** By breaking down systems into smaller, manageable parts (with ports and connectors), composite structure diagrams promote modular design, making systems easier to develop, test, and maintain.
3. **Supports Encapsulation:** Ports allow a classifier to expose only necessary interfaces, promoting encapsulation and reducing tight coupling between components.
4. **Facilitates Reusability:** Clearly defined parts and interfaces make it easier to reuse components in different contexts or systems.
5. **Better Communication:** They serve as a useful communication tool among stakeholders, especially in large or component-based systems, by providing a high-level view of internal collaboration.

Despite their advantages, Composite Structure UML Diagrams come with certain drawbacks that can limit their practicality in some contexts. Their complexity and specialized notation can be challenging for beginners or teams unfamiliar with advanced UML concepts. In simpler systems, they may introduce unnecessary overhead without providing substantial value. Additionally, limited tool support and the need for manual updates can make maintaining these diagrams labor-intensive, especially as system

designs evolve. Their abstract nature also means they often need to be complemented by other diagrams to fully capture system behavior. Drawbacks of Composite Structure UML Diagrams are:

1. **Complexity for Beginners:** The notation and concepts (e.g., ports, parts, connectors) can be difficult to grasp for those new to UML or system modeling.
2. **Overhead in Simple Systems:** For small or less complex systems, using composite structure diagrams may add unnecessary complexity and not provide significant value.
3. **Tool Support Limitations:** Not all UML tools have strong support for creating and maintaining composite structure diagrams, especially in terms of code generation or round-trip engineering.
4. **Abstract Representation:** These diagrams often lack behavioral details, which must be supplemented with other diagrams (like sequence or state diagrams) to get a full picture of the system.
5. **Maintenance Challenges:** As systems evolve, keeping the composite structure diagrams in sync with the actual code and design can require significant effort.

5.4.4 Interaction Overview Diagrams

Interaction Overview Diagrams are a type of behavioral diagram in the Unified Modeling Language (UML) used to visualize the flow of control between interactions within a system. These diagrams combine features of activity diagrams and sequence diagrams, allowing modelers to capture high-level control flow along with detailed interaction scenarios. Each interaction within the diagram is typically represented by a frame that references another interaction diagram, such as a sequence or communication diagram. This makes interaction overview diagrams particularly useful for modeling complex workflows where multiple interactions are coordinated.

The strength of interaction overview diagrams lies in their ability to present a structured, bird's-eye view of a system's dynamic behavior without getting lost in low-level details. They are especially helpful in scenarios where understanding the overall process is more important than focusing on individual message exchanges. Elements like decision nodes, forks, joins, and interaction use nodes allow designers to depict branching, parallelism, and looping within interaction flows, making these diagrams ideal for modeling business processes, user interactions, and system use cases in a modular and scalable way.

5.4.4.1 Purpose of Interaction Overview Diagram

An Interaction Overview Diagram (IOD) is a type of UML (Unified Modeling Language) diagram that illustrates the flow of interactions between various elements in a system or process. It provides a high-level overview of how interactions occur, including the sequence of actions, decisions, and interactions between different components or objects.

Interaction diagrams give a general overview of system behavior that stakeholders can understand without getting bogged down in specifics. They help with requirements analysis, documentation, communication, and system design. Overall, they enable stakeholders to comprehend complicated systems and make educated decisions.

5.4.4.2 Notations for Interaction Overview Diagram

Annotations and explanations from the Interaction Overview UML diagrams feature several fundamental components that help to show control flow and system interactions. Here's a detailed explanation of these notations:

1. Initial Node: Represents the starting point of the interaction. It is shown as a small solid circle with a single outgoing control flow arrow (Fig 5.4.10).

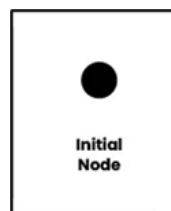


Fig 5.4.10: Initial Node

2. Interaction Use: An Interaction Use is used to represent a reference to another interaction or sequence of interactions defined in another diagram or part of the system. An Interaction Use is represented by a rectangular box. Inside the box, you can include a label that specifies the name or reference of the interaction being used (Fig 5.4.11).

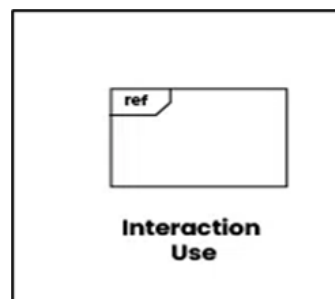


Fig 5.4.11: Interaction Use

3. Interaction Fragment: Interaction fragments are used to group and organize interactions within the diagram (Fig 5.4.12).

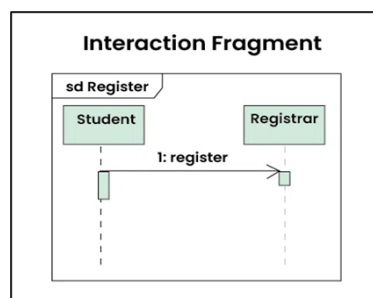


Fig 5.4.12: Interaction Fragment

4. Object Lifeline: Represents the lifespan of an object during the interaction. It is a vertical dashed line that extends from the top of the diagram to the bottom, with the object's name or class at the top (Fig 5.4.13).

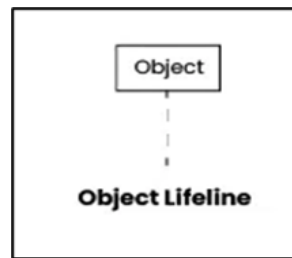


Fig 5.4.13: Object Lifeline

5. Message: Represents communication between objects or components in the system. Messages are shown as arrows between lifelines, indicating the direction of communication (Fig 5.4.14).

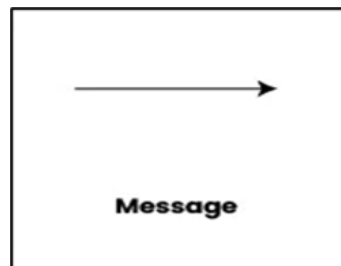


Fig: 5.4.14: Message

6. Control Flow: Represents the flow of control or sequence of actions in the interaction. It is shown as a solid arrow between elements in the diagram, indicating the order in which actions are performed (Fig 5.4.15).

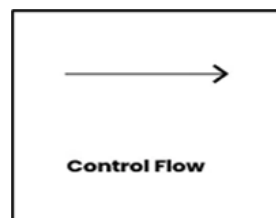


Fig 5.4.15: Control Flow

7. Fork Node/Join Node: Represents a point in the interaction where multiple parallel actions can occur simultaneously. It is shown as a solid rectangle with multiple outgoing control flow arrows in case of Fork Node and ingoing control flow arrows in case of Join Node (Fig 5.4.16).

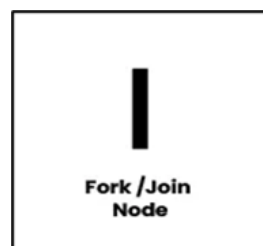


Fig 5.4.16: Fork Node/Join Node

8. Decision Node: Represents a point in the interaction where a decision is made based on a condition. It is shown as a small diamond shape with incoming and outgoing control flow arrows representing different paths based on the decision (Fig 5.4.17).

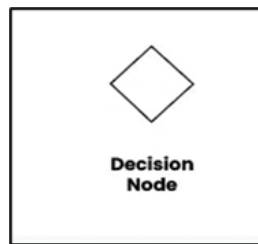


Fig 5.4.17: Decision Node

9. Receive Signal Node: Represents the receipt of a signal or event during the interaction. It is shown as a small rectangle with an incoming control flow arrow (Fig: 5.4.18).

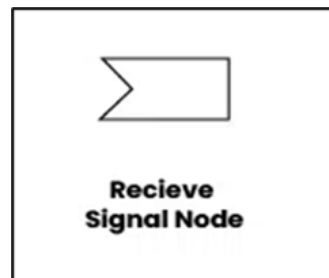


Fig 5.4.18: Receive Signal Node

10. Final Node: Represents the ending point of the interaction. It is shown as a small solid circle with no outgoing control flow arrows (Fig: 5.4.19).

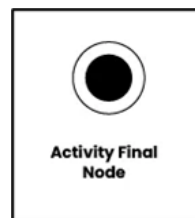


Fig 5.4.19: Final Node

11. Note: Represents additional information or comments about the diagram. Notes are shown as a small rectangle with a label indicating the content of the note (Fig:5.4.20).

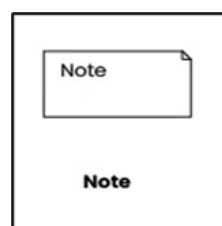


Fig 5.4.20: Note

5.4.4.3 Benefits and Drawbacks

Interaction Overview Diagrams offer several benefits by providing a high-level visualization of complex interaction flows within a system. They combine the clarity of control flow modeling from activity diagrams with the detailed interaction structure of sequence diagrams, enabling modular and reusable representations of behavior. This makes them especially useful for illustrating how multiple interactions work together in business processes, workflows, or system operations, all while maintaining readability and supporting better design and communication across development teams. Benefits are:

1. **High-Level Clarity:** They provide a clear overview of complex interaction flows by abstracting and organizing multiple interaction scenarios into a single diagram, making it easier to understand overall system behavior.
2. **Modular Design:** By referencing other interaction diagrams (like sequence or communication diagrams), they promote modular and reusable modeling, reducing redundancy and improving maintainability.
3. **Combines Control Flow and Interaction:** These diagrams merge the strengths of activity diagrams (control flow) with interaction diagrams (message flow), enabling a comprehensive view of both behavior and structure.
4. **Supports Complex Processes:** Useful for modeling systems with branching, parallel execution, and alternative flows, such as business processes or multi-step workflows.

Despite their usefulness, Interaction Overview Diagrams come with certain drawbacks that can limit their practicality in some scenarios. They often require supplementary diagrams to convey detailed behavior, which adds to modeling overhead and complexity. For larger systems, these diagrams can become difficult to manage and interpret due to nested interactions and control nodes. Additionally, limited support in some UML tools and the learning curve associated with understanding their combined elements can make them less accessible to beginners or non-technical stakeholders. Drawbacks are:

1. **Increased Complexity:** For large systems, the diagram itself can become complex and harder to manage, especially if many nested interactions are involved.
2. **Requires Supporting Diagrams:** Interaction Overview Diagrams often depend on other interaction diagrams to show detailed behavior, which means additional work and documents are needed to get the full picture.
3. **Limited Tool Support:** Some UML tools provide limited features or poor visualization for interaction overview diagrams compared to more commonly used diagrams like class or sequence diagrams.
4. **Steep Learning Curve:** Because they combine elements of multiple diagram

types, they may be harder for beginners or non-technical stakeholders to interpret without prior UML knowledge.

5.4.4.4 How to draw an Interaction Overview Diagram

To draw an Interaction Overview Diagram in UML, follow these structured steps to visually represent the control flow among different interactions:

Step 1: Identify Interactions

Identify the interactions between different elements in your system or process that you want to represent in the diagram. These interactions can include method calls, message exchanges, and other types of communication.

Step 2: Identify Fragments

Determine the types of fragments you will need to use to represent the interactions. This includes identifying loops, alternatives, options, and parallel sequences in the interactions.

Step 3: Create Lifelines

Draw vertical dashed lines (lifelines) for each object or component involved in the interactions. Label each lifeline with the name or class of the object.

Step 4: Add Messages

Use arrows to represent messages between lifelines, indicating the flow of communication between objects. Label the arrows with the type of message (synchronous, asynchronous, return).

Step 5: Add Fragments

Use rectangular boxes to represent fragments, such as combined fragments, interaction operands, or interaction uses. Label each fragment with the type of fragment or the interaction being referenced.

Step 6: Connect Fragments

Use control flow arrows to connect fragments and elements in the diagram, indicating the flow of control between them. Use solid arrows for regular control flow and dashed arrows for exception flow.

Step 7: Add Control Nodes

Use small circles to represent control nodes, such as decision nodes, fork nodes, initial nodes, and final nodes. Place them at appropriate points in the diagram to indicate decisions, parallel actions, and the start and end of the interaction.

Step 8: Review and Validate

Review the diagram to ensure that it accurately represents the interactions and behavior of the system. Validate the diagram against the requirements and specifications of the system to ensure correctness.

Step 9: Refine and Iterate

If necessary, refine the diagram based on feedback and further analysis. Iterate on the diagram until you have a clear and accurate representation of the interactions in the system.

5.4.4.5 Example of Interaction Overview Diagram

The below diagram (Fig: 5.4.21) depicts the interaction overview diagram of the Student Admission System. This example illustrates the admission process for a student who has been accepted into a university. Initially, the student is required to either accept or decline the admission offer. Upon acceptance, the student must proceed to register for classes and apply for housing. Once these steps are completed, the student is then required to make payment to the registrar. Failure to submit payment on time will result in the student being excluded by the registrar.

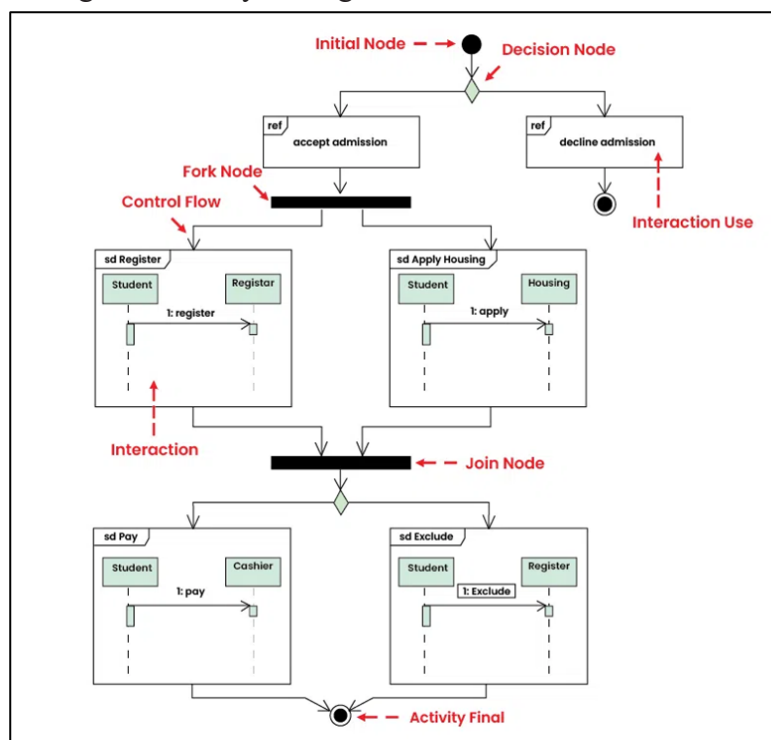


Fig 5.4.21: Interaction Overview Diagram of the Student Admission System

5.4.5 Timing Diagrams

A Timing Diagram in UML is a type of interaction diagram that focuses on the behavior of objects over time. It is particularly useful for modeling and analyzing real-time systems or time-sensitive interactions. Unlike sequence diagrams that emphasize the order of messages, timing diagrams emphasize time constraints, duration, and state changes of one or more objects in a linear time axis. This makes them ideal for visualizing how an object's state evolves in response to events or stimuli over a specified period.

In a timing diagram, time is typically represented on the horizontal axis, while different lifelines or objects are listed vertically. Each lifeline shows changes in state or value, often as continuous or discrete lines, with transitions triggered by specific events.

This format allows developers to analyze issues such as latency, synchronization, or timeouts in a system. Timing diagrams are widely used in embedded systems, control systems, and any domain where the timing of operations is critical to functionality.

5.4.5.1 Key concepts of Timing Diagram

The key components of a Timing Diagram in UML (Fig: 5.4.22) are designed to represent how objects change state or interact over time, with a strong focus on time constraints and sequences. Here are the main components:

1. Lifeline

A lifeline represents an individual object, actor, or system component whose behavior or state is being observed over time. It is usually displayed vertically on the diagram, with a name identifying the entity it represents. The lifeline acts as a reference point for tracking how that particular object's state or values change as time progresses. In a timing diagram, each lifeline typically corresponds to one waveform, which visualizes the transitions of that object.

2. Time Axis

The time axis is a horizontal timeline that runs from left to right across the diagram. It represents the flow of time and is usually marked with time intervals to provide a reference for when events or state changes occur. This axis helps determine the duration between events and analyze timing constraints. By plotting state changes along this axis, users can easily evaluate timing relationships such as delays, response times, and deadlines.

3. State or Value Timeline

This component shows how the state or value of the lifeline changes over time. It is typically visualized as a waveform or segmented line aligned with the time axis. Each segment of the timeline corresponds to a particular state or value that the object holds during a specific time interval. The waveform clearly shows transitions, durations, and any stable or fluctuating conditions, making it especially useful for real-time systems or protocols.

4. State Invariants

State invariants are labels or annotations within the timeline that describe the specific condition or status the object is in during a given time segment. For example, a system might be in states like "Idle", "Active", or "Error" at different times. These invariants are essential for understanding what behavior or logic is active during each portion of the timeline, providing semantic meaning to the waveform.

5. Events or Transitions

Events or transitions mark the moments when a lifeline's state or value changes. These are represented as vertical lines or arrows on the diagram, connecting one state segment to another. Events can be triggered by internal processes or external stimuli, and transitions help illustrate cause-effect relationships. They are crucial for tracing how and why an object moves from one state to another.

6. Duration Constraints

Duration constraints define how long an object must remain in a particular state or how long a transition between states is allowed to take. They are usually shown as labeled brackets or notes along the time axis. These constraints are vital in real-time and embedded systems where exact timing can affect system behavior or safety.

7. Time Constraints

Time constraints specify the allowable time window between two events or state changes. They often appear as annotations or notes beside the control flow, such as "within 3 seconds" or "after 1 ms." Time constraints are used to verify whether the timing behavior of the system meets performance or safety requirements, ensuring that operations occur within acceptable limits.

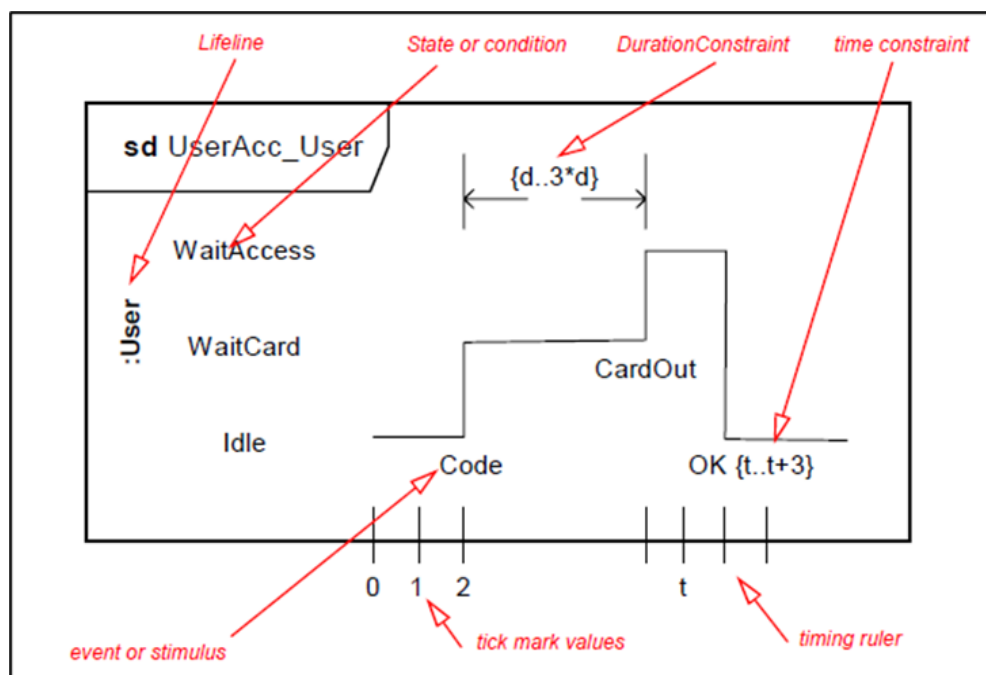


Fig 5.4.22: Key Components of a Timing Diagram

Each of these components plays a critical role in visualizing and analyzing time-dependent behavior in UML Timing Diagrams, making them an essential tool for modeling real-time or embedded systems.

5.4.5.2 Benefits and Drawbacks

Timing diagrams in UML offer significant advantages when modeling systems where the temporal aspect of behavior is critical. These diagrams excel at capturing how objects change state over time, making them particularly valuable in real-time, embedded, or time-sensitive systems. By providing a clear visual representation of state transitions, durations, and time constraints, timing diagrams help developers analyze performance, identify potential timing issues, and ensure system behavior aligns with specified requirements. Their precision and focus on time-based behavior make them an essential tool for both design and validation in technical domains. Benefits are:

1. **Precise Time-Based Modeling:** Timing diagrams provide a detailed and accurate way to model the behavior of systems over time, especially useful in real-time or embedded systems where timing is critical.
2. **Visual Clarity for Temporal Behavior:** They clearly illustrate how an object or system transitions between states, helping developers understand timing relationships, durations, and synchronization between components.
3. **Useful for Performance Analysis:** Timing diagrams help in analyzing and validating system performance, such as response times, delays, and timing constraints, which are crucial for ensuring reliable system behavior.
4. **Ideal for Hardware and Protocol Design:** These diagrams are widely used in hardware engineering, digital design, and communication protocol development, where time-sensitive interactions are central.

Despite their usefulness, timing diagrams in UML have some drawbacks that can limit their applicability in certain scenarios. They primarily focus on time and state changes, which makes them less suitable for modeling complex interactions or intricate control flows. As systems grow in complexity, timing diagrams can become cluttered and difficult to read, especially when multiple lifelines or frequent state transitions are involved. Additionally, the learning curve can be steep for those unfamiliar with real-time systems, and the diagrams may not be easily understood by non-technical stakeholders. Moreover, creating and maintaining timing diagrams often requires specialized tools, which can add to the complexity and overhead of the modeling process. Drawback are:

1. **Limited Scope for Complex Interactions:** Timing diagrams focus primarily on time and state, making them less effective for modeling complex logic, control flow, or interactions between many objects.
2. **Can Become Cluttered:** For systems with multiple lifelines or frequent state changes, the diagram can become dense and hard to read, especially without clear organization or tool support.
3. **Not Beginner-Friendly:** Interpreting timing diagrams can be challenging for those unfamiliar with state machines or real-time system concepts, limiting their accessibility for non-technical stakeholders.
4. **Tool Dependency:** Creating and managing timing diagrams often requires specialized UML tools, as manual drawing can be cumbersome and error-prone, particularly when dealing with precise timing metrics.

5.4.5.3 Example Timing Diagram

This timing diagram example shows a simplified version of the water cycle (Fig: 5.4.23). Since the timing diagram focuses on how long each step takes, not on the system itself, you could think of this as a diagram of the different phases that a water droplet cycles through. If this were a manufacturing process, viewers could easily identify areas for improvement. Timing diagrams are powerful tools for making a system as efficient as possible.

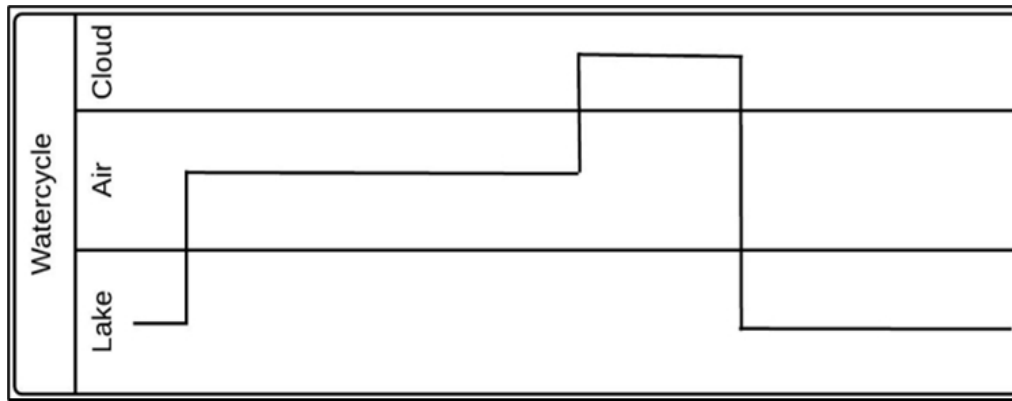


Fig 5.4.23: Timing Diagram Example of the Water Cycle

5.4.6 Comparison

Below table depicts a detailed comparison of Profile Diagrams, Composite Structure Diagrams, Interaction Overview Diagrams, and Timing Diagrams in UML (Table: 5.4.2) based on key aspects:

Table 5.4.2: Comparison of Profile Diagrams, Composite Structure Diagrams, Interaction Overview Diagrams, and Timing Diagrams

Aspect	Profile Diagram	Composite Structure Diagram	Interaction Overview Diagram	Timing Diagram
Purpose	Customize UML with stereotypes, tagged values	Show internal structure of classifiers (like classes)	Visualize control flow between interaction diagrams	Model object behavior over time
Focus	UML extension mechanisms	Internal components and connections	High-level interaction sequencing and control flow	Time constraints, durations, and state transitions
Use Case	Domain specific modeling or creating UML profiles	Modeling parts, ports, connectors of a class or system	Workflow behavior across multiple interactions	Real-time or embedded systems, hardware simulation
Representation	Stereotypes, classes, and constraints	Parts, ports, connectors	Control nodes (initial, decision, merge), interaction references	Lifelines, time axis, state/ value timelines

Detail Level	Meta-modeling level	Structural-level modeling	Behavioral-level modeling	Precise time-based behavior modeling
Strengths	Enables customization and reuse of UML	Clarifies component roles and internal structure	Modular, abstract view of complex interactions	Excellent for time-sensitive behavior and system validation
Weaknesses	Requires understanding of UML meta-model	Can become complex with large systems	Needs supporting sequence/interaction diagrams	Limited scope for interaction logic, can be complex
Best For	Tool creators, domain-specific language designers	System architects, embedded system designers	Business processes, interaction orchestration	Engineers working on real-time, embedded systems
Common Domains	Custom UML tools, modeling standards	Software and hardware architecture	System workflows, business modeling	Automotive, aerospace, IoT, real-time control systems

Recap

- ◆ A Profile Diagram is a UML diagram used to create domain-specific extensions of UML models by applying stereotypes, tagged values, and constraints.
- ◆ It allows the customization of UML elements to fit specific fields like healthcare, aerospace, or banking.
- ◆ The core components of a profile diagram are stereotypes, metaclasses, and extensions, which connect stereotypes to UML base elements.
- ◆ One benefit of using profile diagrams is they provide a lightweight mechanism to adapt UML to domain-specific needs without altering the UML standard.
- ◆ A drawback is that they can become complex to manage and require tool support that fully implements the UML metamodel.
- ◆ Tools that support profile diagrams include Enterprise Architect, MagicDraw, and Papyrus, which allow creating and applying custom profiles.

- ◆ A Composite Structure Diagram describes the internal structure of a class or component and how its parts interact through ports and connectors.
- ◆ This diagram is ideal for representing collaborations between internal elements within a structured classifier.
- ◆ Its main components are structured classifiers, parts, ports, and connectors.
- ◆ A major benefit is its ability to model detailed internal configurations, particularly in component-based and embedded systems.
- ◆ This diagram can be too detailed or complicated for simple applications and may be hard for beginners to grasp.
- ◆ An Interaction Overview Diagram combines elements of sequence and activity diagrams to show high-level control flow across multiple interactions.
- ◆ Key components include interaction occurrences, activity nodes, decision nodes, forks, and control flows.
- ◆ A benefit is that it helps in organizing and visualizing complex interactions in a modular and easy-to-understand manner.
- ◆ Its drawback is the lack of detail within individual interactions, which need to be modeled separately using sequence or communication diagrams.
- ◆ A Timing Diagram is used to represent object behavior and state or value changes over time, making it essential in modeling real-time or time-sensitive systems.
- ◆ Components include lifelines, state/value timelines, messages/events, and time constraints.
- ◆ The main advantage is precise modeling of how objects behave over specific durations, especially useful in embedded systems and hardware design.
- ◆ On the downside, timing diagrams can be visually dense, hard to interpret, and less applicable in non-real-time systems.
- ◆ They are best used when modeling behavior where timing accuracy and synchronization are critical.

Objective Type Questions

1. A Profile Diagram is used to create _____ extensions to UML models.
2. The main components of a profile diagram include stereotypes, _____, and extensions.
3. Composite structure diagrams represent the _____ structure of a class or component.
4. Ports, parts, and connectors are key components of _____ structure diagrams.
5. Timing diagrams are most useful in _____ or embedded systems.
6. In composite structure diagrams, parts are connected using _____.
7. Tools like Enterprise Architect and MagicDraw support the creation of _____ diagrams.
8. What UML diagram extends UML with stereotypes and tagged values?
9. Which diagram shows internal parts and their connections?
10. Which diagram organizes interactions using control flow elements?
11. What component in timing diagrams represents time-dependent object behavior?
12. What component links stereotypes to base UML elements?
13. Which diagram can use forks and joins similar to activity diagrams?
14. Which modeling tool supports UML profile diagrams?
15. What component defines roles within a structured classifier?
16. Which component in a composite structure diagram defines how external interactions occur?
17. In profile diagrams, what do you use to add new properties to existing UML elements?
18. What kind of node in an interaction overview diagram represents a decision point?
19. Name one UML tool other than MagicDraw that supports composite structure diagrams.
20. In timing diagrams, object state changes are shown along a _____ axis.

Answers to Objective Type Questions

1. Domain-Specific
2. Metaclasses
3. Internal
4. Composite
5. Real-time
6. Connectors
7. Profile
8. Profile
9. Composite
10. Interaction
11. Lifeline
12. Extension
13. Interaction
14. MagicDraw
15. Part
16. Port
17. Tagged values
18. Decision node
19. Enterprise Architect
20. Time

Assignments

1. Explain the purpose of a Profile Diagram in UML. Describe its components and discuss how it can be applied in a domain-specific system, including at least one real-world example.
2. Draw and explain a Composite Structure Diagram for a system. Identify its key components and analyze both the advantages and limitations.
3. Explain Interaction Overview Diagrams. Describe the components of an Interaction Overview Diagram and evaluate its usefulness in modeling complex system workflows.
4. Create a Timing Diagram for a real-time system scenario. Explain the components used and discuss the strengths and weaknesses of using Timing Diagrams.

Suggested Reading

1. OMG UML Specification-<https://www.omg.org/spec/UML>
2. Creately UML Resources-[Creately UML Diagram Types](https://creately.com/uml-diagram-types)
3. TutorialsPoint UML Tutorial- <https://www.tutorialspoint.com/uml/>
4. GeeksforGeeks–UMLDiagrams-<https://www.geeksforgeeks.org/types-of-uml-diagrams/>

Reference

1. Martin Fowler (2003), “*ML Distilled: A Brief Guide to the Standard Object Modeling Language*” (3rd Edition.), published by Addison-Wesley.
2. Grady Booch, James Rumbaugh, Ivar Jacobson (2005), “*The Unified Modeling Language User Guide*” (2nd Edition), published by Addison-Wesley.
3. Alan Dennis, Barbara Haley Wixom, David Tegarden (2005), “*Systems Analysis and Design with UML Version 2.0*”, published by Wiley.
4. Larman, C. (2004), “*Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development*” (3rd Edition.), published by Pearson Education.

BLOCK 6

DevOps





Introduction to DevOps

Learning Outcomes

The learner will be able to :

- ◆ familiarize the concept of DevOps
- ◆ identify the challenges in traditional software development
- ◆ explore the benefits of DevOps
- ◆ familiarize with the key components and tools of DevOps

Prerequisites

The world of software development wasn't always as fast and efficient as it is today. A few decades ago, developers and operations teams were like two separate worlds. Developers were focused on writing code and pushing out new features, while operations were solely concerned with maintaining infrastructure and ensuring stability. This split created a lot of challenges—releases took forever, problems went unnoticed until it was too late, and the constant back-and-forth between teams made everything slower. It felt like every time developers pushed new updates, the operations team had to scramble to keep everything running smoothly.

But as technology progressed, so did the demands for faster, more reliable software. Businesses began to see the need for change: software couldn't afford to be slow and clunky anymore. Teams started to realize that collaboration between developers and operations was the key to solving these issues. Instead of working in silos, these two worlds had to come together to create something more seamless, faster, and reliable. This realization sparked a movement that would forever change how software is built, tested, and delivered.

This movement became what we now know as DevOps—a way of thinking, working, and creating together to make software development faster, smarter, and more efficient. The transformation was groundbreaking, making it possible for companies to release software not just quickly, but reliably and with fewer problems. Curious about how this shift took place? Let's dive deeper into the world of DevOps and explore how it has reshaped the software industry forever!

Keywords

DevOps, Collaboration, Continuous Integration, Continuous Delivery, Terraform, Jenkins, Spinnaker, Selenium

Discussion

6.1.1 What is DevOps?

DevOps is a blend of Development and Operations, the two essential teams in software delivery. It is a culture, a process and a mindset that focuses on building, testing, and releasing software faster, more reliably, and with better quality.

In a traditional software development environment, developers and IT operations teams often work separately. Developers write the code, while operations teams are responsible for deploying and maintaining the infrastructure. This division can lead to communication gaps, delays, and misunderstandings.

DevOps breaks down these silos. Developers and operations work together from the start to the end of the lifecycle of an application. They collaborate on everything from code creation to deployment and monitoring. For example, developers write the code, and they collaborate with operations to set up the correct infrastructure (e.g., setting up servers, cloud resources) to deploy the application effectively.

For example, If a developer writes code for a new feature, they might work closely with the operations team to ensure that the infrastructure is properly set up (e.g., server configurations, databases) for smooth deployment. They may even use shared tools, like Docker, to ensure consistency across environments (development, testing, and production).

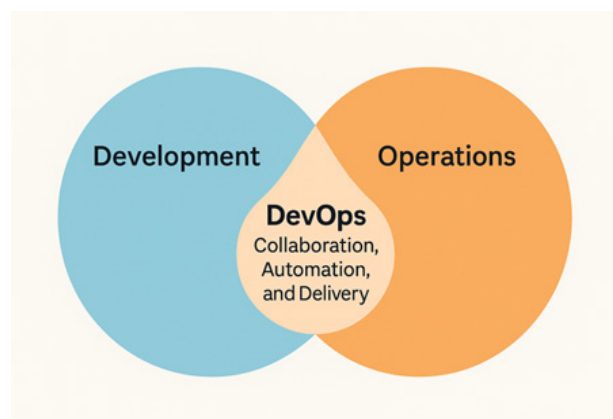


Fig 6.1.1 DevOps – Collaboration, Automation, and Delivery

DevOps = Fast, Reliable, Continuous Software Delivery

The term "DevOps" was first coined in 2009 by Patrick Debois, a Belgian consultant. He was frustrated with the constant conflict between developers (who wanted speed) and operations teams (who wanted stability).

He organized a conference called **"DevOpsDays"** in Ghent, Belgium, and that small gathering sparked a global movement!

6.1.2 Why do we need DevOps?

In order to understand the need of DevOps it is necessary to know The "Wall of Confusion" in Traditional Software Development. In traditional software development, there are two teams: developers and operations (Ops).

- ◆ Developers want to release new features and updates quickly to keep up with user demands and improve the product.
- ◆ Operations want everything to stay stable, meaning they don't want updates to cause bugs or crashes.

This creates a **"wall"** between the teams because developers want speed, while operations want stability. As a result, there's often a lot of miscommunication, and things can slow down.

How DevOps Fixes This....

DevOps helps by bringing developers and operations together to work as one team. The goal is to deliver high-quality software quickly while keeping things stable. See how it fixes the problem.

Working Together: Developers and operations teams collaborate closely from the start, so both teams understand each other's goals.

Shared Responsibility: Both teams are responsible for making sure the software works well and gets to users quickly.

Automation: DevOps uses automated tools to speed up the process of building, testing, and deploying software, so things are faster and more reliable.

DevOps also solves some common problems in traditional software development like:

1. Slow Deployments

Deploying new features or fixes can take a long time because it's often a manual process. *DevOps Solution:* Automation speeds up the process. Code is automatically built, tested, and deployed without much human intervention, making releases faster.

2. Manual Errors

People sometimes make mistakes when doing things by hand, like configuring servers or deploying code.



DevOps Solution: DevOps uses automated tools that handle tasks without human error, making things more consistent.

3. Downtime During Updates

When updates are made, they can cause the application to go offline, disrupting users.

DevOps Solution: With DevOps, updates can be done in a way that avoids downtime. For example, they use smart techniques to update parts of the system without affecting the entire app.

4. Poor Communication

Developers and operations teams often don't communicate well, which can lead to misunderstandings and delays.

DevOps Solution: Collaboration is key. The teams communicate regularly and use shared tools to stay on the same page, ensuring everyone understands what needs to be done.

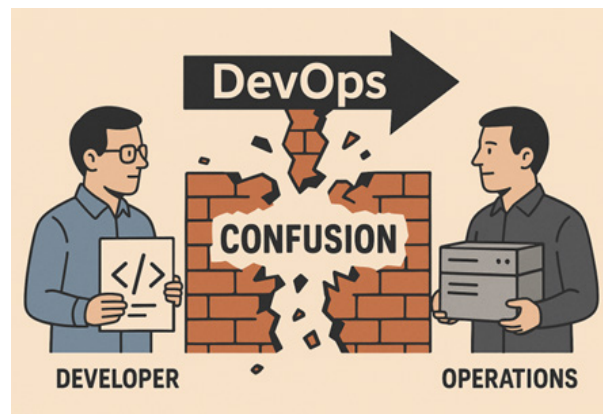


Fig.6.1.2 DevOps breaking the Confusion Wall

6.1.3 Benefits of DevOps

Quantifiable Gains:

- ◆ increase in team collaboration
- ◆ improvement in app quality
- ◆ reduction in development time
- ◆ rise in deployment frequency

Business Impact:

- ◆ Faster Time-to-Market
- ◆ Reduced Costs
- ◆ Happier Customers

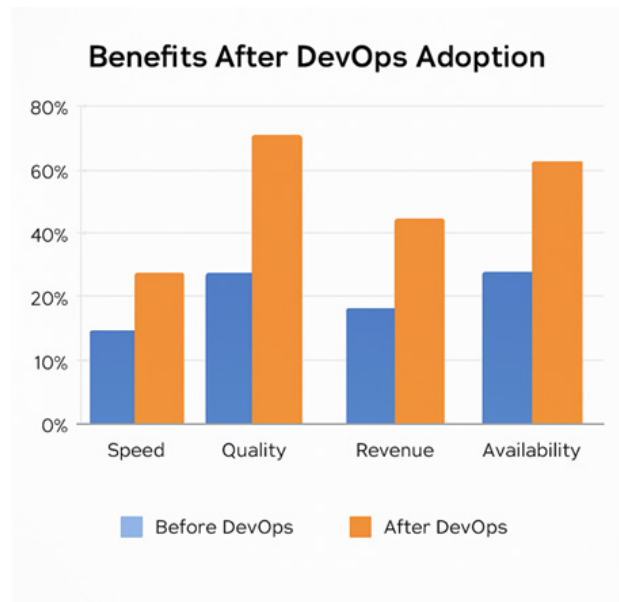


Fig 6.1.3 Bar graph showing benefits after DevOps adoption

6.1.4 Happiest Mind Approach : The 7 C's of DevOps

Happiest Minds follows the agile belief: “People come first, then process, then tools.”

They believe that with the right people, it's possible to build the right processes, which then leads to choosing the right tools to achieve the desired results. The 7 C's of DevOps are :

1. Communication

Good DevOps starts with clear and open communication between developers, testers, operations, and other teams. Everyone should know what's happening, what's expected, and what needs fixing.

2. Collaboration

All teams work together instead of doing their jobs separately. Developers, testers, and operations must share responsibilities and support each other.

Example: A developer checks with the testing team before adding a new feature to ensure it won't break anything.

3. Controlled Process

There should be a clear and consistent way of doing things like how to develop, test, and release software. This avoids confusion and mistakes.

Example: Every code change follows a set process: code → review → test → deploy.

4. Continuous Integration (CI)

Developers keep adding small bits of code regularly, and the system automatically checks if everything still works. This helps catch errors early.

Example: Every time a developer pushes code, an automatic test runs to make sure

nothing is broken.

5. Continuous Deployment (CD)

Once the code is tested, it is automatically released to production without delay. This helps in fast and safe updates.

Example: A new feature is automatically made live after passing all tests—no need for manual deployment.

6. Continuous Testing

Testing happens all the time using automated tools. This ensures the software is always in good shape.

Example: Automated tests check if the login page still works every time someone changes the code.

7. Continuous Monitoring

The system is watched all the time to make sure it runs smoothly. Alerts are sent if something goes wrong.

Example: A monitoring tool tracks how much memory a server is using and warns the team before it crashes.

6.1.5 DevOps Tools and Automation

Automation is the heart of DevOps. Here are some tools used in various stages of DevOps.

Stage	Tools
Provisioning	Terraform, Ansible, Chef, Puppet, AWS CloudFormation
CI	Jenkins, GitLab CI, CircleCI, Travis CI, Bamboo
CD	Spinnaker, ArgoCD, GitHub Actions, Flux, Octopus Deploy
Testing	Selenium, JUnit, TestNG, PyTest, Postman, Cucumber
Monitoring	Prometheus, Grafana, Nagios, Datadog, New Relic, ELK Stack (Elasticsearch, Logstash, Kibana)

6.1.6 Who's using DevOps?

Top tech companies using DevOps:

- ◆ Amazon – Deploys code every 11 seconds, leading the way in automation.

- ◆ Netflix – Relies on DevOps for speed, uptime, and seamless user experience.
- ◆ Google – Combines Site Reliability Engineering (SRE) with DevOps practices.
- ◆ Facebook – Uses DevOps to release updates daily without interrupting service.
- ◆ Spotify – Leverages DevOps for microservices, rapid delivery, and team autonomy.
- ◆ Microsoft – Applies DevOps for Azure deployments and cloud service efficiency.
- ◆ Etsy – Transformed from long deployment cycles to multiple deployments per day using DevOps.

Benefits They Achieved:

- ◆ Faster Recovery from system failures
- ◆ Shorter Time-to-Market for new features
- ◆ Higher Customer Satisfaction
- ◆ Improved Software Quality
- ◆ Increased Deployment Frequency
- ◆ Reduced Human Errors through automation
- ◆ Stronger Collaboration between teams

Recap

- ◆ What is DevOps
 - Blend of Development and Operations
 - Culture + Process + Mindset
 - Builds, tests, releases software faster and reliably
 - Breaks down dev-ops silos
 - Collaboration across lifecycle
 - Uses tools like Docker for consistency
 - Coined in 2009 by Patrick Debois
 - Originated from “DevOpsDays” in Belgium

- ◆ Why Do We Need DevOps
 - Solves the “Wall of Confusion”
 - Devs want speed, Ops want stability → conflict
 - DevOps = shared goals + smooth delivery
- ◆ Problems DevOps Solves:
 - Slow Deployments → Fixed with automation
 - Manual Errors → Fixed with reliable tools
 - Downtime During Update → Fixed with smart deployments
 - Poor Communication → Fixed with collaboration, shared tools
- ◆ Benefits of DevOps
 - increase in team collaboration
 - better app quality
 - faster development
 - more frequent deployments
- ◆ Business Impact:
 - Faster time-to-market
 - Lower costs
 - Happier customers
- ◆ Happiest Minds – The 7 C’s of DevOps
 - Communication – Clear updates between all teams
 - Collaboration – Teams work together, not in silos
 - Controlled Process – Standard workflows avoid errors
 - Continuous Integration – Code added often + auto-checked
 - Continuous Deployment – Auto-release post-testing
 - Continuous Testing – Auto-tests run continuously
 - Continuous Monitoring – Constant system checks + alerts

Objective Type Questions

1. What culture blends Development and Operations?
2. Who coined the term "DevOps"?
3. In which year was DevOps coined?
4. Which country hosted the first DevOpsDays event?
5. What tool ensures consistent environments using containers?
6. What is the biggest obstacle between developers and operations teams?
7. Which process does DevOps use to avoid manual deployment delays?
8. What type of errors does automation help eliminate?
9. What is used to monitor servers and alert teams?
10. What is the term for small, frequent code changes tested automatically?
11. What is the term for automatic release of tested code into production?
12. What is the last 'C' in the 7 C's of DevOps?
13. Which DevOps tool is used for provisioning and infrastructure as code?
14. Name a CI tool used in DevOps.
15. Name a CD tool used in DevOps.
16. Name a testing tool used in DevOps.
17. Name a monitoring tool used in DevOps.
18. Which company deploys code every 11 seconds?
19. Which company uses DevOps to support streaming services?
20. Which search giant uses Site Reliability Engineering with DevOps?
21. Which company releases updates daily using DevOps?
22. Which music app uses DevOps for microservices and autonomy?
23. Which cloud platform uses DevOps for Azure?
24. Which e-commerce company shifted to multiple daily deployments using DevOps?

Answers to Objective Type Questions

1. DevOps
2. Debois
3. 2009
4. Belgium
5. Docker
6. Wall
7. Automation
8. Manual
9. Monitoring
10. CI
11. CD
12. Monitoring
13. Terraform
14. Jenkins
15. ArgoCD
16. Selenium
17. Prometheus
18. Amazon
19. Netflix
20. Google
21. Facebook
22. Spotify
23. Microsoft
24. Etsy

Assignments

1. Explain the "Wall of Confusion" in traditional software development. How does DevOps overcome it?
2. List and describe the 7 C's of DevOps as followed by Happiest Minds. Provide an example for any two.
3. Identify and categorize DevOps tools used in different stages such as Provisioning, CI, CD, Testing, and Monitoring.
4. Choose any one company that uses DevOps (e.g., Amazon, Netflix, or Google) and explain how DevOps benefits their software delivery process.

Suggested Reading

1. <https://everythingdevops.dev/a-brief-history-of-devops-and-its-impact-on-software-development/>
2. DevOps: A Software Architect's Perspective." O'Reilly Media, 2019
3. Site Reliability Engineering: How Google Runs Production Systems" by Niall Richard Murphy, Betsy Beyer, Chris Jones, and Jennifer Petoff

Reference

1. The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win" by Gene Kim, Kevin Behr, and George Spafford
2. The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations" by Gene Kim, Patrick Debois, John Willis, and Jez Humble
3. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations" by Nicole Forsgren, Jez Humble, and Gene Kim



DevOps Life Cycle

Learning Outcomes

The learner will be able to :

- ◆ familiarize the core philosophy of DevOps
- ◆ describe the CALMS model
- ◆ recognize the importance of collaboration and automation
- ◆ identify and explain the stages of the DevOps lifecycle
- ◆ understand the continuous nature of DevOps

Prerequisites

Have you ever wondered how companies like Netflix, Amazon, or Google manage to release new features almost every week without causing disruptions? Behind that kind of speed and reliability is a powerful approach to building and running software called DevOps. It's not just about writing code or maintaining systems. It's about creating a smooth flow where ideas quickly become working products. At the core of this approach is the CALMS model, a simple yet meaningful way to understand what makes DevOps work so well.

As you explore the DevOps lifecycle, you will start to see how modern teams plan effectively, write better code, and deliver updates faster, all while maintaining stability. This is more than just a set of steps—it's a mindset that helps teams solve real problems, learn from experience, and work closely together to create software that truly matters. Learning this approach means preparing yourself to be part of how the tech world works today.

Get ready to see how everything connects and how you can be part of something that transforms ideas into reality with confidence and speed.

Keywords

CALMS model, DevOps lifecycle, continuous loop,

Discussion

6.2.1 How does Devops Work?

DevOps is a modern software development approach that brings together development (Dev) and operations (Ops) teams to work collaboratively throughout the software lifecycle. Unlike traditional models where coding and deployment were handled by separate teams, DevOps integrates these functions to accelerate delivery, improve quality, and respond faster to user needs.

But how exactly does DevOps achieve this?

One popular framework that explains DevOps in action is the CALMS model, which stands for Culture, Automation, Lean, Measurement, and Sharing. These five pillars guide organizations in implementing DevOps practices effectively and sustainably.

6.2.1.1 The CALMS model: In Detail

CALMS is a simple way to understand how DevOps works in a company and how to check if it's working well. The word **CALMS** stands for:

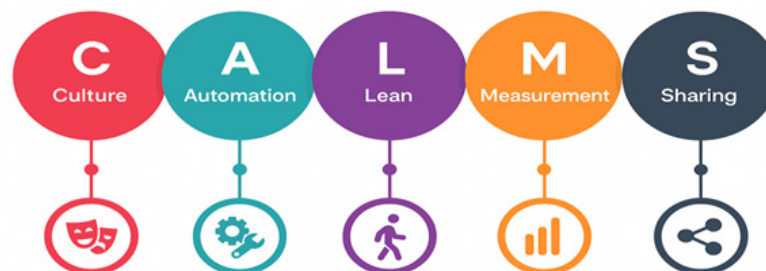


Fig.6.2.1 CALMS

This idea was introduced by Jez Humble, co-author of *The DevOps Handbook*. Let's break down what each part means in a clear way.

C – Culture: Work Together as One Team

DevOps is not just about using tools. It's about changing how people work together. Instead of keeping developers and operations in separate groups (called “silos”), DevOps brings everyone together.

In a good DevOps culture:

- ◆ Teams share goals and plans.
- ◆ Developers, testers, designers, operations – all work side by side.
- ◆ Meetings include everyone – standups, planning, and reviews.

Even if your company isn't ready to form one big team right away, you can start small. Invite operations to developer meetings and vice versa. It helps people understand each other's work and problems.

Real DevOps culture shows when everyone helps during an issue, and no one blames others. They just fix it together.

A – Automation: Let Computers Do the Repetitive Work

Doing the same thing manually over and over? That's where automation helps!

DevOps encourages teams to automate various tasks such as testing new code, deploying applications, and setting up servers and infrastructure.

This saves time and reduces mistakes. It also helps different teams trust each other's work because everything follows a reliable process.

Some useful automation concepts:

Continuous Delivery (CD) – Code is automatically tested and sent to production.

Infrastructure as Code (IaC) – Servers and settings are written like software code.

L – Lean: Small Steps, Fast Feedback

“Lean” means: Don't waste time. Keep improving. Be ready to fail and learn from it.

DevOps teams:

- ◆ Deliver features in small, regular batches
- ◆ Hold retrospective meetings to learn what went wrong
- ◆ Build systems that can bounce back from failure quickly

ie, DevOps teams work by releasing new features in small, steady steps. They regularly meet to look back and understand what didn't go well, so they can do better next time. Their systems are built to handle problems and recover quickly. Instead of waiting a long time to create a perfect product, they focus on giving users something helpful now and then keep making it better over time.

M – Measurement: Use Data to Improve

If you can't measure it, you can't improve it!

DevOps teams keep watch over four main things:

1. How fast they can get new features ready for users?
2. When problems happen in their systems?
3. How quickly they can fix things when they go wrong?
4. How people are actually using their products?

These numbers help teams make better decisions and prove that their work is making a difference.

Data is not just for developers. Share it with marketing, sales, support and everyone!

S – Sharing: Learning is Better Together

Sharing is at the heart of DevOps. That means : Sharing success and failure, Sharing tools and processes and Sharing feedback and ideas. When developers help with operations, and operators help with development, everyone understands the big picture. You might hear the phrase:

“You build it, you run it.”

This means the same team that writes the code also helps run it in real life — so they care about how well it works!

Some companies even rotate team members to handle support for live issues. This helps them learn more and stay connected to real users’ needs.

6.2.2 DevOps Life Cycle

Let's explore the DevOps lifecycle in a general way, using different examples to illustrate each stage. Remember, DevOps is about collaboration, automation, and continuous improvement throughout the software development process. Here are the common stages of the DevOps lifecycle:

1. Plan

This is the initial stage where the team defines the project goals, scope, and requirements. They decide what needs to be built or improved.

Example: For a new e-commerce website, the planning phase would involve deciding on the features (product catalog, shopping cart, user accounts, payment gateway integration), the technologies to be used, and the overall project timeline.

2. Code

Developers write the source code for the application based on the plan. They use version control systems to manage changes and collaborate.

Example: For the e-commerce website, developers would write code in languages like HTML, CSS, JavaScript for the front-end and Python, Java, or Node.js for the back-end logic. They would use Git for version control, allowing multiple developers to work on the codebase simultaneously and track changes.

3. Build

The code is compiled, packaged, and prepared for testing and deployment. This often involves using build automation tools.

Example: Using tools like Maven (for Java), Gradle (for Java and Android), or Webpack (for JavaScript), the developers would compile the source code, bundle dependencies, and create deployable artifacts (like JAR files or bundled JavaScript files) for the e-commerce website.

4. Test

Rigorous testing is performed at various levels to ensure the quality, functionality, and security of the application. This includes unit tests, integration tests, system tests, and user acceptance tests.

Example: Automated testing frameworks like JUnit (for Java), Selenium (for web applications), or Cypress (for end-to-end testing) would be used to automatically run tests against the e-commerce website. This ensures that features like adding items to the cart, processing payments, and user registration work correctly.

5. Release

The tested and approved software is prepared for deployment to the production environment. This might involve packaging the application in a specific format for the target environment.

Example: The build artifacts of the e-commerce website would be packaged into Docker containers for easy deployment and consistency across different environments.

6. Deploy

The released software is deployed to the production servers or infrastructure where it will be accessible to users. Automation tools are often used for this process.

Example: Using container orchestration tools like Kubernetes, the Docker containers holding the e-commerce website would be deployed to a cluster of servers. Infrastructure-as-Code (IaC) tools like Terraform or Ansible might be used to provision and configure the necessary infrastructure.

7. Operate

The application is running in the production environment, and the operations team ensures its stability, performance, and availability. This includes managing servers, databases, and network infrastructure.

Example: The operations team would monitor the e-commerce website's servers, databases, and network to ensure it's running smoothly and can handle user traffic. They would also perform tasks like server maintenance, backups, and security updates.

8. Monitor

The performance and health of the application and infrastructure are continuously monitored. This helps identify issues, understand user behavior, and gather data for

future improvements.

Example: Monitoring tools like Prometheus, Grafana, or Datadog would be used to track key metrics of the e-commerce website, such as website response time, error rates, CPU usage, and database performance. Logs would be analyzed to identify potential problems or areas for optimization. User feedback and reviews would also be monitored.

The Continuous Loop:

The DevOps lifecycle is an iterative process. The insights gained from the "Operate" and "Monitor" stages feed back into the "Plan" stage, driving the next cycle of development and improvement. For instance, if monitoring reveals slow page load times during peak hours, the team will plan to optimize the website's performance in the next iteration.

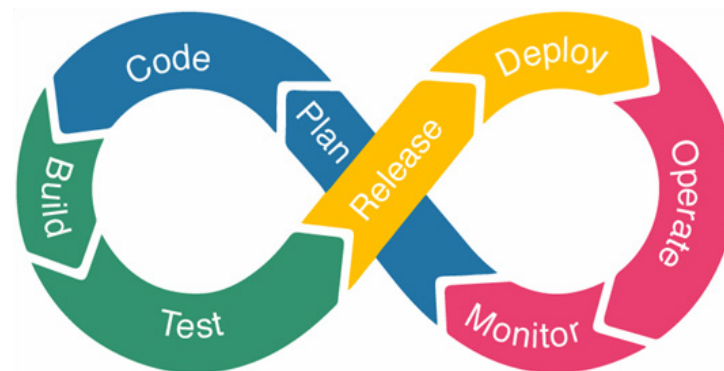


Fig.6.2.2 DevOps Lifecycle

Recap

- ◆ CALMS Model Overview
 - Introduced by Jez Humble
 - Stands for Culture, Automation, Lean, Measurement, Sharing
 - Framework to assess DevOps maturity
- ◆ C – Culture
 - One team mindset
 - Shared goals and responsibilities
 - Cross-functional collaboration
 - No blame during issues
 - Inclusive meetings and communication

- ◆ A – Automation
 - Reduces manual effort
 - Automates testing, deployment, server setup
 - Ensures consistency
 - Builds trust across teams
 - Includes CD and IaC
- ◆ L – Lean
 - Focus on small, frequent releases
 - Continuous improvement
 - Retrospective meetings
 - Fast recovery from failures
 - Deliver value early and often
- ◆ M – Measurement
 - Metrics for speed, reliability, and usage
 - Data-driven decisions
 - Monitors feature delivery and system issues
 - Shares data across departments
- ◆ S – Sharing
 - Open knowledge flow
 - Shared successes and failures
 - Cross-team support
 - Developers handle operations too
 - Team rotations for real-world learning
- ◆ DevOps Lifecycle Stages
 - **Plan:** Define goals, scope, requirements
Code: Develop source code, use version control
 - **Build:** Compile code, create deployable packages
 - **Test:** Perform unit, integration, and system testing

- **Release:** Approve and prepare software for production
 - **Deploy:** Push code to live servers, use automation
 - **Operate:** Maintain system performance and availability
 - **Monitor:** Track metrics, analyze logs, gather feedback
- ◆ Key Tools Mentioned
- Git – version control
 - Maven, Gradle – build tools
 - JUnit, Selenium, Cypress – testing frameworks
 - Docker – containerization
 - Kubernetes – container orchestration
 - Terraform, Ansible – infrastructure as code
 - Prometheus, Grafana, Datadog – monitoring tools

Objective Type Questions

1. What approach combines development and operations teams?
2. Which model describes Culture, Automation, Lean, Measurement, and Sharing?
3. Who introduced the CALMS model?
4. What word in CALMS represents teamwork and collaboration?
5. What word in CALMS focuses on reducing manual work?
6. What word in CALMS is about avoiding waste and delivering in small batches?
7. What word in CALMS involves using data to make decisions?
8. What word in CALMS is about openness and feedback?
9. What is the first stage of the DevOps lifecycle?
10. What is the second stage where developers write code?
11. What is the third stage where the code is compiled and packaged?
12. What stage involves running automated checks for bugs?

13. What stage prepares the software for production use?
14. What stage involves pushing code to production?
15. What stage involves managing systems in production?
16. What stage involves tracking performance and health of systems?
17. What tool is commonly used for version control?
18. What tool is used to build Java projects?
19. What framework is used to test Java code?
20. What container tool is mentioned for packaging applications?
21. What orchestration tool is mentioned for deploying containers?
22. What monitoring tool starts with the letter “P”?
23. What IaC tool is used to set up infrastructure and starts with “T”?

Answers to Objective Type Questions

1. DevOps
2. CALMS
3. Jez
4. Culture
5. Automation
6. Lean
7. Measurement
8. Sharing
9. Plan
10. Code
11. Build
12. Test
13. Release
14. Deploy
15. Operate
16. Monitor
17. Git
18. Maven

19. JUnit
20. Docker
21. Kubernetes
22. Prometheus
23. Terraform

Assignments

1. List and briefly explain the five components of the CALMS model used in DevOps.
2. Outline the key stages of the DevOps lifecycle. Provide one real-world example for any three stages.
3. What role does automation play in the DevOps process? Mention any two tools commonly used for automation.
4. How does the 'Sharing' principle in the CALMS model contribute to team performance and learning?

Suggested Reading

1. DevOps: *A Software Architect's Perspective*." O'Reilly Media, 2019
2. Site Reliability Engineering: *How Google Runs Production Systems*" by Niall Richard Murphy, Betsy Beyer, Chris Jones, and Jennifer Petoff

Reference

1. The Phoenix Project: *A Novel About IT, DevOps, and Helping Your Business Win*" by Gene Kim, Kevin Behr, and George Spafford
2. The DevOps Handbook: *How to Create World-Class Agility, Reliability, & Security in Technology Organizations*" by Gene Kim, Patrick Debois, John Willis, and Jez Humble
3. Accelerate: The Science of Lean Software and DevOps: *Building and Scaling High Performing Technology Organizations*" by Nicole Forsgren, Jez Humble, and Gene Kim



Agile Manifesto and Scrum Framework

Learning Outcomes

The learner will be able to :

- ◆ understand the need for Agile
- ◆ explain the origin and core values of the Agile Manifesto
- ◆ describe the key roles in a Scrum team
- ◆ familiarize the Scrum process flow

Prerequisites

Have you ever tried planning a group trip with your friends? You set a date, decide the destination, make a checklist, and everything looks perfect on paper. But then someone changes their mind about the location, another person cannot make the dates, and suddenly the perfect plan falls apart. What started as a well-organized trip turned into confusion because plans kept changing and no one adjusted fast enough. Now, imagine trying to build a software project the same way, with a fixed plan but constantly shifting needs. Sounds frustrating, right?

Now picture this instead. What if your group started small, just choosing a weekend and a rough idea of where to go, then made decisions one step at a time, checking in with everyone along the way? If someone had a better idea or ran into a problem, you could tweak the plan without ruining everything. That kind of approach does not just work for trips, it works for big and complicated projects too. And that is where things start to get really interesting.

Let us step into a world where change is not a threat, but a natural part of the plan. Where teams build things piece by piece, learning and improving as they go. Curious? You should be. Let us find out what makes this approach so powerful.

Keywords

Agile, Scrum, Sprint, product backlog, scrum master

Discussion

6.3.1 Introduction

Have you ever worked on a group project where everyone had different ideas, timelines kept shifting, and in the end, the final product didn't match what people actually wanted? That's a common scenario in traditional software development. Projects would take months or even years to complete, and by the time they were done, the needs of the users might have changed or even worse, the final product wouldn't meet those needs at all.

Agile was born to solve this problem.

Agile is a way of working that helps teams deliver small, useful parts of a product quickly, gather feedback, and continuously improve. It's like building a house room by room. After completing each room, you check with the owner to see if it meets their expectations. If it doesn't, you can make changes before moving on. This approach is far better than building the entire house at once and only then asking if the owner likes it. Agile allows teams to stay flexible, adapt to changes, and deliver real value to users—step by step.

6.3.2 Agile Manifesto

In February 2001, seventeen software developers gathered at The Lodge at Snowbird ski resort in Utah to find common ground and address frustrations with traditional, heavyweight software development methods. These professionals, who were leaders in methodologies like Extreme Programming, SCRUM, DSDM, Crystal, and others, shared a desire for a better, more flexible way to build software. Despite being from different backgrounds and sometimes even competitors, they managed to align on a shared set of principles - what became the **Manifesto for Agile Software Development**. This marked the birth of the *Agile Alliance*.

The meeting was not just about processes, but about values, trust, collaboration, adaptability, and delivering value to the customer. Agile wasn't intended to discard methodology but to restore balance: focusing on people over rigid processes, working software over comprehensive documentation, and customer collaboration over contract negotiation. While it challenged traditional corporate structures and mindsets, Agile offered a refreshing alternative that emphasized continuous improvement and meaningful work environments.

The idea of "light" or "lightweight" methods had been circulating prior to Snowbird, but the term never sat well with the group. With humor, thoughtful debate, and a shared

sense of purpose, they settled on the term *Agile*, capturing the spirit of flexibility and responsiveness. The Snowbird meeting laid the foundation for a movement that continues to influence software development practices worldwide.

In a nutshell, the Agile Manifesto is built on four core values:

- ◆ Individuals and interactions over processes and tools
- ◆ Working software over comprehensive documentation
- ◆ Customer collaboration over contract negotiation
- ◆ Responding to change over following a fixed plan

This doesn't mean the things on the right aren't important—it just means the things on the left matter more in Agile projects.

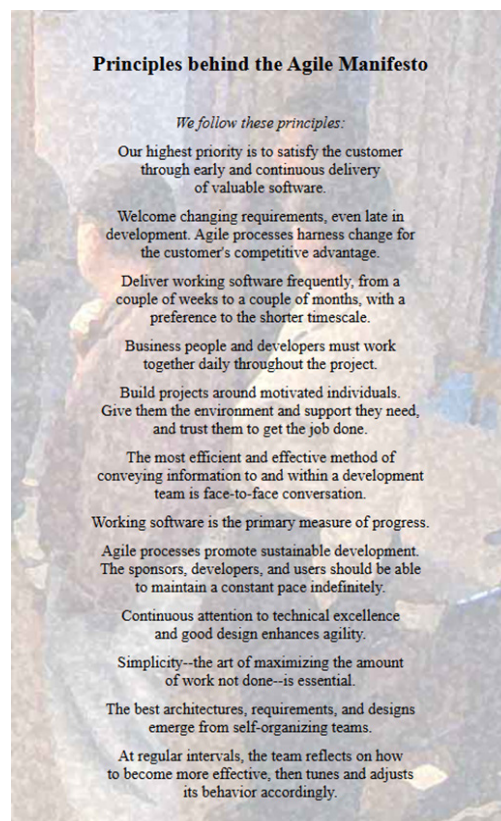


Fig.6.3.1 Principles of Agile Manifesto

6.3.2.1 The 12 Principles of Agile

The Agile Manifesto thus outlined the following 12 core principles for the development process:

1. Satisfy customers through early and continuous delivery of valuable work.
2. Break big work down into smaller tasks that can be completed quickly.
3. Recognize that the best work emerges from self-organized teams.

4. Provide motivated individuals with the environment and support they need and trust them to get the job done.
5. Create processes that promote sustainable efforts.
6. Maintain a constant pace for completed work.
7. Welcome changing requirements, even late in a project.
8. Assemble the project team and business owners daily throughout the project.
9. Have the team reflect at regular intervals on how to become more effective, then tune and adjust behavior accordingly.
10. Measure progress by the amount of completed work.
11. Continually seek excellence.
12. Harness changes for a competitive advantage.

6.3.2.2 Understanding Agile Requirements

In traditional software development methods, all the requirements were gathered and documented in detail right at the beginning of the project. This approach assumed that customers knew exactly what they needed from the start and that those needs wouldn't change. But in reality, that's rarely the case. As the project progresses, customers often realize new needs, priorities shift, or market conditions change. The problem with traditional methods is that they don't easily accommodate these changes.

Agile takes a different approach. It understands that requirements can and will evolve. Instead of locking everything down early, Agile welcomes change, even late in the process. Agile teams gather requirements in small chunks and continuously update them throughout the project. This makes the development process more flexible, adaptive, and responsive to the customer's true needs.

One of the key tools used in Agile for managing requirements is the **user story**. A user story is a short, simple sentence that describes what the user wants and why. It usually follows this format:

"As a [user], I want [something] so that [reason]."

For example:

"As a student, I want to check my attendance online so that I can track my progress."

User stories are easy to understand because they are written in the language of the customer, not in technical jargon. This helps everyone involved. i.e, developers, testers, and clients stay on the same page. The story is not the full requirement but a starting point for further discussion. Teams and customers collaborate to clarify details, identify priorities, and decide what needs to be done.

As the project goes on, Agile teams revisit and update these user stories. They might add new ones, remove outdated ones, or change existing ones based on feedback or new business goals. This ongoing process ensures that the product remains relevant and useful and truly solves the problems the customer cares about.

In short, Agile doesn't try to predict everything at the beginning. Instead, it focuses on building the right thing, one small piece at a time, with constant input from the people who will actually use it.

6.3.3 Types of Agile Methodologies

The most widely used Agile methods include the following:

- ◆ Scrum.
- ◆ Lean software development.
- ◆ Extreme programming.
- ◆ Crystal.
- ◆ Kanban.
- ◆ Dynamic systems development method.
- ◆ Feature-driven development.

In this unit, we are discussing the Scrum method.

6.3.3.1 Scrum: An Agile Framework

Agile is a big umbrella, and Scrum is one of the most popular ways to put Agile into practice.

Think of Scrum as a way to organize teamwork when using Agile ideas.

Scrum is like playing a sport: it has rules, roles, and a way of playing. The goal is to keep moving the ball (the product) forward a little bit at a time, checking in regularly, and adjusting when needed.

Key Roles in Scrum

1. Product Owner – Think of this person as the curator of a museum. They decide which exhibits (features) should be showcased, in what order, and how they will benefit the visitors (users). They carefully select and prioritize what brings the most value, always keeping the audience in mind. Like a curator, the Product Owner connects the creators (developers) with the needs and expectations of the public (customers). They represent the customer's voice.
2. Scrum Master – Think of this person as a coach of a sports team. They don't play the game themselves, but they make sure the team knows the rules, stays focused, and works well together. They remove any blockers (like a referee calling out a foul) and help the team improve their game every sprint. Their goal is to create the best environment for the team to succeed. They make sure the team follows Scrum rules and help remove any obstacles.
3. Development Team – These are the players. They design, build, test, and deliver the product. They are self-organizing, meaning they decide how to do the work.

6.3.3.2 Scrum Process Flow: How Scrum Works

Let's imagine the team is building an online library system. Here's how Scrum would help at each step:

1. Product Backlog

This is like a to-do list for the whole project, created and maintained by the Product Owner. It's filled with **user stories**—simple descriptions of what different users want from the system.

Examples:

- ◆ “As a user, I want to search for books so I can find what I need quickly.”
- ◆ “As a librarian, I want to add new books so the catalog stays updated.”

The backlog is flexible. It can grow and change as new ideas come in or priorities shift. This makes it easier to respond to customer feedback or changing business needs.

2. Sprint Planning

Before starting the actual work, the team holds a **Sprint Planning meeting**. They look at the top items in the product backlog and pick a few to work on in the upcoming sprint (which usually lasts **2 to 4 weeks**).

They also plan *how* to get things done - breaking down the user stories into smaller tasks and estimating the effort needed. The team commits to a realistic amount of work for the sprint.

3. Sprint (Work Cycle)

This is the heart of Scrum—the focused period where the team works on the selected tasks. Each day, they hold a short **Daily Stand-up** (around 15 minutes), where every team member answers three quick questions:

- ◆ What did I do yesterday?
- ◆ What will I do today?
- ◆ Is there anything blocking my progress?
- ◆ This keeps everyone on the same page and helps identify issues early.

4. Sprint Review

At the end of the sprint, it's time to show off what's been built. The team presents their work to the Product Owner and other stakeholders in a Sprint Review meeting.

For instance, they might demonstrate the working “Search for Books” feature or show how librarians can now add new books. This feedback session helps guide future decisions and improvements.

5. Sprint Retrospective

After the review, the team meets privately for the Sprint Retrospective. This is a moment for honest reflection:

- ◆ What went well?
- ◆ What didn't go so well?
- ◆ What can we improve for the next sprint?

It's all about learning and getting better, one sprint at a time. Then the cycle begins again! As each sprint concludes, the online library system evolves, becoming smarter, more useful, and more polished. This continuous improvement ensures the system remains closely aligned with actual user needs.

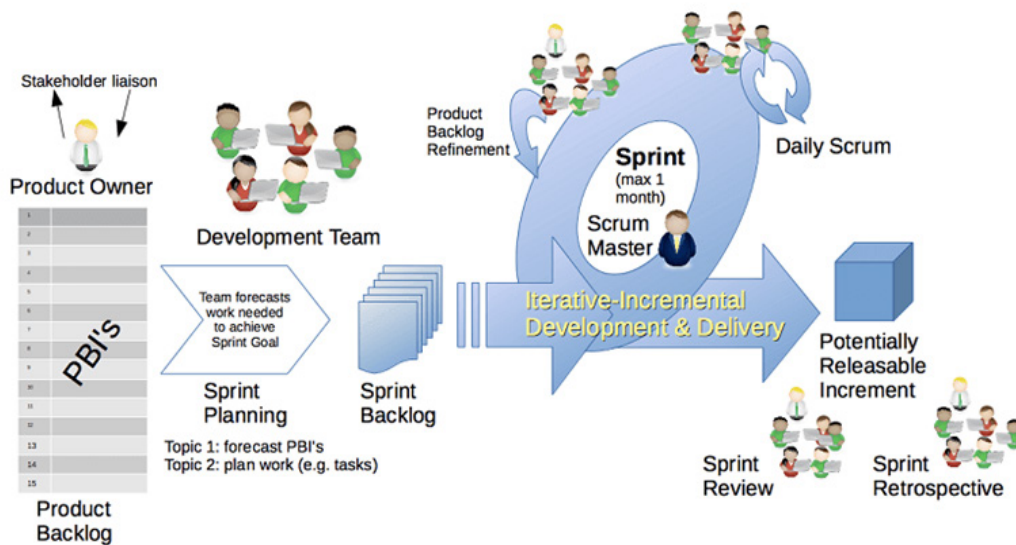


Fig.6.3.2 Scrum development process

Conclusion

Agile and Scrum are more than just buzzwords. They're practical approaches to working smarter, staying flexible, and focusing on delivering value. Whether you're building a software app or managing any project, these methods teach us to listen, adapt, and improve step by step.

In the next chapters, we'll explore Agile tools, Scrum boards, and how teams handle real-life challenges using these techniques.

Recap

- ◆ Traditional development: long timelines, shifting needs, mismatched products
- ◆ Agile: small deliveries, continuous feedback, flexible and adaptive
- ◆ Agile Manifesto
 - Created in 2001 by 17 developers at Snowbird, Utah
 - Goal: lightweight, flexible, user-focused development
 - Focus on values, collaboration, adaptability, trust, and customer value
- ◆ 4 Core Values
 - Individuals & interactions over processes & tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan
- ◆ The 12 Principles of Agile
 - Early & continuous delivery of valuable work
 - Break large work into smaller chunks
 - Self-organizing teams produce best results
 - Support and trust motivated individuals
 - Promote sustainable development
 - Maintain constant pace
 - Welcome changing requirements
 - Daily collaboration between team & business
 - Regular reflection & improvement
 - Progress = completed work
 - Focus on technical excellence
 - Use change as a competitive edge
- ◆ Scrum: An Agile Framework
 - Popular method to apply Agile
 - Like a sport: rules, roles, feedback loops
 - Small, repeatable cycles (Sprints)
- ◆ Key Roles
 - Product Owner – prioritizes features, represents customer

- Scrum Master – enforces rules, removes blockers, improves team workflow
- Development Team – builds, tests, delivers product; self-organizing
- ◆ Scrum Process Flow
 - Product Backlog
 - List of features/user stories
 - Maintained by Product Owner
 - Flexible, updated often
- ◆ Sprint Planning
 - Team selects tasks from backlog
 - Plans how to complete them
- ◆ Sprint (2–4 weeks)
Team works on selected items
 - Daily Stand-up:
 - What did I do yesterday?
 - What will I do today?
 - Any blockers?
- ◆ Sprint Review
 - Team demos work to stakeholders
 - Collect feedback for next steps
- ◆ Sprint Retrospective
 - Team reflects on the sprint
 - Identifies improvements
 - Plan better for next sprint
 - Cycle repeats, product evolves each sprint

Objective Type Questions

1. Where was the Agile Manifesto created?
2. How many core values are in the Agile Manifesto?
3. Which term refers to short cycles in Scrum?
4. What type of team produces the best results in Agile?

5. What is prioritized more in Agile: interactions or tools?
6. What is the common format used to write Agile requirements?
7. Who maintains the product backlog?
8. What kind of development does Agile promote: rigid or flexible?
9. Which role removes blockers in Scrum?
10. What kind of pace does Agile encourage: variable or constant?
11. What meeting is held every day during a sprint?
12. What is the final meeting after a sprint called?
13. Which Agile method is compared to a sport?
14. What is the name of the initial list of user stories and features?

Answers to Objective Type Questions

1. Utah
2. Four
3. Sprint
4. Self-organizing
5. Interactions
6. Story
7. Owner
8. Flexible
9. ScrumMaster
10. Constant
11. Stand-up
12. Retrospective
13. Scrum
14. Backlog

Assignments

1. Explain the four core values of the Agile Manifesto with examples.
2. Describe the roles in a Scrum team. How do they collaborate during a Sprint?
3. What is a Product Backlog? Who manages it, and how is it used throughout development?
4. Compare Agile methodology with the traditional Waterfall model. Mention at least three major differences.
5. Discuss the importance of customer collaboration in Agile development.

Suggested Reading

1. DevOps: *A Software Architect's Perspective*. " O'Reilly Media, 2019
2. Site Reliability Engineering: *How Google Runs Production Systems*" by Niall Richard Murphy, Betsy Beyer, Chris Jones, and Jennifer Petoff

Reference

1. The Phoenix Project: *A Novel About IT, DevOps, and Helping Your Business Win*" by Gene Kim, Kevin Behr, and George Spafford
2. The DevOps Handbook: *How to Create World-Class Agility, Reliability, & Security in Technology Organizations*" by Gene Kim, Patrick Debois, John Willis, and Jez Humble
3. Accelerate: *The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*" by Nicole Forsgren, Jez Humble, and Gene Kim



Continuous Integration and Deployment

Learning Outcomes

The learner will be able to :

- ◆ explain the principles and features of Continuous Integration (CI)
- ◆ describe the benefits of Continuous Integration
- ◆ identify the key features of Continuous Deployment
- ◆ understand the structure and stages of a CI/CD pipeline

Prerequisites

In the fast-moving world of software development, writing great code isn't enough. Getting that code into the hands of users quickly, safely, and reliably is just as important. That's exactly what Continuous Integration and Continuous Deployment help achieve.

Imagine building an app with your team where every change you make is automatically tested, built, and ready to deploy in minutes. No more waiting days to merge your work or worrying about last-minute bugs before a release. CI/CD empowers teams to move faster, collaborate better, and innovate with confidence.

In this chapter, you'll uncover how CI/CD works behind the scenes, why it's becoming a must-have in modern DevOps workflows, and how you can use it to streamline your own software projects. Whether you're a budding developer or a future DevOps engineer, this journey into CI/CD will change the way you think about building and delivering software.

Keywords

Continuous integration, continuous deployment, automated build, automated test, CI/CD pipeline



Discussion

In the fast-paced world of software development, delivering high-quality applications quickly and efficiently is essential. DevOps, a combination of Development and Operations which aims to bridge the gap between software development and IT operations. One of the key practices in DevOps is CI/CD, which stands for Continuous Integration and Continuous Deployment (or Delivery). These practices help automate and streamline the software release process, making deployments faster, more reliable, and less error-prone.

6.4.1 What is Continuous Integration?

Continuous Integration is a DevOps practice where developers regularly merge their code changes into a shared repository, usually several times a day. Each merge triggers an automated build and test process, ensuring that the new code integrates smoothly with the existing codebase.

6.4.1.1 Features of CI

1. Frequent Code Integration

In a CI environment, developers integrate their code into a shared repository frequently, often multiple times a day. This practice ensures that code conflicts are detected and resolved early, rather than accumulating over time. Frequent integration encourages collaboration, promotes shared code ownership, and reduces the complexity of merges and debugging.

2. Automated Builds

Every time a developer commits code to the repository, it triggers an automated build process. This includes compiling the code, checking dependencies, and preparing the application for testing or deployment. Automated builds ensure that the application remains in a deployable state, making it easier to deliver features, updates, or fixes at any time.

3. Automated Testing

CI systems run a suite of automated tests every time new code is integrated. These tests often include:

- ◆ Unit Tests to check individual components or functions.
- ◆ Integration Tests to verify that modules interact correctly.
- ◆ Regression Tests to ensure new changes haven't broken existing functionality.

Automated testing catches bugs early in the development cycle, which saves time and effort down the line and helps maintain a high standard of quality.

4. Immediate Feedback

One of the core advantages of CI is rapid feedback. If the code integration introduces a problem such as build failures, failed tests, or code conflicts, developers are notified immediately. This allows them to quickly address the issue, minimizing the risk of the problem affecting other team members or reaching production.

6.4.1.2 Benefits of CI

- ◆ Early detection of bugs and errors.
- ◆ Reduced integration problems.
- ◆ Faster development cycles.
- ◆ Improved team collaboration.

6.4.2 What is Continuous Deployment?

Continuous Deployment is the practice of automatically releasing every change that passes automated tests to production. Sometimes, the term Continuous Delivery is used, which means changes are automatically prepared for release but may require manual approval before deployment.

6.4.2.1 Features of CD

1. Automated Release Process

CD automates the steps required to release software to production, minimizing manual tasks and human errors. Once the code passes all tests and validations in the CI pipeline, it's automatically prepared for release.

In Continuous Delivery, the release is still triggered manually but is ready to be deployed at any time. But in Continuous Deployment, the release happens automatically once the code passes through the pipeline.

This automation ensures faster, more reliable, and consistent software releases, allowing teams to deliver new features and updates more frequently.

2. Monitoring and Alerting

After deployment, CD systems include robust monitoring and alerting mechanisms to track application performance, stability, and user experience in real-time. These tools detect issues like:

- ◆ Latency spikes
- ◆ Service outages
- ◆ Error rates
- ◆ Resource usage

Automated alerts notify teams immediately when problems arise, enabling rapid response and minimizing downtime or negative user impact.

3. Rollback Mechanisms

One of the essential safety features of CD is the ability to quickly roll back a deployment if something goes wrong. This can include automatically reverting to a previous stable version, Using feature toggles to disable problematic functionality or Applying hotfixes in a controlled manner.

Rollback mechanisms reduce the risk associated with frequent deployments and ensure that failures can be contained and corrected swiftly without long-term disruption.

6.4.3 CI/CD Pipeline

A CI/CD pipeline is an automated sequence of steps that code changes go through to be built, tested, and deployed. It usually consists of the following stages:

1. Source Code Management: Developers commit code to a version control system (e.g., Git).
2. Build Stage: Code is compiled and prepared.
3. Test Stage: Automated tests verify functionality.
4. Deploy Stage: Code is deployed to staging and/or production environments.
5. Monitor Stage: System behavior is monitored post-deployment.

Source Stage

In the Source Stage, an organization stores its source code in a centralized version control system, such as GitHub, GitLab, or similar platforms. This stage is crucial for developers to track and manage code changes, ensuring proper coordination within the team.

This stage also enhances collaboration, as different teams can check in their code, conduct code reviews, and approve changes for the next steps. Integrated Development Environments (IDEs), like Visual Studio Code, can automate version control processes to streamline this workflow.

Build Stage

In the Build Stage, developers commit their code to the organization's version control system. This action triggers the continuous integration tool, which then runs the predefined tests associated with the code.

During this phase, the code is compiled, dependencies are resolved, and artifacts are created and stored in repositories such as JFrog. Software packages are prepared for deployment, which may include files like WAR files for platforms such as WebLogic and Tomcat. Additionally, in this stage, environments are set up using infrastructure as code, containers are created and deployed on various pods, and pre-configured images for future rollouts are stored.

Test Stage

The Test Stage focuses on maintaining a fast and consistent feedback loop for testing. The goal is to facilitate continuous and early testing, but this remains one of the main challenges many organizations face.

Efficient testing requires breaking down the CI/CD process and improving testing practices. One significant challenge is managing testing data, as teams often lack the necessary historical data for comprehensive testing. Automating data creation, management, and maintenance is essential for a fully automated CI/CD pipeline.

Proper data testing helps organizations by accelerating both Continuous Integration and Continuous Deployment processes, ensuring a smoother and faster flow of code changes through the pipeline. It also plays a crucial role in maintaining immutability, which ensures that data remains unchanged once deployed, preserving its integrity. Data testing supports the enforcement of security measures and governance protocols, safeguarding sensitive information and ensuring compliance. Furthermore, it boosts the productivity of scrum teams by enabling them to focus on development and testing without being hindered by data-related issues.

Deploy Stage

The Deploy Stage is where the code is ready for deployment to various environments such as QA, pre-production, or production, according to the organization's deployment strategy. This stage may involve automated continuous deployment, where deployment is handled automatically.

Organizations typically adopt deployment strategies such as canary releases, rolling deployments, or blue/green deployments to ensure minimal downtime and seamless transitions between versions.

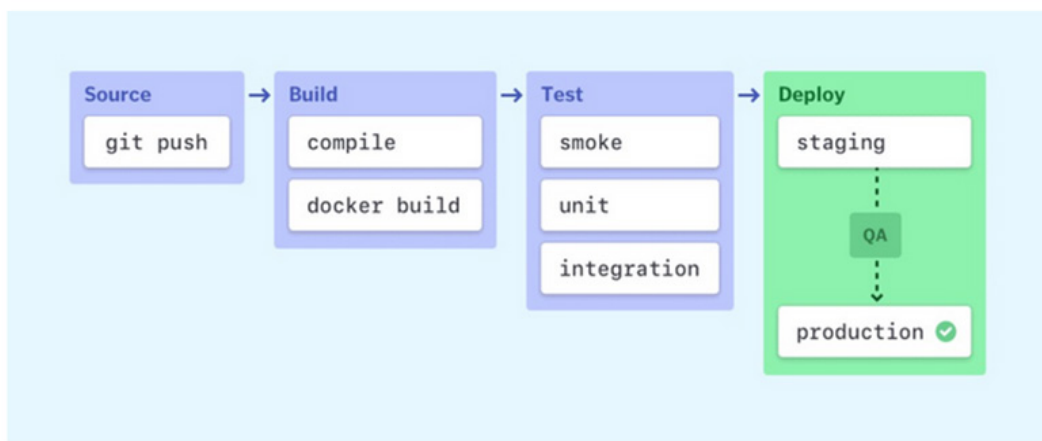


Fig. 6.4.1 Phases of CI/CD Pipeline

Recap

- ◆ Continuous Integration (CI)
 - Developers merge code frequently (multiple times/day)
 - Each merge triggers automated build and test
 - Early bug detection and conflict resolution
- ◆ Features of CI
 - **Frequent Code Integration**
 - Small, regular updates
 - Reduced merge conflicts
 - Shared code ownership
- ◆ **Automated Builds**
 - Compile code, check dependencies
 - Prepare for testing/deployment
 - Keeps code in deployable state
- ◆ **Automated Testing**
 - Unit tests: Check individual components
 - Integration tests: Check module interaction
 - Regression tests: Detect unintended side-effects
- ◆ **Immediate Feedback**
 - Quick notification on failures
 - Rapid debugging and resolution
- ◆ Benefits of CI
 - Early bug detection
 - Reduced integration issues
 - Faster development cycles
 - Better team collaboration
- ◆ Continuous Deployment (CD)
 - Automatically deploy code that passes tests
 - Continuous Delivery = Manual trigger
 - Continuous Deployment = Fully automated

- ◆ Features of CD
 - **Automated Release Process**
 - Reduces manual effort and errors
 - Consistent, reliable releases
- ◆ **Monitoring and Alerting**
 - Real-time performance tracking
 - Alerts on errors, latency, outages
- ◆ **Rollback Mechanisms**
 - Revert to stable version
 - Use of feature toggles
 - Safe and quick recovery
- ◆ CI/CD Pipeline Stages
 - **Source Stage**
 - Version control (GitHub, GitLab)
 - Code reviews and team collaboration

1. **Build Stage**

- Triggered by code commit
- Compile, resolve dependencies
- Create deployment artifacts (e.g., WAR files)
- Use of containers and infrastructure as code
- ◆ **Test Stage**
 - Continuous and automated testing
 - Manage and automate test data
 - Ensure immutability and security
 - Improve scrum team productivity
- ◆ **Deploy Stage**
 - Deploy to QA, pre-prod, or production
 - Use of strategies like canary, rolling, blue/green
 - Enable zero-downtime deployments
- ◆ **Monitor Stage**
 - Monitor app behavior post-deployment
 - Track metrics like performance, error rates

Objective Type Questions

1. What practice merges code frequently into a shared repository?
2. Which process is triggered automatically after every code commit?
3. What kind of tests are used to check individual components?
4. What type of tests verify the interaction between modules?
5. Which tests ensure new changes don't break existing functionality?
6. What provides rapid feedback in CI/CD?
7. Which process involves automatically releasing code to production?
8. What is the manual version of Continuous Deployment called?
9. What automates the steps for releasing software?
10. What detects performance issues after deployment?
11. What compiles code and resolves dependencies in CI/CD?
12. Which type of test data ensures consistency and reliability?
13. What ensures that data remains unchanged once deployed?
14. What development methodology does CI/CD support?
15. Which stage involves storing deployment artifacts like WAR files?
16. What is used to manage code versions like GitHub or GitLab?

Answers to Objective Type Questions

1. Integration
2. Build
3. Unit
4. Integration
5. Regression
6. Feedback
7. Deployment
8. Delivery
9. Automation
10. Monitoring
11. Build
12. Test
13. Immutability
14. Agile
15. Build
16. VersionControl

Assignments

1. Explain the differences between Continuous Integration, Continuous Delivery, and Continuous Deployment. Use examples to support your answer.
2. Draw and label a CI/CD pipeline, describing each stage (Source, Build, Test, Deploy, Monitor) with tools and practices used at each step.
3. Discuss the role of automated testing and rollback mechanisms in ensuring software quality and reliability in a CI/CD workflow.

Suggested Reading

1. DevOps: *A Software Architect's Perspective*." O'Reilly Media, 2019
2. Site Reliability Engineering: *How Google Runs Production Systems*" by Niall Richard Murphy, Betsy Beyer, Chris Jones, and Jennifer Petoff

Reference

1. The Phoenix Project: *A Novel About IT, DevOps, and Helping Your Business Win*" by Gene Kim, Kevin Behr, and George Spafford
2. The DevOps Handbook: *How to Create World-Class Agility, Reliability, & Security in Technology Organizations*" by Gene Kim, Patrick Debois, John Willis, and Jez Humble
3. Accelerate: *The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*" by Nicole Forsgren, Jez Humble, and Gene Kim



SREENARAYANAGURU OPEN UNIVERSITY

QP CODE:

SET-1

Reg. No:

Name :

END SEMESTER BCA EXAMINATION

DISCIPLINE CORE - **B21CA08DC SOFTWARE ENGINEERING (CBCS - UG)**

2024-25 - Admission Onwards

Time: 3 Hours

Max Marks: 70

SECTION A

Objective Types Questions; Answer **any Ten** (1x 10=10)

1. Which model allows flexibility and is risk-driven, suitable for large and complex projects?
2. What is a deployment tool?
3. Which software process model emphasizes creating early, working versions of the system to gather user feedback.
4. What does KLOC mean?
5. What is the first step in identifying the critical path?
6. What is modular design in software engineering?
7. What is stepwise refinement?
8. What is the main goal of coding standards?
9. Name any one type of software testing.
10. A Profile Diagram is used to create _____ extensions to UML models.
11. Which tool is commonly used for code consistency checking?*
12. Which element in a class diagram holds variables?
13. What is the primary objective of DevOps?
14. Name one advantage of Continuous Deployment.
15. Write any two features of the spiral model.

SECTION B

Very Short Answer Questions; answer **any Five** (2x5=10)

16. Name any two advantages of the Agile Model.
17. What is the purpose of the Software Development Life Cycle (SDLC)?
18. What are the limitations of Gantt Charts?



19. What is object-oriented design in software engineering?
20. What is the purpose of code review?
21. Write any four advantages of UML.
22. What are the four core values of the Agile Manifesto?
23. What is cohesion in software design?
24. What is configuration management in software deployment?
25. What do you mean by Software Design Verification?

SECTION C

Short Answer Questions; answer **any five**

(4x5=20)

26. Explain about coding and testing.
27. What are the key features of the Incremental model?
28. What is a software product? How is it different from a software program?
29. What is the main goal of the Risk Analysis phase in the Spiral model?
30. Explain coupling and cohesion in modular design. Why are they important in software design?
31. Differentiate between Unit Testing and System Testing with examples.
32. List the steps to create a UML diagram.
33. What is Continuous Deployment, and how is it different from Continuous Integration?
34. Explain various stages in the DevOps Life Cycle.
35. What is a profile diagram? Explain its purpose and importance.

SECTION D

Long answer Question; Answer **any two**

(15x2=30)

36. Compare different SDLC models.
37. Explain the concept of the Critical Path Method (CPM) in project management. Why is identifying the critical path essential for successful project completion? Illustrate your answer with a suitable example.
38. Describe the different types of UML diagrams.
39. Describe in detail the Agile Scrum framework, covering roles, events, and artifacts, along with a neat process flow diagram.



SREENARAYANAGURU OPEN UNIVERSITY

QP CODE:

SET-2

Reg. No:

Name :

END SEMESTER BCA EXAMINATION

DISCIPLINE CORE - **B21CA08DC SOFTWARE ENGINEERING (CBCS - UG)**

2024-25 - Admission Onwards

Time: 3 Hours

Max Marks: 70

SECTION A

Objective Types Questions; Answer **any Ten** (1x 10=10)

1. What does COCOMO stand for?
2. What is the primary purpose of a Software Requirements Specification (SRS)?
3. Which term refers to the breaking down software into separate, distinct modules that can be named and addressed individually.
4. What are the two different strategies or approaches of software design?
5. What is the full form of UML?
6. What does 'software validation' mean?
7. Which approach uses actors and interactions to describe system requirements?
8. What component links stereotypes to base UML elements?
9. What is a design walkthrough?
10. Name a CD tool used in DevOps.
11. What container tool is mentioned for packaging applications?
12. What is the full form of SDLC?
13. What is the final phase in the Waterfall model?
14. What is the main objective of project size estimation in software project management?
15. Why are project size metrics important?

SECTION B

Very Short Answer Questions; answer **any Five** (2x5=10)

16. What is a Software Program?
17. Why is requirement validation important in software development?
18. Explain how a Gantt Chart is used for project scheduling and tracking.



19. Write about any four types of coupling.
20. What is Critical Design Review (CDR) in software engineering?
21. Mention two advantages of using UML in software engineering
22. Explain the benefits of DevOps.
23. Write four types of User Interfaces
24. What is Continuous Deployment?
25. Differentiate between static and dynamic UML diagrams (any two points)

SECTION C

Short Answer Questions; answer **any five**

(4x5=20)

26. Explain about coding and testing.
27. What are the key features of the Incremental model?
28. What is a software product? How is it different from a software program?
29. What is the main goal of the Risk Analysis phase in the Spiral model?
30. Explain coupling and cohesion in modular design. Why are they important in software design?
31. Differentiate between Unit Testing and System Testing with examples.
32. List the steps to create a UML diagram.
33. What is Continuous Deployment, and how is it different from Continuous Integration?
34. Explain various stages in the DevOps Life Cycle.
35. What is a profile diagram? Explain its purpose and importance.

SECTION D

Long answer Question; Answer **any two**

(15x2=30)

36. Explain the stages of SDLC.
37. Explain in detail the various project estimation techniques with suitable examples.
38. Explain about coding standards and code review.
39. Explain the various types of UML diagrams with their uses and advantages. Also, discuss the steps involved in creating UML diagrams and the tools used for creating them.



സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ് വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരിപ്പുഴ ശ്രീകുമാർ

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kozhikode

Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin: 673002
Ph: 04952920228
email: rckdirector@sgou.ac.in

Thalassery

Govt. Brennen College
Dharmadam, Thalassery,
Kannur, Pin: 670106
Ph: 04902990494
email: rctdirector@sgou.ac.in

Tripunithura

Govt. College
Tripunithura, Ernakulam,
Kerala, Pin: 682301
Ph: 04842927436
email: rcedirector@sgou.ac.in

Pattambi

Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad,
Kerala, Pin: 679303
Ph: 04662912009
email: rcpdirector@sgou.ac.in

**DON'T LET IT
BE TOO LATE**

**SAY
NO
TO
DRUGS**

**LOVE YOURSELF
AND ALWAYS BE
HEALTHY**



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

Software Engineering

COURSE CODE: B21CA08DC



YouTube



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841

ISBN 978-81-989642-0-5



9 788198 964205