SOFTWARE ENGINEERING

Mr. Rakesh Kumar Roshan
Mr. Deepak Kumar Ravi
Dr. B.Jothi
Dr. N. Kumar

AIP

Author's Profiles

Mr. Rakesh Kumar Roshan is working as an Assistant Professor in the Department of Computer Science & Engineering, RRSDCE, Begusarai has about 10 years of teaching experience. He received his B.Tech degree in Computer Science & Engineering and M.E. degree in Software Engineering from Birla Institute of Technology, Mesra, Ranchi. He has published 5 research papers in international journals and various international conferences. His areas of research include Artificial Intelligence, Deep Learning, Deep Learning Applications for Biomedical Engineering, Optimization Algorithms, and Bioinformatics.

Mr. Deepak Kumar Ravi is currently working as an Assistant Professor in the Department of Computer Science and Engineering at Government Engineering College, Palamu. He has approximately five years of teaching experience. He received his B.Tech degree in Computer Science and Engineering from Ranchi University Ranchi and completed M.E. degree in Software Engineering from Birla Institute of Technology, Mesra, Ranchi. He has published one research paper in a refereed international journal. His research interests include Software Engineering, Machine Learning, Federated Learning, and Algorithms Analysis.

B.JOTHI is currently working as Associate Professor in the Department of Computer Science and Engineering, SRM Institute of Science and Technology, Chennai, India. She has received his Doctorate degree from SRM University in 2022.Her area of interest in research are IoT Networks, Data Mining, and Software Engineering. She is also guiding UG and PG students in various research problems in the above-mentioned areas and also published more than 35 papers in peer reviewed journals and in various National and International conferences. She has published 3 patents grants. She is a member of Institution of Engineers in India (IEI) and Indian Science Congress Association (ISCA).

Dr. N. Kumar is currently working as a Professor in the Department of Computer Science and Engineering, Vels Institute of Science, Technology and Advanced Studies (VISTAS), Chennai. He obtained his Ph.D. in Computer Science and Engineering from Karpagam University, Coimbatore, in 2015. With a distinguished academic career spanning 20 years, he has extensive teaching and research experience in the field of Computer Science and Engineering. He has published 83 research articles in reputed international journals, presented 46 papers at international conferences, authored 3 books, and holds 14 published patents, reflecting his strong commitment to innovation, research, and academic contribution. He has completed 13 Research Scholar. His key research interests include Computer Networks, Mobile Ad Hoc Networks (MANET), Wireless Sensor Networks (WSN), Cloud Computing, Image Processing, Network Security, Big Data Analytics, and Machine Learning.

αIP

Alpha International Publication (AIP)

www.alphainternationalpublication.com | editoraippublications@gmail.com

ISBN : 978-93-7361-317-8

9 789373 613178

SOFTWARE ENGINEERING

AUTHORS

Mr. RAKESH KUMAR ROSHAN
Assistant Professor
Department of Computer Science & Engineering, Rashtrakavi Ramdhari Singh Dinkar College of Engineering, Begusarai, Bihar, 851134

Mr. DEEPAK KUMAR RAVI
Assistant Professor
Department of Computer Science & Engineering, GEC Palamu-822118

Dr. B. JOTHI
Associate professor,
Department of Computational Intelligence, SRM Institute of Science and Technology, Kattankulathur, Chennai, India.

Dr. N Kumar
Professor,
Department of Computer Science and Engineering,
Vels Institute of Science, Technology and Advanced Studies,
Pallavaram, Chennai -600117

Alpha International Publication (AIP)

Title of the Book: ***Software Engineering***

Edition: First - 2026

Disclaimer

The authors are solely responsible for the contents published in this text book. The publishers or editors do not take any responsibility for the same in any manner. Errors, if any, are purely unintentional and readers are requested to communicate such errors to the editors or publishers to avoid discrepancies in future.

ISBN: 978-93-7361-317-8
MRP: 640/-
PUBLISHER & PRINTER: Alpha International Publication (AIP)
Contact: +917019991025
Website: https://www.alphainternationalpublication.com/

ACKNOWLEDGEMENT

The creation of this book, ***Software Engineering,*** has been a journey of collective intellect, rigorous inquiry, and a shared commitment to the evolution of computing sciences. While the pages that follow bear the fruits of our research and synthesis, they also represent the silent contributions of countless individuals whose support made this ambitious project a reality.

We owe an immense debt of gratitude to the global community of software engineers, researchers, and educators. This work is built upon the foundational theories and innovative practices established by those who preceded us. We are particularly grateful to the peer reviewers and subject matter experts who provided invaluable feedback during the manuscript's developmental stages. Their critical insights, often challenging our assumptions, have significantly sharpened the technical accuracy and pedagogical clarity of this volume.

Our sincere thanks go to the editorial and production teams at ***ALPHA International Publication.*** From the initial proposal to the final press run, their professionalism has been exemplary. We appreciate their patience with the iterative nature of documenting a field as rapidly changing as software engineering. Their commitment to high production standards ensures that the complex diagrams, architectural patterns, and code snippets within these chapters are presented with the precision required for academic excellence.

We would like to acknowledge the various academic institutions and research laboratories that provided the environment necessary for this work. The access to cutting-edge resources, licensed software suites, and collaborative forums allowed us to bridge the gap between theoretical software models and industry-standard applications.

A special mention must be reserved for our students—past and present. It was through the act of teaching that the gaps in current literature became apparent, sparking the inspiration for this book. Their inquisitive minds, frequent

questions, and unique perspectives on debugging and systems design served as the ultimate litmus test for the explanations provided herein. They are the primary reason this book exists, and their success in the industry remains our greatest reward.

Software engineering is a field defined by practice. We extend our thanks to the numerous industry professionals and consultants who shared "real-world" case studies and anecdotal evidence of systemic failures and triumphs. These insights have allowed us to ground abstract concepts—such as Agile methodologies, DevOps integration, and Cloud Architecture—in the practical realities of modern enterprise environments.

The demands of authoring a comprehensive textbook are not borne by the authors alone. We are deeply grateful to our families for their unwavering support, patience, and understanding during the countless hours of late-night drafting and weekend revisions. Their encouragement provided the emotional sustenance required to see this project through to completion.

Finally, we acknowledge those whose contributions are often overlooked: the librarians who sourced obscure citations, the technical assistants who verified code repositories, and the administrative staff who managed the logistical hurdles of a multi-author collaboration.

This book is a testament to the belief that software engineering is not merely a technical discipline, but a collaborative human endeavor. We hope this work serves as a valuable guide for the next generation of engineers who will build the digital infrastructure of tomorrow.

With gratitude and warm regards,

Mr. Rakesh Kumar Roshan
Mr. Deepak Kumar Ravi
Dr. B. Jothi
Dr. N. Kumar

PREFACE

Software has become the backbone of modern civilization. From healthcare and banking to aerospace, education, governance, and artificial intelligence systems, software systems shape the way individuals, industries, and governments operate. As systems grow in scale, complexity, and criticality, the need for systematic, disciplined, and engineering-driven approaches to software development has become more important than ever. This book, "***Software Engineering***", is designed to provide a comprehensive foundation in the principles, practices, and evolving methodologies of modern software development.

The discipline of Software Engineering emerged to address the challenges of reliability, scalability, maintainability, cost control, and quality assurance in software systems. Unlike ad-hoc programming, Software Engineering emphasizes structured processes, formal documentation, systematic design, rigorous testing, and continuous improvement. This book aims to bridge the gap between theoretical foundations and real-world application, making it suitable for undergraduate and postgraduate students, educators, and industry practitioners.

Chapter 1: Software Process and Planning introduces the evolution of software process models, from traditional approaches to contemporary frameworks. It discusses feasibility studies, project estimation techniques, scheduling, risk management, and configuration management. The chapter lays the groundwork for understanding how structured planning ensures successful project execution.

Chapter 2: Requirement Analysis and Specifications focuses on one of the most critical phases of software development—understanding what the system must achieve. It covers requirement elicitation techniques, stakeholder analysis, Software Requirement Specification (SRS) documentation, modeling

approaches, and validation methods. Emphasis is placed on clarity, completeness, and traceability of requirements.

Chapter 3: Software Design explores architectural and detailed design principles. Topics include modularity, abstraction, coupling and cohesion, design patterns, UML modeling, and user interface design. The chapter demonstrates how strong design decisions influence maintainability, performance, and scalability.

Chapter 4: Software Testing highlights quality assurance mechanisms essential for delivering reliable systems. It covers unit testing, integration testing, system testing, validation and verification techniques, test case design strategies, automation tools, and defect management. The chapter emphasizes the importance of testing throughout the development lifecycle.

Chapter 5: Agile Development and DevOps addresses modern, adaptive methodologies that promote flexibility, collaboration, and rapid delivery. It introduces Agile principles, Scrum, Kanban, continuous integration, continuous delivery, and DevOps culture. The chapter explains how automation and collaboration between development and operations teams accelerate deployment and improve product quality.

This book has been structured in a progressive manner, beginning with fundamental concepts and advancing toward contemporary industry practices. Each chapter is organized to include conceptual explanations, practical insights, illustrative examples, and discussion-oriented topics to enhance analytical thinking.

The objective of this book is not only to help students succeed in academic examinations but also to equip them with the mindset and skills required to develop high-quality, dependable, and scalable software systems in real-world environments. In a rapidly evolving technological world, disciplined Software Engineering practices remain the cornerstone of innovation and sustainability.

We hope this book serves as a valuable resource for learners, educators, and professionals striving to build robust and impactful software solutions.

Sincerely,

Mr. Rakesh Kumar Roshan

Mr. Deepak Kumar Ravi

Dr. B. Jothi

Dr. N. Kumar

UNIT NO	CONTENTS	PAGE NO
I	**SOFTWARE PROCESS AND PLANNING**	
	1.1 Introduction to Software Engineering	**4**
	1.2 Objectives	**7**
	1.3 Principles and Practices	**11**
	1.4 The Software Development	**13**
	1.5 Life Cycle	**20**
	1.6 Methodologies Paradigm and Practices	**26**
	1.7 Software Development Paradigms	**34**
	1.8 Modern SDLC Practices and Best Practices	**36**
	1.9 Project Planning Process	**37**
	1.10 Software Project Estimation	**39**
	1.11 Decomposition techniques	**43**
	1.12 Empirical estimation models	**49**
	1.13 The make/buy decision	**51**
	1.14 Project scheduling	**53**
	1.15 Risk Management	**55**
	1.16 Handling Ethical Dilemmas	**59**
II	**REQUIREMENT ANALYSIS AND SPECIFICATIONS**	
	2.1 Software Requirements	**61**
	2.2 Functional and Non-functional Requirements	**68**
	2.3 Security requirements	**73**

UNIT I

SOFTWARE PROCESS AND PLANNING

1.1 Introduction to Software Engineering

Software is a program or set of programs containing instructions that provide the desired functionality. Engineering is the process of designing and building something that serves a purpose efficiently and cost-effectively.

Figure 1.1

Software Engineering is the systematic process of designing, developing, testing, and maintaining software using techniques like requirements analysis, design, testing, and maintenance. It ensures software is high-quality, reliable, and maintainable, especially for large software systems.

Software Engineering improves efficiency, quality, and budget management, helping deliver software on time, within budget, and meeting all requirements. It continuously adopting new tools and methodologies to handle complexity and evolving technologies.

Key Principles

Modularity: Divide software into small, independent parts that are easy to develop and test.

Abstraction: Show only what is necessary and hide internal implementation details.

Encapsulation: Keep data and related functions together and protect them from outside access.

Reusability: Build components that can be reused in different projects to save time and effort.

Maintenance: Regularly update software to fix bugs, improve features, and enhance security.

Testing: Check the software to ensure it works correctly and meets requirements.

Design Patterns: Use proven solutions for common software design problems.

Agile Methodologies: Develop software in small steps with frequent feedback and flexibility.

Continuous Integration & Deployment (CI/CD): Frequently merge code changes and automatically deploy updates.

Main Attributes of Software Engineering

Efficiency:

It provides a measure of the resource requirement of a software product efficiently.

- Reliability: It assures that the product will deliver the same results when used in similar working environment.
- Reusability: This attribute makes sure that the module can be used in multiple applications.
- Maintainability: It is the ability of the software to be modified, repaired, or enhanced easily with changing requirements.

Dual Role of Software

There is a dual role of software in the industry. The first one is as a product and the other one is as a vehicle for delivering the product.

1. As a Product

- It delivers computing potential across networks of Hardware.
- It enables the Hardware to deliver the expected functionality.
- It acts as an information transformer because it produces, manages, acquires, modifies, displays, or transmits information.

2. As a Vehicle for Delivering a Product

- It provides system functionality (e.g., payroll system).
- It controls other software (e.g., an operating system).
- It helps build other software (e.g., software tools).

What Careers Are There in Software Engineering?

A degree in software engineering and relevant experience can be utilized to explore several computing job choices. Software engineers have the opportunity to seek well-paying careers and professional progress, although their exact possibilities may vary depending on their particular school, industry, and region.

Following are the job choices in software engineering:

- SWE (Software Engineer)
- SDE (Software Development Engineer)
- Web Developer
- Quality Assurance Engineer
- Web Designer
- Software Test Engineer
- Cloud Engineer ·
- Front-End Developer
- Back-End Developer
- DevOps Engineer.

- Security Engineer.

What Tasks do Software Engineers do?

The main responsibility of a software engineer is to develop useful computer programs and applications. Working in teams, you would complete various projects and develop solutions to satisfy certain customer or corporate demands.

Some of the key responsibilities of software engineer are:

Requirement Analysis: Collaborating with stakeholders to understand and gather the requirements to design and develop software solutions.

Design and Development: Creating well-structured, maintainable code that meets the functional requirements and adheres to software design principles.

Testing and Debugging: Writing and conducting unit tests, integration tests, and debugging code to ensure software is reliable and bug-free.

Code Review: Participating in code reviews to improve code quality, ensure adherence to standards, and facilitate knowledge sharing among team members.

Maintenance: Updating and maintaining existing software systems, fixing bugs, and improving performance or adding new features.

Documentation: Writing clear documentation, including code comments, API documentation, and design documents to help other engineers and future developers understand the system.

1.2 Objectives

A program or set of programs containing instructions that offer desired functionality is referred to as software. And engineering is the process of creating and building anything that serves a certain function and solves issues in a cost-effective manner. Software engineering is the systematic, disciplined, quantitative study and approach to designing, developing, operating, and maintaining a software system.

Maintainability

Software is not a static entity. In today's market, if you produce a valuable product that functions flawlessly but is difficult to tweak and adapt to new requirements, it will not survive. Maintainability is a long-term component of software that indicates how easily it may adapt and alter, which is crucial in today's agile environment.

What is Software Maintenance

The simplicity with which you can repair, improve, and comprehend software code is referred to as maintainability. Software maintenance is a step of the software development cycle that begins after the product has been delivered to the client. Developers provide maintainability by constantly modifying software to suit new client expectations and handling customer complaints. This includes bug fixes, optimizing existing functionality, and modifying code to prevent future problems.

Why Does Software Require Maintenance?

There are several reasons to maintain software after it has been provided to the customer:

- Bug fixing - This entails searching for and resolving faults for the software to work smoothly.
- Enhancement - Improving the program to provide additional features requested by clients.
- Replacement - The removal of undesirable functions to increase adaptability and efficiency.
- Security issues - Resolving security flaws discovered in your proprietary code or third-party code, particularly open-source components

Efficiency

In general, efficiency is defined as the ratio of energy consumed to work performed to produce a product. However, when it comes to software development, "efficiency" is defined as the amount of software developed or

user requirements fulfilled divided by the number of resources consumed, such as time and effort, among other things. In other words, being efficient usually implies avoiding wastage. Defects, waits, overproduction, innovation, and excessive revisions are all examples of waste in software development. As a result, efficiency in software development leads to shorter product life cycles, faster time to market, and, ultimately, a better final result.

Correctness

In software engineering, correctness is accomplished when a program or system operates exactly as planned for all of its use cases. Engineers create a list of specs that their system must follow in order to be correct before developing software. A system is only ready for use if it achieves accuracy, as it may be hazardous or incomplete.

Best Practices to Achieve Correctness

It is recommended that the following best practices be followed to make the process of establishing correctness easier:

- No matter how complex the problem or use case is, it must be explained explicitly and in-depth.
- Dry run your algorithm and test its logic on paper before beginning to code.
- As soon as a component's development is finished, try to add testing.
- Reuse existing code and components that have previously been thoroughly tested.

Reusability

Software reusability is a property that refers to a software component's predicted reuse potential. Software reuse not only boosts productivity but also enhances the quality and maintainability of software products. Reusability in computer science and software engineering is defined as the use of existing assets within the software product development process; these assets are products and byproducts of the software development life cycle and include code, software components, test suites, designs, and documentation.

Testability

Testability in software refers to the extent to which any module, requirement, subsystem, or other components of the architecture can be confirmed as satisfactory or not. High testability means that errors are simple to detect, and isolating them as part of your team's regular testing procedure is straightforward.

A testable architecture should clearly indicate integration points between swappable, isolated components. It should also provide a scriptable test architecture that allows programmers and testers to replicate exact production conditions. This will enable them to duplicate failures, isolate them, and discover the appropriate solution.

Reliability

Software reliability is defined as a system's or component's ability to perform its required functions under static conditions for a set amount of time. It is the likelihood that a software system will complete its assigned task in a given environment for a set number of input cases, provided that the hardware and input are error-free. Software Functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation are all components of software quality.

Portability

Software portability is the capacity to use the same software in different contexts. It refers to software that is available on two or more platforms or that can be recompiled for them. Common types of portability include application, source code, and data portability.

Why is Software Portability Required?

Portability is synonymous with stability in Dev-ops. You don't want different behaviour on different platforms, and you don't want to waste time dealing with environment-related configurations. Imagine tuning your environments for each and every stage; no dev-ops team wants to deal with unnecessary labour simply because the code is not adaptable.

Portable software can be easily used on other platforms by development teams. So, if your development team relocates to a new environment, you don't want to waste time and resources re-developing. They also wish to avoid contractual obligations (on certain libraries or dependencies).

The key purpose for porting for sales teams is to reach a larger audience. There are different hardware and software platforms available. More users equal more profit.

Adaptability

Adaptation of software systems is almost an inevitable process due to the change in customer requirements, the need for faster development of new or maintenance of existing software systems, etc. Numerous techniques have been developed to deal with the adaptation of software systems.

Interoperability

The capacity of different solutions to freely and readily communicate with one another is referred to as software interoperability. Interoperable systems exchange real-time data without needing specialized IT assistance or behind-the-scenes code.

1.3 Principles and Practices

Software engineering principles and practices involve applying structured engineering methodologies to develop high-quality, reliable, and maintainable software. Core principles include modularity, abstraction, encapsulation, and "keeping it simple" (KISS). Key practices involve Agile development, continuous integration/deployment (CI/CD), code reviews, and early, continuous testing to meet user needs.

Core Principles of Software Engineering

These fundamental guidelines guide decision-making throughout the development lifecycle:

Modularity & Separation of Concerns: Breaking down complex systems into smaller, independent, and manageable parts.

Abstraction & Encapsulation: Hiding internal implementation details and protecting data from outside access, improving security and reducing complexity.

KISS (Keep It Simple, Stupid): Prioritizing simple designs over complex ones, as they are easier to maintain, test, and understand.

Reusability & Generality: Designing components that can be reused in different projects and ensuring software is not limited to specific, restrictive cases.

Maintainability & Quality Focus: Ensuring the software is easy to update, fix, and enhance, with quality being the primary objective.

Incremental Development: Building software in smaller steps to allow for frequent feedback and easier adjustments.

Essential Practices in Software Engineering

These are the practical actions taken by engineers to implement the principles:

Agile & Scrum Methodologies: Using iterative development cycles for flexibility and rapid delivery.

Continuous Integration/Deployment (CI/CD): Frequently merging code changes into a central repository and automatically deploying updates to detect issues early.

Code Reviews: Conducting regular peer reviews to ensure code quality, consistency, and adherence to standards.

Automated Testing: Utilizing unit, functional, and integration tests to ensure software correctness and meet requirements.

Version Control (e.g., Git): Using tools to manage changes, track history, and facilitate team collaboration.

Documentation: Keeping documentation up-to-date to aid in maintenance and knowledge sharing.

Design Patterns: Utilizing proven, standard solutions for common design problems.

Key Attributes of Quality Software

The application of these principles and practices aims to achieve:

- Reliability: The system behaves consistently under similar conditions.
- Efficiency: The software uses resources effectively.
- Scalability: The system can handle increased demand.
- Security: The system protects data and resists threats.

1.4 The Software Development

Software Development is defined as the process of designing, creating, testing, and maintaining computer programs and applications. Software development plays an important role in our daily lives. It empowers smartphone apps and supports businesses worldwide.

Software development is defined as the process of designing, creating, testing, and maintaining computer programs and applications. This diverse field combines creativity, engineering expertise, and problem-solving abilities to produce software that satisfies particular requirements and goals. Software developers, also known as programmers or coders, use a variety of programming languages and tools to create solutions for end-users or businesses.

Software developers develop the software, which itself is a set of instructions in order to perform a specific task. software have three types.

Types of Softwares

There are three basic types of Software

1. System Software

System software is software that directly operates computer hardware and provides basic functionality to users as well as other software for it to run smoothly.

2. Application Software

Application software is a software that is designed for end-user to complete a specific task. It is a product or programm that is only intended to meet the

needs of end users. It includes word processors, spreadsheets, database management, inventory, and payroll software, among other things.

3. Programming Software

Programming software is a software that is designed for programmers to develop program. It consist of code editor, compiler, interpreter, debugger etc.

Under Software Development, developers develop all the software that comes under these three category.

Steps of Software Development

Software development is a well-structured process with several key stages. While different methodologies exist, such as Agile and Waterfall, most software development projects include the following steps:

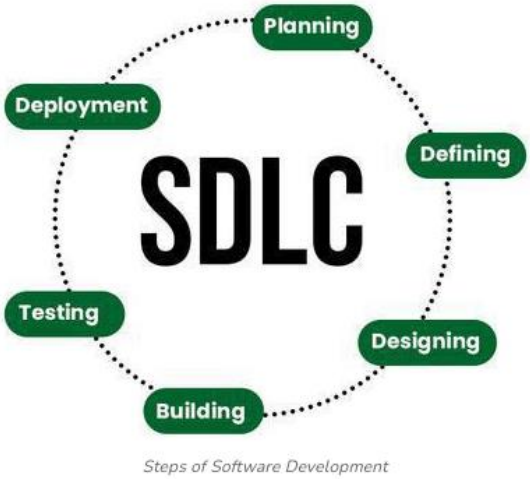


Figure 1.2

1. Requirement Analysis:

The first step in software development is understanding the requirements and based on that requirement gathering happen. This stage involves identifying the needs, objectives, and constraints of the project. The goal is to define what the software should do and what problems it will solve.

2. Design:

In the design phase, the software's architecture and user interface are developed. This step defines how the software will work and how users will

interact with it. Design includes creating wireframes, prototypes, and system architecture diagrams.

After completing the architectural design phase, developers move on to creating detailed designs for each component of the system. This includes designing not only the user interface but also encompassing databases and APIs. The intricate decisions made in these detailed designs provide valuable guidance throughout the coding phase.

3. Implementation

The most important phase of the Software Development is the implementation phase, which comes after the design phase. This phase will see the implementation of the design phase's output.

All of the planning done in the planning phase and the designing done in the designing phase are implemented in this phase. Physical source code is created and deployed in the real world during this phase.

4. Testing:

Developers utilize unit tests to evaluate small code components, such as functions or methods. These tests play a crucial role in identifying and resolving bugs within isolated elements.

Integration testing evaluates the smooth functioning of various software components. Its purpose is to ensure seamless interactions between modules and efficient data transfer among them, resulting in a robust system.

In order to ensure that the software meets all the specified requirements, system testing evaluates it as a whole. This comprehensive evaluation includes functional, performance, security, and other necessary types of testing.

User Acceptance Testing (UAT) occurs during the phase where end-users or clients validate the software to ensure it meets their requirements. Identified issues or discrepancies are promptly addressed before proceeding with deployment.

5. Deployment:

Before deployment, the development team configures the target environment, whether it's on-premises servers, cloud-based infrastructure, or end-user devices. This may involve setting up servers, databases, and configuring software dependencies.

Developers carefully plan the process of deploying software, which includes aspects such as data migration strategies, software installation procedures, and contingency measures for unexpected issues.

The software is deployed to end-users or production environments. Ongoing monitoring is critical for quickly identifying and addressing any issues that may arise following the deployment.

6. Maintenance and Updates:

Once the software has been deployed, it is common for issues and bugs to arise. The dedicated team of developers actively works on identifying, fixing, and thoroughly testing these problems. Regular updates are provided to address any necessary improvements or changes that may arise

Feature enhancements are made to the software as user needs evolve or new requirements arise. Developers consistently implement new features and improvements in response to these changes.

Regular security updates are crucial to address vulnerabilities and protect the software from cyber threats.

7. Documentation:

The software developer provides user guides, manuals, and online help documentation to assist end-users effectively navigate its features.

Developers are responsible for creating technical documentation that outlines the architecture, code structure, and APIs of a system. This documentation is crucial in helping future developers comprehend and maintain the software.

Features of Software Development

Collaborative Nature: Software development is a collaborative process that involves a diverse group of professionals, including developers, designers, project managers, and stakeholders. Software project success is heavily dependent on effective communication and seamless teamwork.

Continuous Learning: In Software Development it's super important to keep learning because things are always changing. New ways of writing code, tools, and technologies are always popping up. To do well and keep up, programmers need to keep on learning and getting better at what they do. It's like an ongoing adventure of picking up new skills to stay on top of the game.

Problem-Solving: Developers play a crucial role as problem solvers. They actively identify and address issues, craft innovative solutions, and optimize code to achieve the desired outcomes. Problem-solving skills lie at the heart of the software development process.

Creativity: When Developers making computer programs, it's not just about following rules . There's also room for being creative. Coding needs a lot of attention to detail and clear thinking, but it's also a chance to let developers imagination run wild.

Quality Assurance: In development, ensuring the quality and reliability of the software is a crucial aspect. To ensure exceptional results, the development cycle includes stringent testing and quality assurance procedures.

Why is software development important?

Software development is critical because it creates the computer program and apps that we use every day, allowing things to run more smoothly and making our lives easier. It's like the hidden magic that makes technology work for us.

1. Enabling Technological Innovation

Software development plays a crucial role in technological advancements. Software developers are responsible for creating innovative smartphone applications, designing websites, or developing complex enterprise software.

2. Improved Efficiency

In various industries, software development plays a crucial role in automating tasks and processes. This automation leads to enhanced efficiency. Consider the business sector as an example. It utilizes software applications to streamline operations, effectively manage resources, and facilitate informed decision-making processes.

3. Adapting to Changing Needs

Software development offers the necessary flexibility and adaptability, allowing developers to continually update and modify software in response to evolving user needs, regulatory requirements, and business demands. This ability to adapt holds paramount importance in effectively navigating the rapid changes of the digital domain.

4. Global Reach

The internet has revolutionized connectivity by bridging gaps across continents. With the aid of software applications, both businesses and individuals can effortlessly tap into a worldwide audience, shattering geographical boundaries and unlocking boundless market potential.

Jobs that Require Software Development

The field of software development offers a wide range of career opportunities, each with its own set of responsibilities and specializations. Some of the key roles in the software development industry include:

Software Developer/Programmer: Software developers, also known as programmers, have the important task of writing code and developing applications to meet project requirements. They specialize in various areas such as web development, mobile app development, or back-end systems development. Their role involves ensuring that the software functions effectively and fulfills its intended purpose.

Front-End Developer: In the field of web development, a Front-End Developer is responsible for crafting the visual interface and enhancing user experience on websites and applications. Their expertise lies in utilizing HTML, CSS, and

JavaScript to design and implement visually compelling elements within software.

Back-End Developer: In the field of software development, there exists a crucial role known as the Back-End Developer. These talented individuals possess expertise in server-side programming, managing databases, and e-nsuring efficient server functionality. It is their responsibility to construct the underlying infrastructure

DevOps Engineer: The DevOps Engineer plays a crucial role in bridging the gap between development and IT operations. They facilitate a seamless proce-ss by automating deployment, testing, and monitoring of software. Their responsibilities encompass ensuring efficient development and deployment procedures.

Quality Assurance (QA) Engineer: The QA engineer is responsible for testing and ensuring the quality and functionality of software. They carefully design test cases, execute tests, and diligently report any defects to the development team.

Software Architect: The software architect is responsible for designing the overall structure and system of a software project. They make important high-level design decisions and establish the project's technical direction.

Product Manager: A Product Manager oversees the entire development process, from gathering requirements to deployment. They are responsible for defining project goals, prioritizing features, and ensuring that the final product aligns with business objectives.

Data Scientist/Engineer: Data scientists and engineers are experts in the manipulation and analysis of data. Their focus lies in creating data-driven applications and algorithms that benefit both businesses and research endeavors.

Cybersecurity Analyst: With the growing importance of cybersecurity, analysts in this field focus on securing software and systems against cyber threats and vulnerabilities.

Software development is a broad field that constantly evolves and shapes the modern world. Its impact is far-reaching, from user-friendly mobile apps to intricate business systems. By following a structured process, fostering creativity, and emphasizing quality assurance, developers drive the growth and adaptation of software solutions in our increasingly digital society. The diverse range of career opportunities within this industry provides passionate individuals with a chance to make a significant impact on the future of innovation and technology.

1.5 Life Cycle

The Software Development Life Cycle (SDLC) is a structured framework used by software organizations to design, develop, and test high-quality software. It defines the entire process of software production, from the initial idea to the final deployment and maintenance.

The goal of the SDLC is to produce high-quality software that meets or exceeds customer expectations, reaches completion within time and cost estimates, and is efficient to maintain.

Without a structured SDLC, software development becomes chaotic. Teams may miss requirements, blow budgets, or deliver buggy code that doesn't solve the user's problem. SDLC provides:

- Visibility: Stakeholders know exactly what is happening at every stage.
- Quality Control: Testing is integrated into the process, not an afterthought.
- Risk Management: Potential pitfalls are identified during the planning phase.
- Cost Estimation: Helps in predicting timelines and budgets accurately.

Stages of the Software Development Life Cycle

SDLC specifies the tasks to be performed at various stages by a software engineer or developer. It ensures that the end product is able to meet the customer's expectations and fits within the overall budget. Hence, it's vital for

a software developer to have prior knowledge of this software development process. SDLC is a collection of these six stages, and the stages of SDLC are as follows:

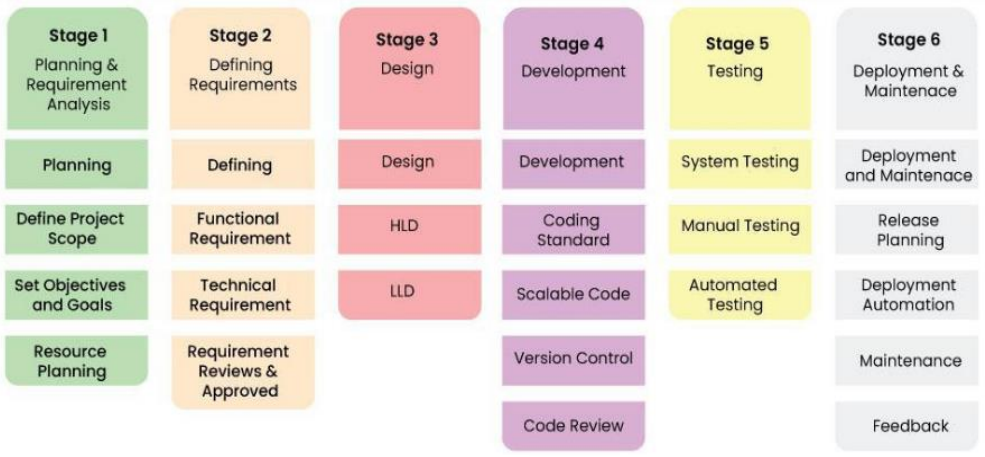


Figure 1.3

The SDLC Model involves six phases or stages while developing any software.

Stage 1: Planning and Requirement Analysis

This is the most fundamental stage. Before writing code, the team must define what they are building and why.

- Activities: Feasibility studies (technical, operational, economic), resource allocation, project scheduling, and cost estimation.
- Output: Project Plan, Feasibility Report.
- Key Players: Senior Engineers, Project Managers, Stakeholders.

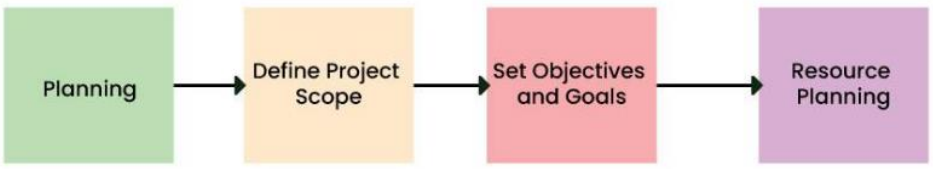


Figure 1.4

Stage 2: Defining Requirements (SRS)

Once the plan is approved, the specific product requirements must be defined and documented. This is done through the Software Requirement Specification (SRS) document.

- Activities: Gathering detailed functional and non-functional requirements from customers.
- Output: SRS Document (The "Bible" for the development team).
- Key Players: Business Analysts (BAs), Product Owners

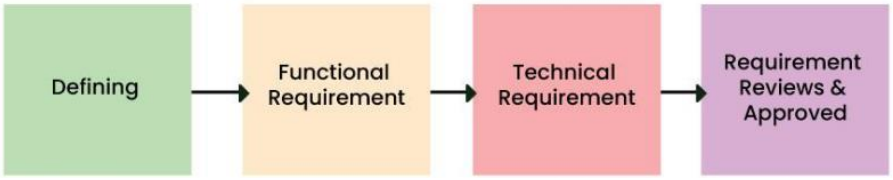


Figure 1.5

Stage 3: Designing Architecture

The requirements are turned into a technical blueprint. This phase defines the overall system architecture and technology stack.

- High-Level Design (HLD): Defines the architecture, database design, and relationships between modules.
- Low-Level Design (LLD): Defines the logic of individual components, API interfaces, and database tables.

Output: Design Document Specification (DDS).

Key Players: System Architects, Lead Developers.

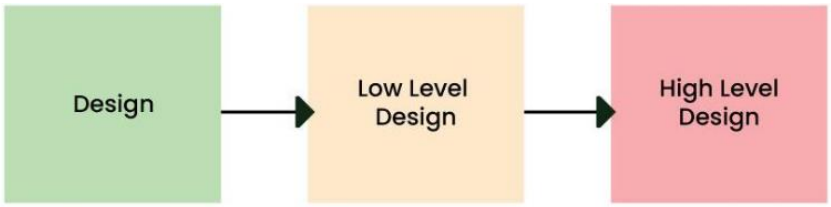


Figure 1.6

Stage 4: Development (Coding)

This is the longest phase where the actual building takes place. Developers write code based on the Design Document.

- Activities: Coding, Code Reviews, Unit Testing, and Static Code Analysis.
- Tools: Compilers, Debuggers, IDEs (VS Code, IntelliJ), Version Control (Git).

- Output: Source Code, Executable Software.
- Key Players: Developers (Frontend, Backend, Full Stack).

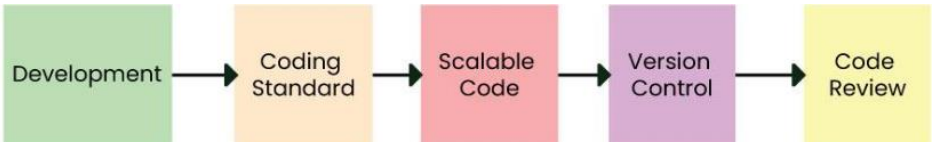


Figure 1.7

Stage 5: Testing

Once the code is written, it moves to the Quality Assurance (QA) team. The goal is to find bugs before the customer does.

Types of Testing:

- Unit Testing: Testing individual functions.
- Integration Testing: Ensuring modules work together.
- System Testing: Testing the entire application flow.
- User Acceptance Testing (UAT): Verifying the software meets business needs.

Output: Bug Reports, Test Cases, Quality Report.

Key Players: QA Engineers, Testers.

Figure 1.8

Stage 6: Deployment

The software is released to the end-users. In modern DevOps environments, this is often automated via CI/CD pipelines.

- Activities: Setting up production environments, deploying code, and smoke testing the live environment.
- Output: Live Application.
- Key Players: DevOps Engineers, Release Managers.

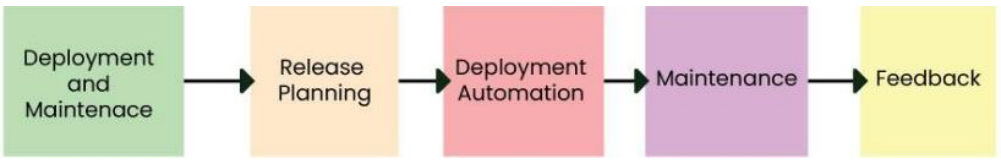


Figure 1.9

Stage 7: Maintenance

The cycle doesn't end at deployment. Software must be maintained to remain useful.

- Activities: Bug fixing, upgrading libraries, performance tuning, and adding new features.
- Output: Patches, Updates, New Versions.
- Key Players: Support Engineers, Developers.

Software Development Life Cycle Models

Software Development Models are structured frameworks that guide the planning, execution, and delivery of software projects. They define the sequence of development stages, such as requirements, design, coding, testing, and deployment.

Common Models

- Waterfall Model
- Agile Model
- V-Model
- Spiral Model
- Incremental Model
- RAD Model

What is the need for SDLC?

SDLC is a method, approach, or process that is followed by a software development organization while developing any software. SDLC models were introduced to follow a disciplined and systematic method while designing software. With the software development life cycle, the process of software design is divided into small parts, which makes the problem more understandable and easier to solve. SDLC comprises a detailed description or

step-by-step plan for designing, developing, testing, and maintaining the software.

How does SDLC Address Security?

A frequent issue in software development is the delay of security-related tasks until the testing phase, which occurs late in the software development life cycle (SDLC) and occurs after the majority of crucial design and implementation has been finished. During the testing phase, security checks may be minimal and restricted to scanning and penetration testing, which may fail to identify more complicated security flaws.

Security issue can be address in SDLC by following DevOps. Security is integrated throughout the whole SDLC, from build to production, through the use of DevSecOps. Everyone involved in the DevOps value chain have responsibility for security under DevSecOps.

Real Life Example of SDLC

Developing a banking application using SDLC:

Planning and Analysis: During this stage, business stakeholders' requirements about the functionality and features of banking application will be gathered by program managers and business analysts. Detailed SRS (Software Requirement Specification) documentation will be produced by them. Together with business stakeholders, business analysts will analyze and approve the SRS document.

Design: Developers will receive SRS documentation. Developers will read over the documentation and comprehend the specifications. Web pages will be designed by designers. High level system architecture will be prepared by developers.

Development: During this stage, development will code. They will create the web pages and APIs needed to put the feature into practice.

Testing: Comprehensive functional testing will be carried out. They will guarantee that the banking platform is glitch-free and operating properly.

Deployment and Maintenance: The code will be made available to customers and deployed. Following this deployment, the customer can access the online banking. The same methodology will be used to create any additional features.

1.6 Methodologies Paradigm and Practices

The Software Development Life Cycle (SDLC) is a structured, multi-step framework used by organizations to plan, design, build, test, deploy, and maintain high-quality software efficiently. It ensures that the software produced meets or exceeds customer expectations while staying within time and cost estimates.

Here is a comprehensive breakdown of the methodologies, paradigms, and practices of SDLC.

1. Core Phases of the SDLC (The Standard Process)

Regardless of the methodology, most SDLC processes include these fundamental stages:

Planning and Requirement Analysis: Gathering business requirements, feasibility studies (technical, operational, economic), resource allocation, and project scheduling.

Design (Architecture): Turning requirements into a technical blueprint (High-Level Design/Low-Level Design).

Development (Coding): The longest phase where developers build the software.

Testing: Quality Assurance (QA) team runs unit, integration, and system tests to find bugs before release.

Deployment: Releasing the software to users (often automated via CI/CD pipelines).

Maintenance: Fixing bugs, updating, and enhancing the system after launch.

2. SDLC Methodologies (Models & Approaches)

Different methodologies determine how the phases are ordered and executed.

The agile methodology

The agile methodology is an incremental and iterative approach that allows frequent changes in the project. The emphasis is mainly on flexibility and taking an adaptive approach while creating the software.

The agile model quickly became one of the most popular SDLC methodologies and an industry standard, sometimes even used in non-tech initiatives. The approach considers fast failure a good thing. It involves ongoing release cycles that consist of incremental changes from the previous release. Each iteration includes testing the product. The work is broken into segments called sprints. The agile model works great for projects that need flexibility and speed. Such projects are often found in startups and small organizations.

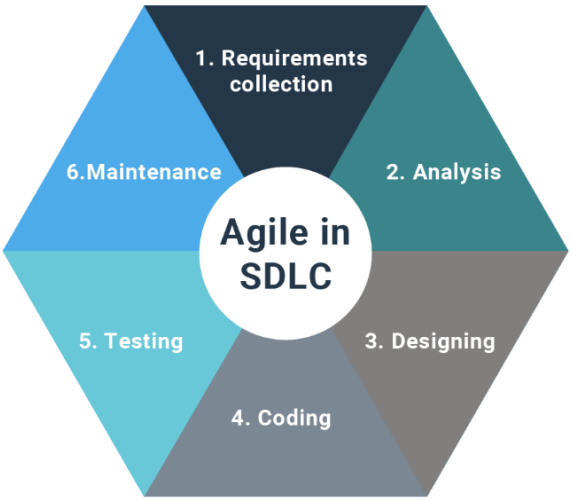


Figure 1.10

The key advantages of the agile model:

- It adapts to changes more quickly than other SDLC models. There's a focus on flexibility.
- The software can be released to customers after every sprint cycle, which is beneficial for the customer. Customers can provide feedback instantly so that changes to the requirements can be added at the later stages of the software development process.

- There is constant collaboration between teams and stakeholders throughout the SDLC.

The key disadvantages of the agile model:

- It may be difficult to estimate the total time, cost, and resources required. In addition, it's challenging to predict the final result.
- Documentation is kept to the bare minimum required by the development teams. It evolves with the project.
- The model needs experienced and highly skilled people.

The lean methodology

Lean software development is about working only on what must be worked on at the time. It's interconnected with agile – they both focus on flexibility and speed. This SDLC methodology is inspired by lean manufacturing practices: waste elimination, amplifying learning, late decision making, team empowerment, fast delivery, built-in integrity, and viewing the application as a whole. As opposed to the agile model, the lean model emphasizes the elimination of waste as a way to build more value for customers.

Figure 1.11

The key advantages of lean model:

- The model simplifies the software development process by removing unnecessary stages.

- It prioritizes essential functions so that developers spend less time on valueless builds.
- It delivers the product very early when compared to other methods.

The key disadvantages of lean model:

- The overall success strongly depends on the technical skills of team members.
- The model is not as scalable as others.

The iterative methodology

The iterative model puts emphasis on repetition. It evaluates the phases within the software development life cycle until the desired results are produced. This SDLC methodology is similar to the agile model except that there is less scope for external involvement.

The development team initializes the software requirements, then tests and evaluates them to establish any further requirements. Then, the design and implementation phases follow. A new version of the product is built with each iteration. Improvements are applied until the completed software is ready.

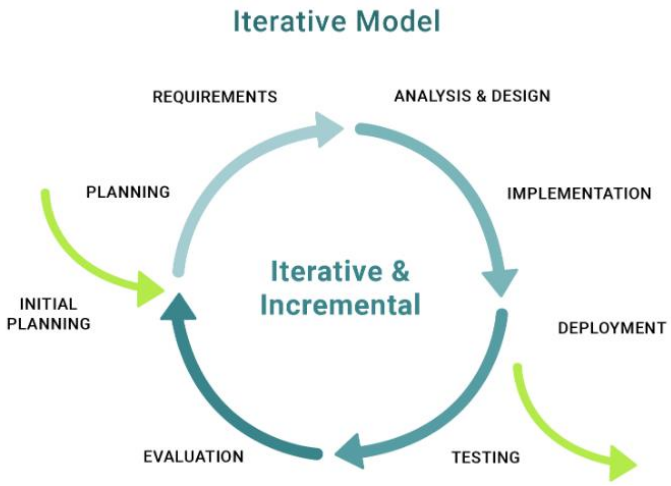


Figure 1.12

The key advantages of the iterative model:

- It’s easy (and less expensive) to implement changes.
- Mistakes and errors are detected early.
- A huge focus on testing and immediate modifications.

- Results are produced quickly and periodically.
- Product value is delivered in short time intervals

The key disadvantages of the iterative model:

- Repetitive processes can consume resources quickly.
- The end goal of the project may shift with each iteration.

The DevOps methodology

This is one of the newest methodologies. In this model, developers and operations teams collaborate closely to speed up innovation and the deployment of higher quality, as well as more reliable, software. The DevOps methodology's main goal is to unite development and operations to streamline delivery and support. The model reduces the organizational risk and makes changes more fluid.

In the DevOps model, updates to products are small but frequent. It's an entire philosophy that requires a non-traditional mindset in an organization.

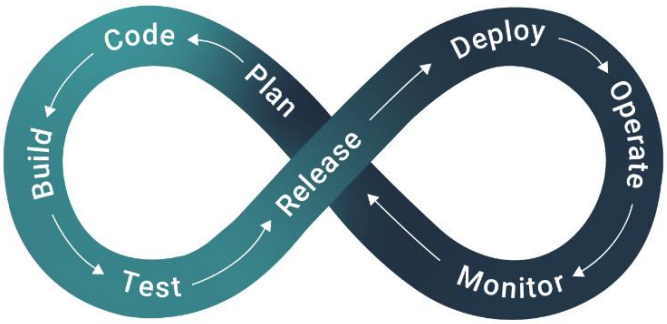


Figure 1.13

The key advantages of the DevOps model:

- It delivers fully-functional software within a short time.
- Cross-functional teams collaborate closely to ensure quick bug fixes at any stage of the development process.
- Workflow management is improved.
- The model enables the teams to build fully scalable and reliable applications.

The key disadvantages of the DevOps model::

- It requires experienced and skilled teams.
- The model often requires a huge investment (in both money and time).

The waterfall methodology

The waterfall model is the oldest SDLC methodology. It's a simple and straightforward approach with a rigid structure that needs to be strictly followed: once a phase is finished, you move on to the next without going back.

The model is easy to understand and simple to manage but there is no room for revisions. Once a stage is completed, issues won't be fixed until the maintenance stage starts. The waterfall methodology is also called the linear sequential model. This model is a strong solution for government contractors but not for projects that require flexibility in the long term.

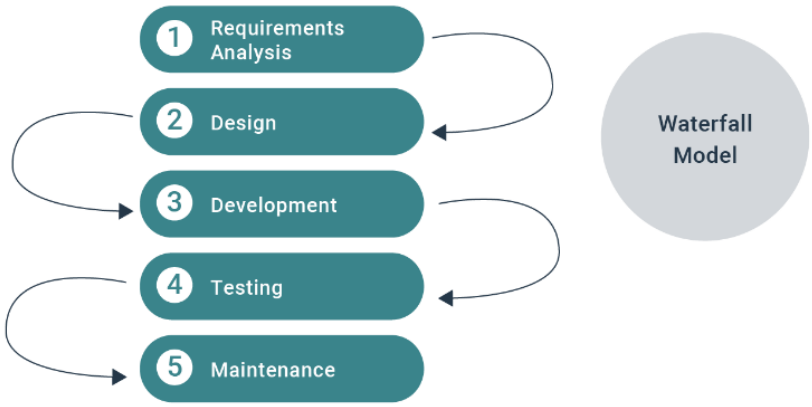


Figure 1.14

The key advantages of the waterfall model:

- It's simple and intuitive.
- All the phases are done bit by bit.
- Design errors can be identified in the analysis and design stages.
- It's easy to estimate the total cost.
- There is no complexity in execution.

The key disadvantages of the waterfall model:

- It only suits projects with clear and stable requirements.

- It’s time-consuming.
- Requirements cannot actually change.
- The planning phase is long and requires skilled professionals.

The prototyping methodology

In the prototyping model, development teams focus on building an early model of the software or application. A prototype is built before developing an actual, fully-functioning product. While a prototype is not fully functional or tested, it’s a great opportunity to get user feedback. The process of refining the prototype goes on until the customer is satisfied. Then, the actual product is built by using the prototype as a reference.

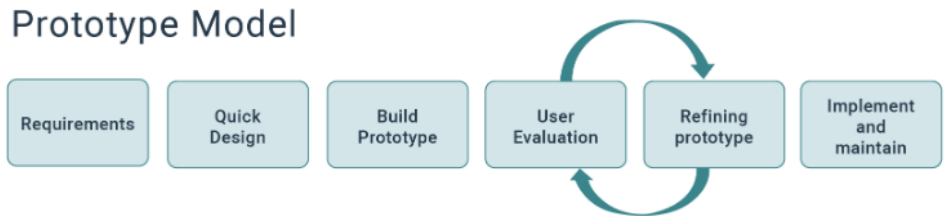


Figure 1.15

The key advantages of the prototyping model:

- It has a flexible design.
- The total cost and time are reduced since potential risks are identified in the prototype.
- Missing features or needed changes can be easily implemented while creating a refined prototype.
- The model ensures customer satisfaction.

The key disadvantages of the prototyping model:

- It’s easy for the customer to change the requirements of the end product.
- Documentation isn’t clear because requirements keep changing.
- The model can increase the complexity of the product.

The spiral methodology

The spiral model is one of the most flexible SDLC models. It's very often chosen by organizations that aren't sure about their requirements or modifications during the risk analysis.

The spiral model is similar to the iterative model – it passes through four stages (planning, risk analysis, development, and evaluation) that allow for multiple rounds of refinement. The spiral methodology mainly focuses on risk identification – potential risks are identified by developers and solutions are implemented to avoid and mitigate them.

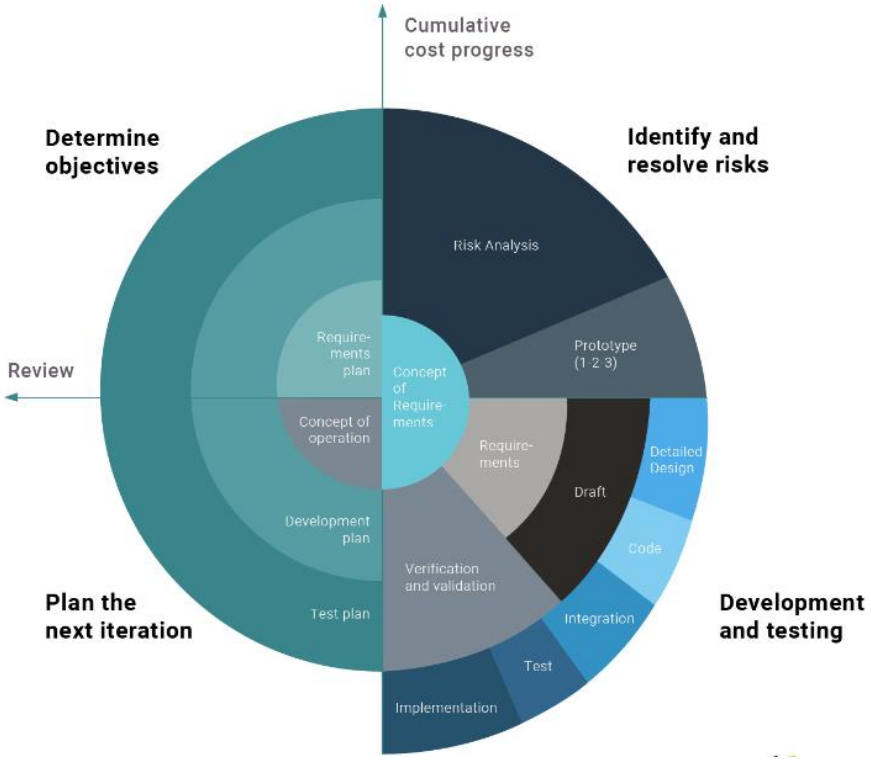


Figure 1.16

The key advantages of the spiral model:

- The spiral methodology perfectly suits large projects, especially those that require risk analysis and changes in requirements.
- End-users can see the product at the early stages.
- Thanks to the detailed analysis, risks are less likely to occur.

The key disadvantages of the spiral model:

- The model may seem to be complicated.
- It's not suitable for small and simple projects.
- Sometimes, it takes time to reach the final product, especially when there are many iterations.

1.7 Software Development Paradigms

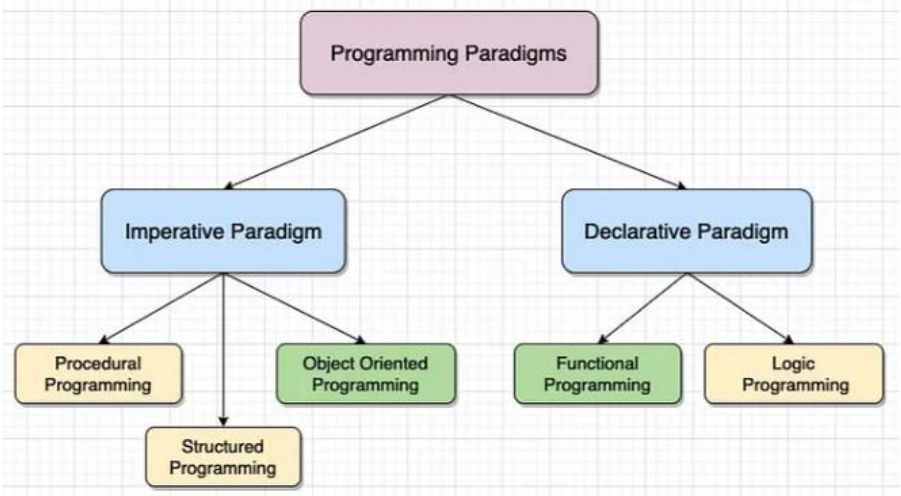


Figure 1.17

A programming paradigm is a fundamental approach or style of programming that provides a set of principles, concepts, and techniques for designing and implementing computer programs. It defines the structure, organization, and flow of the code, as well as the methodologies for problem-solving and expressing computations.

Programming paradigms dictate how programmers should think about and structure their code. They influence the way programs are written, the techniques used to solve problems, and the overall design philosophy. Different paradigms have their strengths and weaknesses, and choosing the right paradigm for a given task can greatly impact the efficiency, maintainability, and scalability of a program.

Each programming paradigm has its own set of concepts and features. For example, procedural programming focuses on procedures and functions,

object-oriented programming revolves around objects and classes, functional programming emphasizes immutability and pure functions, and so on. These paradigms provide guidelines and best practices for organizing code, managing data, controlling program flow, and solving specific types of problems.

Procedural Programming:

Procedural programming is a paradigm where the program is structured around procedures or functions that manipulate data. It focuses on step-by-step instructions and emphasizes code reusability through the use of functions. C and Pascal are examples of languages that follow this paradigm.

Object-Oriented Programming (OOP):

Object-Oriented Programming revolves around the concept of objects, which are instances of classes. It organizes code into objects that encapsulate data and behavior. OOP promotes modularity, reusability, and allows for concepts such as inheritance, polymorphism, and encapsulation. Java, C++, and Python are popular languages that support OOP.

Functional Programming (FP):

Functional Programming treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes immutability, pure functions, and higher-order functions. FP languages focus on expressing computations as the evaluation of expressions. Haskell, Lisp, and Erlang are examples of languages that follow this paradigm.

Declarative Programming:

Declarative Programming focuses on describing the desired result rather than specifying the detailed steps to achieve it. It deals with what should be computed rather than how. SQL, a language used for database querying, is a prominent example of a declarative language.

Event-Driven Programming:

Event-Driven Programming is based on the concept of events and event handlers. It involves programming the flow of a program based on events and

the reactions to those events. GUI programming and asynchronous programming often use this paradigm. JavaScript is a language that prominently supports event-driven programming.

Concurrent Programming:

Concurrent Programming is concerned with handling multiple tasks that run simultaneously and potentially interact with each other. It focuses on managing shared resources, synchronization, and communication between concurrent processes or threads. Languages like Java and Go provide built-in support for concurrent programming.

Note that these paradigms are not mutually exclusive, and many languages incorporate elements of multiple paradigms. The choice of paradigm depends on the problem domain, language capabilities, and personal preferences of the programmer.

1.8 Modern SDLC Practices and Best Practices

Modern Software Development Life Cycle (SDLC) practices focus on speed, quality, and security through automation, incorporating DevOps, CI/CD, and AI-driven workflows. Best practices include "shifting left" (early security/testing), using small, iterative releases, and maintaining high visibility via observability, which reduces risks and improves deployment frequency.

Key Modern SDLC Practices

DevSecOps & Security Integration: Integrating security early in the lifecycle (shifting left) using SAST, DAST, and software composition analysis (SCA) to identify vulnerabilities immediately.

CI/CD Pipelines: Automating builds, testing, and deployment to make the release process repeatable and reduce human error.

AI Integration: Utilizing AI beyond code generation for requirement analysis, automated code reviews, and documentation, with human oversight.

Git Branching Strategies: Using feature-branch workflows (e.g., GitFlow, trunk-based development) to isolate work and enable preview environments.

Observability & Telemetry: Implementing real-time monitoring and logging to catch issues early, shifting from reactive to proactive maintenance.

Small, Iterative Releases: Preferring frequent, incremental updates to shorten feedback loops and reduce deployment risk.

SDLC Best Practices for High-Performing Teams

Measure Performance: Utilize metrics like lead time, change failure rate, and developer satisfaction to drive improvement, rather than just tracking velocity.

Implement Threat Modeling: Proactively identify security threats during the design phase.

Standardize Code Reviews: Use automated tools and checklists to maintain code quality and security standards.

Manage Technical Debt: Treat technical debt as actionable backlog work, not as an afterthought.

Create Temporary Environments: Use preview environments for each branch/pull request to improve testing accuracy.

Foster Collaboration: Break down silos between development, security, and operations teams.

These practices collectively enable teams to deliver software faster while maintaining high-quality standards.

1.9 Project Planning Process

Once a project is found to be possible, computer code project managers undertake project design. Project designing is undertaken and completed even before any development activity starts. Project designing consists of subsequent essential activities: Estimating the subsequent attributes of the project:

- Project size: What's going to be the downside quality in terms of the trouble and time needed to develop the product?
- Cost: What proportion is it reaching to value to develop the project?
- Duration: How long is it to reach design plate amended development?

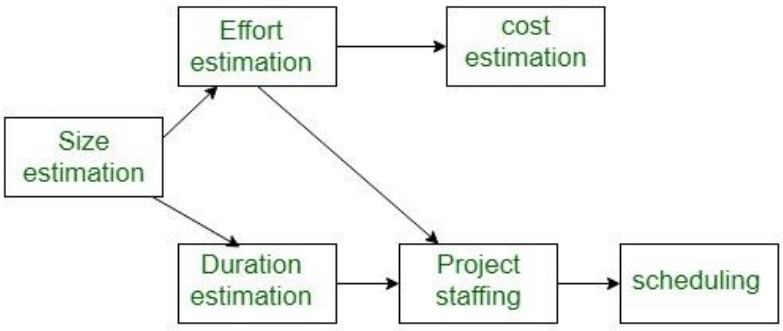
- Effort: What proportion of effort would be required?

The effectiveness of the following design activities relies on the accuracy of those estimations.

- planning force and alternative resources
- workers organization and staffing plans
- Risk identification, analysis, and abatement designing
- Miscellaneous arrangements like quality assurance plans, configuration, management arrangements, etc.

Precedence ordering among project planning activities:

The different project-connected estimates done by a project manager have already been mentioned. The below diagram shows the order in which vital project coming up with activities is also undertaken. It may be simply discovered that size estimation is the 1st activity. It's conjointly the foremost basic parameter supported that all alternative coming up with activities square measure dispensed, alternative estimations like the estimation of effort, cost, resource, and project length also are vital elements of the project coming up with.



Precedence ordering among planning activities

Figure 1.18

Sliding Window Planning:

Project designing needs utmost care and a spotlight since commitment to unrealistic time and resource estimates ends in schedule slippage. Schedule

delays will cause client discontent and adversely affect team morale. It will even cause project failure. However, project designing could be a difficult activity. particularly for giant comes, it's pretty much troublesome to create correct plans. A region of this issue is thanks to the fact that the correct parameters, the scope of the project, project workers, etc. might be amended throughout the project. So as to beat this drawback, generally project managers undertake project design little by little. Designing a project over a variety of stages protects managers from creating huge commitments too early. This method of staggered designing is thought of as window designing. Within the window technique, beginning with the associate initial setup, the project is planned additional accurately in sequential development stages. At the beginning of a project, project managers have incomplete information concerning the main points of the project. Their info base step by step improves because the project progresses through completely different phases. When the completion of each section, the project managers will set up every ulterior section accurately and with increasing levels of confidence.

1.10 Software Project Estimation

Software project estimation in software engineering is a crucial process that involves time and budget predictions of a project. In addition, it deep dives into the experience of the software development company and the techniques they use to define the perfect way to develop and finish the project.

For that, complex software estimation tools and mathematical knowledge help craft a suitable plan. Project estimation may be a cost for the company at the initial stage. However, this must be considered as a stepping stone for creating a final result that should be more realistic, credible, and customer-centric. A well-planned software project estimation will help companies address several pain points through these actionable tasks:

- Framing a detailed task sheet of what should be done and who will be responsible for every task

- Building a solid task force that should be adequate to complete the project
- Determining the estimated budget of the project cost centres, cost control areas, software development cost estimation, and much more
- Drawing the estimated deadline of the project

We need to consider the following six elements of a project to ensure the use of project estimation techniques will prove beneficial:

Cost Of Software Project

Cost is one of the primary elements when estimating software project management. Lack of adequate funds may hamper the project continuity. In such a scenario, software project estimation techniques will enhance clients' confidence, ensuring the efficient use of resources and materials to complete the work. Estimating costs will give a clear picture of the fund usage at present and future.

Project Requirement Size Or Scope

The second crucial project constraint is the scope of the project. You must analyze the total number of tasks needed to complete the project on time. By doing this, you can ensure the materials and expertise required for the project estimation and the amount of work involved. This is where IT Project Management Consulting Services helps you establish the project requirement size with expert guidance based on the project's scalability and complexity.

A perfect understanding of the scope and schedule of the project will increase the chances of developing an accurate budget estimation.

Software Development Time

The following important constraint is to address the unavailability of sufficient time. Plan a project by estimating the overall project duration and timelines of each task.

This also includes planning the workforce and other resources that you may need to create a solution at the precise time, as per software development

trends. Your client satisfaction ratio will also improve as you are ready with the actionable tasks to provide key deliverables on time.

Number Of Resource Requirement

Project resources such as people, tools, subcontractors, materials, software, and others help in project completion. Managing all these resources efficiently, which is one of the key benefits of custom software development, needs a lot of knowledge about the work dynamics as you cannot afford to keep any of these resources idle for a long time.

Common Risks In Software Projects

Taking project risks can either turn into a positive or a negative impact. When you estimate your risk-taking ability, you visualize the events that will occur during the ongoing project and the seriousness of taking the risk. Analyze the potential issues that may come up and draw a risk management plan that will prevent hampering your project by taking unwanted risks.

Project Deliverables

Project estimation includes designing an ideal plan that meets quality standards and complies with environmental regulations. You may need additional money, time, and resources to adopt and implement your project as per the standards.

However, this will prove better than practising the lower standards. Ensuring the best product quality is available to the clients will call for a quality-level estimation plan. When you implement all the above six elements in all your software project estimation techniques, you will observe an enhancing level of accuracy in the workflows.

Software Project Estimation Techniques

Different projects have different challenges. No solution available spot on can address the potential issues in a software development project. Developing software evolves programmers constantly and teaches them new technologies and innovations.

Let’s understand the most outstanding types of software development estimation techniques:

Top-Down Estimate

Your entire project development process breaks down into several tasks and phases and works according to the breakdown structure. We use the Top-Down Estimating technique for clients with a limited duration for the project completion. We estimate the overall timeline, dividing it into project activity tasks and giving deadlines to each.

Bottom-Up Estimate

In contrast to the top-down estimate, this technique focuses on individual tasks or smaller yet crucial aspects of the project. All the individual estimates are combined, creating an overall project estimate. The bottom-up is one of the type of estimates that provides more accurate results than the top-down estimate.

Even if the process is lengthier and more time-consuming, the project managers’ and business analysts’ efforts prove their worth during implementation.

Expert Judgment

The most popular project estimation technique is expert judgment. It is known for providing quick estimation results with simple methods. Many project experts and specialists design software project estimations based on their past experiences and intuition. This works only if your team has worked on similar projects. Even top-down and bottom-up can provide expert judgment.

Parametric Model Estimating

The technique fetches crucial data from past project details and compares each project to draw differences between them. After pro-rating the previous project details, the current project is then estimated.

Comparative Or Analogous Estimation

With comparative estimation and a top-down approach, we estimate the project duration based on past project data. If we find similar projects done in

the past that took eight months to complete, we estimate the current project timeline with the same amount of time. We further find the lower-level work estimates, spreading each task and activity into eight months.

Three-Point Estimating

Three-point estimating generates bottom-up estimates by assigning three durations to a task instead of one. They are either optimistic, pessimistic, or most likely. We average out these three numbers to calculate the actual estimate.

We also use the PERT (Program Evaluation and Review Technique) method for three-point estimation. Generally, the weighted average method is used, and more weightage is given to "most likely". This method reduces the chances of an inflated estimate.

1.11 Decomposition techniques

The function decomposition works by simplifying the complex types of software systems. We can also identify the functions of the different types of components that we may have to develop for designing complex systems. The engineers or team members use this method to understand the requirements easily. In simple terms, function decomposition is just a process of tearing apart a system into interrelated parts. The main objective of this is to identify various functions and then understand how they interact with other functions so that we can successfully decompose the functions that are large and complex.

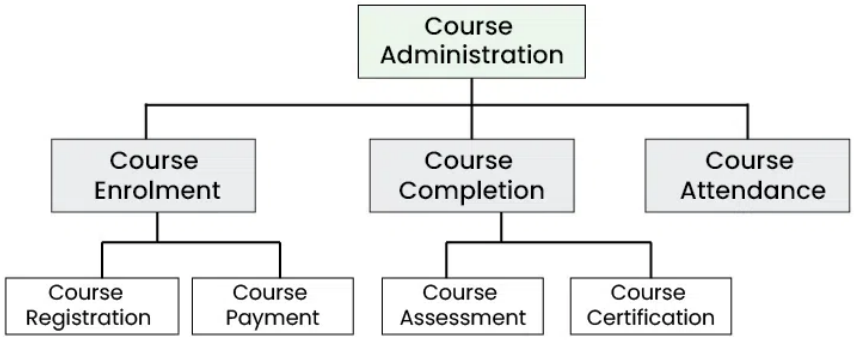


Figure 1.19

It involves various steps so that whenever we want to turn a complex system into a simple one we can use these steps and successfully decompose a particular function now.

Steps in Function Decomposition

In software engineering whenever we have complex systems or large systems that we want to decompose into small subsystems or modules or if we want to turn complex systems into simple systems, then we can use function decomposition. The following are the steps that are required in any function decomposition:

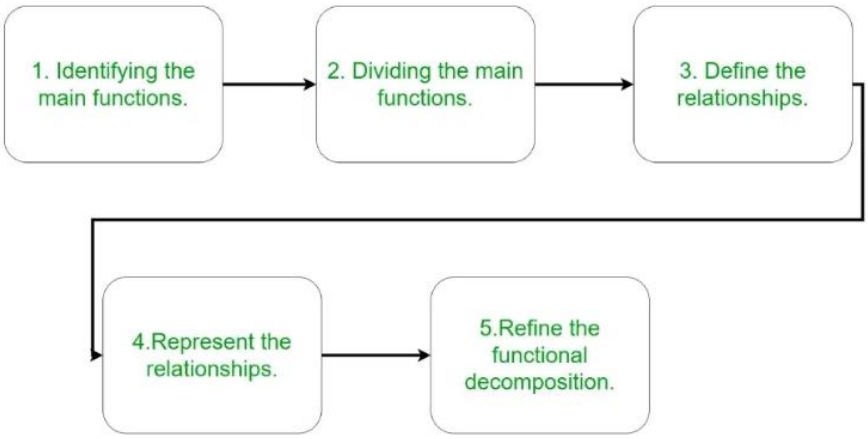


Figure 1.20

1. Identifying the Main Functions

This is the first step in function decomposition. In this system, we try to identify the malfunctions of the system and what this means in this step, we try to understand what the system does and what the working it is supposed to after this we move to the next step.

Identifying the main functions means that we try to figure out exactly what a system is supposed to do and also figure out how it works. It means understanding the particular job of the system. For example, if we talk about a car the main function of a car can include moving forward stopping and turning etc similarly we try to identify the main functions for this method as well.

2. Dividing the Main Functions

In this step, we try to divide the main functions of the system and we try to divide the complex main functions so that we can convert them into simple sub-functions then we move to the next step.

Once we understand the first step and know what the system is we break down the main jobs into two tasks which are smaller and simpler. The main reason behind this step is that it helps us to understand the functions more easily as we understood above step by an example of a car in this step we can break down the moving forward task into smaller tasks such as a break-in accelerating or steering etc.

3. Define the Relationships

In this step, we define the relationship among the various subfunctions and the main function, this helps us to understand how the various functions are connected and interact together so that we can achieve the end goal of the system.

So now we have defined the malfunctions and also managed to break the tasks into smaller functions so now that we have smaller tasks we can look at how they have connected the reason for defining this relationship is because this helps us to see how everything works together to achieve the goal of the system.

4. Represent the Relationships

After defining the relationships, we try to represent the relationship among the functions so that we can have a pictorial view and understanding of the relationship between the functions.

In this, we make a picture or a simple diagram that can be used to show how all of the tasks are connected. This process is important because it helps us to visualize how the system is functioning, after this step we move to the final step of function decomposition.

5. Refine the Functional Decomposition

This is the final step of the function decomposition. In this step, we review the system's functional block diagram and we also try to add some modifications according to our needs so that the system and its functions do the work that they are supposed to.

In this step, we also review our diagram and then we make any necessary changes that are required to make sure that the system and its tasks are doing exactly what they are intended to do. After this refining of the functional decomposition, we can say that we have successfully decomposed a particular system.

Applications of Function Decomposition

Now that we have understood the steps which are involved in function decomposition, also take a look at the applications of function decomposition. The Following are the applications of function decomposition in software engineering:

Software Design: Function decomposition is very important during the design phase of software development because it helps the software developers break down the system and its requirements into smaller modules that are easy to handle.

Modular Programming: The function decomposition also helps in modular programming. In this software systems are divided into independent models where each module is responsible for a specific function.

Code Reusability: Function decomposition also promotes code reusability by breaking the complex functions of the system into smaller and simpler modules.

Testing and Debugging: Decomposing the functions into smaller units helps in the easier testing and easier debugging of the software. When we have smaller testing, debugging can be done more efficiently.

Performance Optimization: It also helps in the performance optimization of the system because when it breaks down the complex algorithms or tasks into smaller tasks it helps the developers to identify bottlenecks easily.

Collaborative Development: It helps us to achieve collaborative development because it divides the software into small components which means that different team members can work on the separate components together and helps in the reduction of dependencies so overall it allows the software development team for parallel development.

Maintenance and Updates: The function decomposition helps in breaking the module into smaller modules which helps us to maintain and update the system more easily because by decomposing the functions the developers can easily focus on the specific modules whenever they try to make a change. This helps to enhance the overall maintenance of the software.

Advantages of Function Decomposition

Function decomposition has many advantages in software engineering. Some of the advantages of using function decomposition in software engineering are:

Simplified Complexity: By breaking a system into smaller functions, we also reduce the overall complexity of the system because by following this method we focus on functions that are smaller and manageable. This helps the developers to understand the system in a much better way which helps us to simplify the complexity of the system.

Improved Maintainability: By using the function decomposition, we decompose the functions into smaller and more well-defined modules. This helps us to enhance the maintainability of the software because the software developers can make changes to the specific modules without having to look at the other parts of the system. This is more helpful because it improves maintainability.

Supports Reusability: It helps in the reusability of code because it breaks down the complex functions into two modules which are more reusable. This means

that the developers can use these modules for different projects or different parts of the software and save time in the development of the software.

Supports Scalability: It also helps in support of scalability because it allows the software systems to easily scale at a faster rate because software and its functions are organized into smaller modules which helps in adding new features or scaling the system to support the increased usage more easily.

Disadvantages of Function Decomposition

As we discussed some of the advantages of the function decomposition, there are some disadvantages also, present in the function decomposition:

Increased Complexity in Coordination: As we discussed function decomposition helps in simplifying each function but one of its drawbacks is that it can sometimes lead to increased complexity in coordinating the interaction between the multiple modules so in some cases it can cause more complexity.

Difficulty in System Understanding: The function decomposition may make it challenging for the team to understand the system because when the functions are broken into smaller modules then it becomes harder and more challenging to understand the overall structure as well as the overall flow of the system. So, it can require more time in understanding of the system because of this method.

Increased Testing Complexity: It helps by breaking down the functions into smaller units but it can also increase the testing complexity as testing the interaction between different modules can require extra effort which can also slow down the testing process and leads to an increase in the complexity of testing for the system.

Performance Overhead: By decomposing the functions into smaller modules, we may have a problem with performance overhead because when a function increases, the inter-module communication between these functions also increases. In some cases, this performance overhead may be negligible but

when we have a system that runs on tight resource then it becomes a drawback.

Function decomposition is an important concept in software engineering. It helps in breaking down the complex tasks into simpler tasks and make the work easy. So in conclusion, we have understood how function decomposition works in software engineering.

1.12 Empirical estimation models

Cost estimation simply means a technique that is used to find out the cost estimates. The cost estimate is the financial spend that is done on the efforts to develop and test software in Software Engineering. Cost estimation models are some mathematical algorithms or parametric equations that are used to estimate the cost of a product or a project. Various techniques or models are available for cost estimation, also known as Cost Estimation Models.

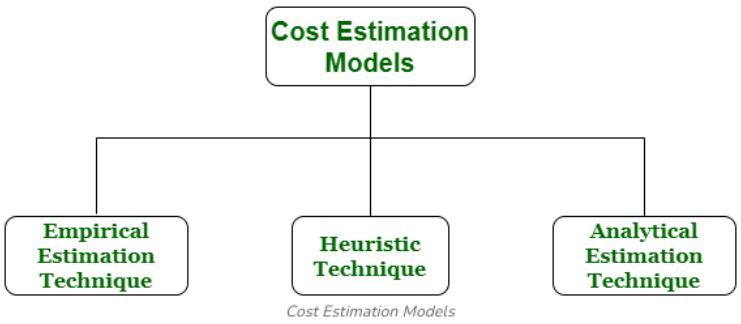


Figure 1.21

Empirical Estimation Technique - Empirical estimation is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step. These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions. It uses the size of the software to estimate the effort. In this technique, an educated guess of project parameters is made. Hence, these models are based on common sense. However, as there are many activities involved in empirical estimation

techniques, this technique is formalized. For example Delphi technique and Expert Judgement technique.

Heuristic Technique - Heuristic word is derived from a Greek word that means "to discover". The heuristic technique is a technique or model that is used for solving problems, learning, or discovery in the practical methods which are used for achieving immediate goals. These techniques are flexible and simple for taking quick decisions through shortcuts and good enough calculations, most probably when working with complex data. But the decisions that are made using this technique are necessary to be optimal. In this technique, the relationship among different project parameters is expressed using mathematical equations. The popular heuristic technique is given by Constructive Cost Model (COCOMO). This technique is also used to increase or speed up the analysis and investment decisions.

Analytical Estimation Technique - Analytical estimation is a type of technique that is used to measure work. In this technique, firstly the task is divided or broken down into its basic component operations or elements for analyzing. Second, if the standard time is available from some other source, then these sources are applied to each element or component of work. Third, if there is no such time available, then the work is estimated based on the experience of the work. In this technique, results are derived by making certain basic assumptions about the project. Hence, the analytical estimation technique has some scientific basis. Halstead's software science is based on an analytical estimation model.

Other Cost Estimation Models

Function Point Analysis (FPA): This technique counts the number and complexity of functions that a piece of software can perform to determine how functional and sophisticated it is. The effort needed for development, testing and maintenance can be estimated using this model.

Putnam Model: This model is a parametric estimation model that estimates effort, time and faults by taking into account the size of the the programme,

the expertise of the development team and other project-specific characteristics.

Price-to-Win Estimation: Often utilized in competitive bidding, this model is concerned with projecting the expenses associated with developing a particular software project in order to secure a contract. It involves looking at market dynamics and competitors.

Models Based on Machine Learning: Custom cost estimating models can be built using machine learning techniques including neural networks, regression analysis and decision trees. These models are based on past project data. These models are flexible enough to adjust to changing data and project-specific features.

Function Points Model (IFPUG): A standardized technique for gauging the functionality of software using function points is offered by the International Function Point Users Group (IFPUG). It is employed to calculate the effort required for software development and maintenance.

1.13 The make/buy decision

A make-or-buy decision is the act of choosing between manufacturing a product in-house or purchasing it from an external supplier. Also referred to as an outsourcing decision, a make-or-buy decision compares the costs and benefits associated with producing a necessary good internally to the costs and benefits of hiring an outside supplier to provide the resources in question.

To compare costs accurately, a company must consider all aspects of the acquisition and storage of the items versus creating them in-house, which may require the purchase of new equipment, as well as added labor costs.

Analyzing the Make-or-Buy Decision Process

In considering in-house production, a company must take into account any expenses related to the purchase and maintenance of new machinery or other equipment as well as the cost of the necessary raw materials. Making a product includes labor costs like wages and benefits, storage, holding costs,

and disposing of leftovers. Still another consideration is whether the business can produce what it needs at the required levels.

Buying costs include the product's price, shipping, import fees, and sales taxes. Companies also need to consider storage and labor costs for incoming products. Another consideration involves the signing of contracts with suppliers that lock in the prices it must pay for a certain period of time, sometimes to the company's benefit and sometimes not. In larger companies, these decisions are often rely heavily on the expertise of a chief procurement officer.

Factors to Consider When Choosing to Make or Buy

The results of a quantitative analysis may be sufficient to make a determination as to which approach is more cost-effective. At times, a qualitative analysis is also necessary to address any concerns a company might have that it cannot measure specifically.

For example, factors that may influence a company's decision to buy a part rather than produce it internally include a lack of in-house expertise, small volume requirements, a desire for multiple sourcing, and the fact that the item may not be critical to the firm's strategy or its need to differentiate itself from competitors.

A company may give additional consideration to buying if it has the opportunity to work with a company that has previously provided outsourced services successfully and can sustain a long-term relationship.

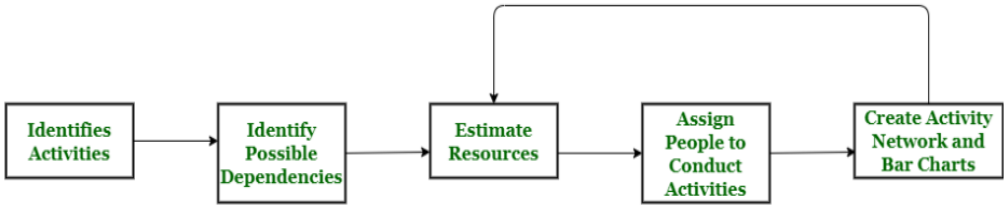
Similarly, factors that may tilt a company toward making an item in-house include existing idle production capacity, the potential for better quality control, or proprietary technology that needs to be protected. A company might also consider concerns regarding the reliability of the supplier, especially if the product in question is critical to normal business operations.

Make-or-buy decision is one of the key techniques for management practice. Due to the global outsourcing, make-or-buy decision making has become popular and frequent.

Since the manufacturing and services industries have been diversified across the globe, there are a number of suppliers offering products and services for a fraction of the original price. This has enhanced the global product and service markets by giving the consumer the eventual advantage. If you make a make-or-buy decision that can create a high impact, always use a process for doing that. When such a process is followed, the activities are transparent and the decisions are made for the best interest of the company.

1.14 Project scheduling

A schedule in your project's time table actually consists of sequenced activities and milestones that are needed to be delivered under a given period of time. Project schedule simply means a mechanism that is used to communicate and know about that tasks are needed and has to be done or performed and which organizational resources will be given or allocated to these tasks and in what time duration or time frame work is needed to be performed. Effective project scheduling leads to success of project, reduced cost, and increased customer satisfaction. Scheduling in project management means to list out activities, deliverables, and milestones within a project that are delivered. It contains more notes than your average weekly planner notes. The most common and important form of project schedule is Gantt chart.



Project Scheduling Process

Figure 1.22

Process:

The manager needs to estimate time and resources of project while scheduling project. All activities in project must be arranged in a coherent sequence that

means activities should be arranged in a logical and well-organized manner for easy to understand. Initial estimates of project can be made optimistically which means estimates can be made when all favorable things will happen and no threats or problems take place. The total work is separated or divided into various small activities or tasks during project schedule. Then, Project manager will decide time required for each activity or task to get completed. Even some activities are conducted and performed in parallel for efficient performance. The project manager should be aware of fact that each stage of project is not problem-free. Problems arise during Project Development Stage:

- People may leave or remain absent during particular stage of development.
- Hardware may get failed while performing.
- Software resource that is required may not be available at present, etc.

The project schedule is represented as set of chart in which work-breakdown structure and dependencies within various activities are represented. To accomplish and complete project within a given schedule, required resources must be available when they are needed. Therefore, resource estimation should be done before starting development. Resources required for Development of Project:

- Human effort
- Sufficient disk space on server
- Specialized hardware
- Software technology
- Travel allowance required by project staff, etc.

Advantages of Project Scheduling:

There are several advantages provided by project schedule in our project management:

- It simply ensures that everyone remains on same page as far as tasks get completed, dependencies, and deadlines.

- It helps in identifying issues early and concerns such as lack or unavailability of resources.
- It also helps to identify relationships and to monitor process.
- It provides effective budget management and risk mitigation.

1.15 Risk Management

Managing risk in software engineering is about equipping teams to respond to uncertainty. It means identifying potential barriers (technology, people, external) to success, gauging the risks' possible severity, and determining how to mitigate them. The goal of risk management is not to eliminate the risk claims (which is impossible) but produce projects that are resilient enough to deal with interruptions without being derailed.

Risks in software engineering come in different forms; therefore, every risk will require a somewhat different management approach:

- Security and Trust Risk - Cyberattacks, data breaches, or compromised open-source libraries threaten user trust and compliance.
- Technical Risk - Technical debt, vendor lock-in, and systems that adopt AI models (which no one can explain or understand).
- People and Process Risk - Skills gaps, misalignment from remote teams, and "cargo cult" agile practices that sound good on paper, but don't work in reality.
- Ethical and Compliance Risk - Conflicting regulations, biased algorithms, and the push toward greener systems.
- Business and Operational Risk - System outages, nightmares integrating legacy systems, and poor user experience that can hurt your brand image.

Why is Risk Management Important?

In software engineering, it is rare that projects go exactly as expected. New requirements emerge, systems react unexpectedly, and resources do not

always come together at the right time. This is where risk management in software engineering stands out.

1. Increases Project Efficiency

Risk management in software project management allows teams to uncover risks and their root causes so they can create response plans and resource allocation plans before risks become issues; less rework, fewer delays, and improved delivery is the direct result.

2. Encourages Communication & Collaboration

Risk management in project management encourages the team to share its insights on risk freely. As a result, transparency is created, priorities can be aligned, and decisions can be made faster. In the end, it is to create a culture of shared responsibility.

3. Improves Project Outcomes

Proactive risk management helps reduce schedule overruns, budget overruns, and scope creep. That means your software engineering team can deliver to the requested time, budget, with higher satisfaction to stakeholders.

4. Improves Software Quality

Risk management in software engineering mitigates risks related to performance, usability, and reliability early, ensuring you have a well-functioning and usable product.

5. Promotes Long-Term Resiliency

With a structured risk process consisting of risk identification, assessment, and mitigation, organizations stand poised to defend against unanticipated risks, regulatory fines, and loss of reputation.

What is The Risk Management Process In Software Engineering?

Features, deadlines, technologies, and people all play within the mix of creating value, or confusion. For this reason, a well-defined process for managing risks is essential for software engineering today. Rather than treating these risks as unwanted surprises, this process can be used to help

teams prepare, adapt, and turn these risks into opportunities to reinforce their work.

Let's take a look at the processes for managing risk in software engineering:

1. Risk Identification

This first step is easy, but powerful: just name the risk. In the process of risk management in software engineering, this to explore the possible failure points across the technology, people, and processes. Security holes, scope creep, sudden staff departures, and integration challenges, are all considerations.

2. Risk Assessment

Risks are not created equally. Some risks are speed bumps, while others are brick walls. Assessing risks denotes ranking them by probability and consequence. For instance:

- A small bug in a non-critical feature = low risk.
- A potential vulnerability in the security of payment processing = high risk.

Whatever the nomenclature, most software teams will assign a score or bucket (low, medium, high) to clarify priorities. By prioritizing risk upfront, teams can ensure they avoid spreading themselves too thin and focus their energy on what will actually derail delivery.

3. Risk Planning

This is where the plan comes into play. After you have identified the most important risks, you have to determine how to respond to the risks. Common responses to risk include:

- Avoiding the risk: change the plan to eliminate the risk.
- Mitigating the risk: do something to reduce the impact or probability.
- Transferring the risk: outsource or share the risk (e.g., using a cloud provider for infrastructure, etc.).
- Accepting the risk: live with the risk if it is minor or unavoidable.

This could look like adding automated tests to mitigate against quality risks, obtaining backup vendors to avoid dependency risks, or simplifying the scope to avoid overloading the team with work.

4. Risk Monitoring

Risk management isn't a "one-and-done" checklist. Once identified, risks need continuous attention. Circumstances change—new technologies, updated regulations, or even a competitor's move can introduce new risks mid-project.

What is The Difference Between People Risks vs. Operational Risks?

When you think about risk in software engineering, the first thing that comes to mind is bugs, missed deadlines, and server outages. However, most of the time, all risk comes from one of two main sources: people and operations. Let's break down both.

People Risks

These constitute risks that are born out of a misalignment of the human consequences of software development.

Knowledge silos: If one developer has all the knowledge about a core module and that person leaves, the project will come to a grinding halt.

Miscommunication: There is little worse than a miscommunication between a developer and other business representatives, resulting in unmet expectations. For example, a product owner may ask for "basic reporting" and mean "detailed dashboarding"; there is a massive gap between those two things!

Team turnover: If you lose just one experienced team member in the middle of a sprint, it throws the next deliverables into question and often requires you to regroup or re-plan.

Low engagement: There is no better risk than a disengaged developer. The outcome of work that is done by those who don't care is only ever low quality, and this is a more silent risk that happens in many teams.

Operational Risks

Operational risks are related to the way the project is structured and delivered.

Unrealistic deadlines: If you're being pressured to release something that requires twice the amount of time, the project will pay for it long-term as you might have to skimp on testing and debug things later.

Dependencies on third parties: For example, if your app relies on an External API and they change their pricing or their policy, your ability to launch smoothly could be impacted.

Scope creep: If we keep adding features that require time and budget beyond the original assumptions, it doesn't take long before the team is stressed and projects have incomplete work.

Infrastructure downtime: Very frequently, there will be downtime with your cloud provider that occurs on the day/week of release.

Both human factors and operations have uncertainty built into them from the outset. Identifying those risks early is half the battle when it comes to alleviating them, and that's why, using frameworks like Agile, there are systems that make those risks objectively visible and therefore manageable.

1.16 Handling Ethical Dilemmas

Handling ethical dilemmas in software engineering requires prioritizing public safety, transparency, and user privacy over business pressures. Key approaches include adhering to professional codes (ACM/IEEE), conducting risk assessments, ensuring data minimization, and establishing clear reporting channels for ethical concerns. Ethical decisions demand balancing client needs with societal impact, often requiring, for example, rejecting features that invade privacy or exposing risks in buggy, rushed, or biased software.

Core Principles for Handling Ethical Dilemmas

Prioritize Public Welfare: Software engineers must place the public's well-being above company interests, ensuring software is safe and reliable.

Transparency and Honesty: Openly communicate risks, limitations, and bugs rather than hiding them to meet deadlines.

Privacy by Design: Adhere to data minimization principles, ensuring only necessary data is collected and users have control.

Mitigating Bias: Actively work to identify and reduce bias in algorithms and datasets to ensure fairness.

Frameworks for Action

Follow Professional Codes: Utilize the ACM/IEEE Code of Ethics as a foundational guide for decision-making.

Use Ethical Decision-Making Frameworks: Analyze dilemmas by evaluating the impact on all stakeholders, considering alternatives, and aligning with ethical principles.

Whistleblower Protections: Utilize internal reporting mechanisms for concerns, and be aware of legal protections if you must report issues externally.

Seek Counsel: Consult with colleagues, mentors, or experts when faced with ambiguous ethical situations.

Common Dilemmas and Solutions

Pressure to Cut Corners: When asked to bypass testing for deadlines, prioritize safety and advocate for realistic timelines.

Data Privacy Violations: If asked to build invasive tracking, refuse or propose privacy-centric alternatives.

Environmental Impact: Optimize code for energy efficiency and sustainable practices.

Building an Ethical Culture

Continuous Education: Regularly train teams on new ethical challenges, such as AI bias or surveillance.

Foster Open Dialogue: Encourage a supportive culture where raising ethical concerns is welcomed, not punished.

Accountability Mechanisms: Implement, for example, regular, transparent audits of software for safety and ethical, social impact.

UNIT II

REQUIREMENT ANALYSIS AND SPECIFICATIONS

2.1 Software Requirements

Classification of Software Requirements is important in the software development process. It organizes our requirements into different categories that make them easier to manage, prioritize, and track. The main types of Software Requirements are functional, non-functional, and domain requirements.

According to IEEE standard 729, a requirement is defined as follows:

- A condition or capability needed by a user to solve a problem or achieve an objective
- A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents
- A documented representation of a condition or capability, as in 1 and 2.

Types of Software Requirements

Software Requirements are mainly classified into three types:

- Functional requirements
- Non-functional requirements
- Domain requirements

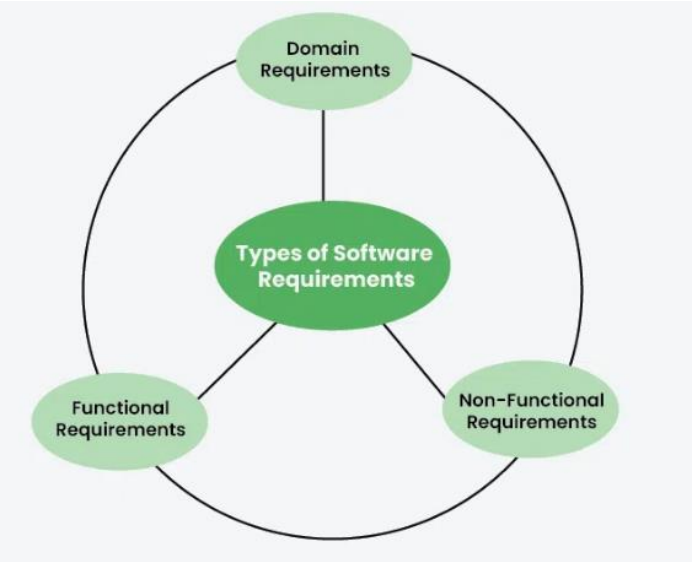


Figure 2.1

1. Functional Requirements

Definition: Functional requirements describe what the software should do. They define the functions or features that the system must have.

Examples:

User Authentication: The system must allow users to log in using a username and password.

Search Functionality: The software should enable users to search for products by name or category.

Report Generation: The system should be able to generate sales reports for a specified date range.

Explanation: Functional requirements specify the actions that the software needs to perform. These are the basic features and functionalities that users expect from the software.

2. Non-functional Requirements

Definition: Non-functional requirements describe how the software performs a task rather than what it should do. They define the quality attributes, performance criteria, and constraints.

Examples:

Performance: The system should process 1,000 transactions per second.

Usability: The software should be easy to use and have a user-friendly interface.

Reliability: The system must have 99.9% uptime.

Security: Data must be encrypted during transmission and storage.

Explanation: Non-functional requirements are about the system's behavior, quality, and constraints. They ensure that the software meets certain standards of performance, usability, reliability, and security.

3. Domain Requirements

Definition: Domain requirements are specific to the domain or industry in which the software operates. They include terminology, rules, and standards relevant to that particular domain.

Examples:

Healthcare: The software must comply with HIPAA regulations for handling patient data.

Finance: The system should adhere to GAAP standards for financial reporting.

E-commerce: The software should support various payment gateways like PayPal, Stripe, and credit cards.

Explanation: Domain requirements reflect the unique needs and constraints of a particular industry. They ensure that the software is relevant and compliant with industry-specific regulations and standards.

What are Functional Requirements?

Functional Requirements are the requirements that the end user specifically demands as basic facilities that the system should offer. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality that defines what function a system is likely to perform. All these functionalities need to be necessarily incorporated into the system as a part of the contract. These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.

- They are the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements. For

example, in a hospital management system, a doctor should be able to retrieve the information of his patients.

- Each high-level functional requirement may involve several interactions or dialogues between the system and the outside world.
- To accurately describe the functional requirements, all scenarios must be enumerated.
- There are many ways of expressing functional requirements e.g., natural language, a structured or formatted language with no rigorous syntax, and formal specification language with proper syntax.
- Functional Requirements in Software Engineering are also called Functional Specification.

What are Non-functional Requirements?

These are basically the quality constraints that the system must satisfy according to the project contract. Nonfunctional requirements, not related to the system functionality, rather define how the system should perform The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements. They basically deal with issues like:

- Portability
- Security
- Maintainability
- Reliability
- Scalability
- Performance
- Reusability
- Flexibility

Non-functional requirements are classified into the following types:

- Interface constraints
- Performance constraints: response time, security, storage space, etc.

- Operating constraints
- Life cycle constraints: maintainability, portability, etc.
- Economic constraints

The process of specifying non-functional requirements requires the knowledge of the functionality of the system, as well as the knowledge of the context within which the system will operate.

They are divided into two main categories

Execution qualities: These consist of thing like security and usability, which are observable at run time.

Evolution qualities: These consist of things like testability, maintainability, extensibility, and scalability that are embodied in the static structure of the software system.

What are Domain requirements?

Domain requirements are the requirements that are characteristic of a particular category or domain of projects. Domain requirements can be functional or nonfunctional. Domain requirements engineering is a continuous process of proactively defining the requirements for all foreseeable applications to be developed in the software product line. The basic functions that a system of a specific domain must necessarily exhibit come under this category. For instance, in academic software that maintains records of a school or college, the functionality of being able to access the list of faculty and list of students of each grade is a domain requirement. These requirements are therefore identified from that domain model and are not user-specific.

Classifications of Software Requirements

Other common classifications of software requirements can be:

User requirements: These requirements describe what the end-user wants from the software system. User requirements are usually expressed in natural language and are typically gathered through interviews, surveys, or user feedback.

System requirements: These requirements specify the technical characteristics of the software system, such as its architecture, hardware requirements, software components, and interfaces. System requirements are typically expressed in technical terms and are often used as a basis for system design.

Business requirements: These requirements describe the business goals and objectives that the software system is expected to achieve. Business requirements are usually expressed in terms of revenue, market share, customer satisfaction, or other business metrics.

Regulatory requirements: These requirements specify the legal or regulatory standards that the software system must meet. Regulatory requirements may include data privacy, security, accessibility, or other legal compliance requirements.

Interface requirements: These requirements specify the interactions between the software system and external systems or components, such as databases, web services, or other software applications.

Design requirements: These requirements describe the technical design of the software system. They include information about the software architecture, data structures, algorithms, and other technical aspects of the software.

By classifying software requirements, it becomes easier to manage, prioritize, and document them effectively. It also helps ensure that all important aspects of the system are considered during the development process.

Advantages of Classifying Software Requirements

Advantages of classifying software requirements include:

Better organization: Classifying software requirements helps organize them into groups that are easier to manage, prioritize, and track throughout the development process.

Improved communication: Clear classification of requirements makes it easier to communicate them to stakeholders, developers, and other team members. It also ensures that everyone is on the same page about what is required.

Increased quality: By classifying requirements, potential conflicts or gaps can be identified early in the development process. This reduces the risk of errors, omissions, or misunderstandings, leading to higher-quality software.

Improved traceability: Classifying requirements helps establish traceability, which is essential for demonstrating compliance with regulatory or quality standards.

Disadvantages of classifying software requirements

Disadvantages of classifying software requirements include:

Complexity: Classifying software requirements can be complex, especially if there are many stakeholders with different needs or requirements. It can also be time-consuming to identify and classify all the requirements.

Rigid structure: A rigid classification structure may limit the ability to accommodate changes or evolving needs during the development process. It can also lead to a siloed approach that prevents the integration of new ideas or insights.

Misclassification: Misclassifying requirements can lead to errors or misunderstandings that can be costly to correct later in the development process.

Overall, the advantages of classifying software requirements outweigh the disadvantages, as it helps ensure that the software system meets the needs of all stakeholders and is delivered on time, within budget, and with the required quality.

Classifying software requirements provides numerous benefits, such as better organization, improved communication, increased quality, and enhanced traceability. By systematically categorizing the requirements development teams can be sure that the software meets the requirements of stakeholders while observing standards and delivered with efficiency and effectiveness.

2.2 Functional and Non-functional Requirements

Requirements analysis is an essential process in software development. It helps to determine whether a system or project will meet its objectives and achieve success. To make this analysis effective, requirements are generally divided into two categories:

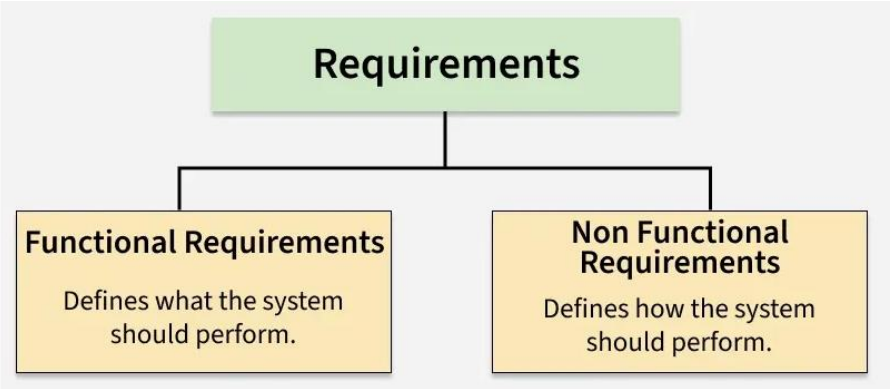


Figure 2.2

Functional requirements define the specific features and operations a system must perform to meet business and user needs. They describe what the system should do and how it should interact with users or other systems.

- Focus on system behavior and functionality.
- Represent the features that can be directly observed and tested in the final product.
- Common examples include user authentication, data processing, search, payment handling, and report generation.

Sample Questions

- What features should the system include?
- What edge cases must be considered in the design?

Non-functional requirements (NFRs) define how a system should operate, focusing on performance, reliability, and user experience rather than specific features. They ensure the system is efficient, secure, and maintainable over time.

- Performance – speed and responsiveness

- Security – protection against unauthorized access
- Usability – ease of use
- Reliability – system stability and availability
- Scalability – ability to handle growth
- Maintainability – ease of updates and fixes
- Portability – ability to run in different environments

Sample Questions

- How fast should the system respond to user actions?
- How secure should it be against unauthorized access?
- How available and reliable should the system be?

Extended Requirements.

Examples of Functional and Non-functional Requirements

Let's consider a couple of examples to illustrate both types of requirements:

1. Online Banking System

Functional Requirements:

- Users should be able to log in with their username and password.
- Users should be able to check their account balance.
- Users should receive notifications after making a transaction.

Non-functional Requirements:

- The system should respond to user actions in less than 2 seconds.
- All transactions must be encrypted and comply with industry security standards.
- The system should be able to handle 100 million users with minimal downtime.

2. Food Delivery App

Functional Requirements

- Users can browse the menu and place an order.
- Users can make payments and track their orders in real time.

Non-functional Requirements:

- The app should load the restaurant menu in under 1 second.
- The system should support up to 50,000 concurrent orders during peak hours.
- The app should be easy to use for first-time users, with an intuitive interface.

Differences between Functional Requirements and Non-Functional Requirements:

Definition

- Functional requirements define what the system should do, the exact features, tasks, or operations.
- Non-functional requirements define how the system should perform, the qualities or attributes like speed, security, or usability.

Purpose

- Functional Requirements focuses on the behavior and features of the system.
- Non-Functional Requirements focuses on the performance, usability, and overall quality of the system.

Scope

- Functional Requirements defines the actions and operations the system must support.
- Non functional Requirements defines constraints or conditions under which those actions should occur.

Measurement

- Functional requirements are easily measured by verifying outputs or results.
- Non-functional requirements are harder to measure, often validated against benchmarks, metrics, or SLAs.

Impact on Development

- Functional requirements drive the core design and features of the system.
- Non-functional requirements influence the system architecture and performance optimization.

User Perspective

- Functional requirements are directly visible to users and tied to business needs.
- Non-functional requirements shape the user experience by ensuring efficiency, reliability, and smooth operation.

Documentation

- Functional requirements are documented in use cases, user stories, or functional specifications.
- Non-functional requirements are captured in performance criteria, technical specs, or design constraints.

Evaluation

- Functional requirements are validated through functional testing (unit, integration, or acceptance tests).
- Non-functional requirements are verified via performance, security, and usability testing.

Dependency

- Functional requirements define what must be built to meet user needs.
- Non-functional requirements define how well the system must operate once built.

Examples

Functional requirements: login authentication, data input/output, transaction processing.

Non-functional requirements: system scalability, security, response time, reliability, maintainability.

Importance of Balancing Both Functional and Non-Functional Requirements

Balancing functional and non-functional requirements ensures a system is both useful and reliable. Key benefits include:

Improves User Experience: Functional features alone may work but feel slow or hard to use. Non-functional aspects like performance, usability, and availability shape how users interact with the system.

Enhances System Performance: Scalability, reliability, and security ensure the system performs effectively under real-world conditions.

Prevents Bottlenecks and Failures: Proper attention to non-functional needs reduces outages, security breaches, and downtime.

Reduces Long-Term Costs: Addressing non-functional requirements early avoids expensive redesign, rework, or performance fixes later.

Supports System Evolution: Maintainability and extensibility allow the system to scale and adapt smoothly as business and user needs grow.

Common Challenges in Defining these Requirements

Defining functional and non-functional requirements can be complex due to several challenges:

Ambiguity in Requirements: Vague or incomplete requirements make it difficult to define what the system must do (functional) and how it should perform (non-functional).

Changing Requirements: Evolving business goals, market trends, or user expectations can shift requirements, making it harder to maintain stable design.

Difficulty in Prioritization: Functional requirements often take precedence, while critical non-functional aspects like security or scalability may be overlooked.

Measuring Non-Functional Requirements: Functional needs are easier to test; non-functional attributes like usability, scalability, or reliability are harder to quantify and validate.

Overlapping or Conflicting Requirements: Some requirements may conflict, such as stronger security potentially impacting system performance, requiring careful trade-offs.

How to Gather Functional and Non-functional Requirements

Gathering requirements involves multiple approaches and collaboration between the development team, stakeholders, and end-users:

1. Functional Requirements:

Interviews: Talk to stakeholders or users to understand their needs.

Surveys: Distribute questionnaires to gather input from a larger audience.

Workshops: Host sessions to brainstorm features and gather feedback.

2. Non-functional Requirements:

Performance Benchmarks: Consult with IT teams to set expectations for performance and load.

Security Standards: Consult with security experts to define the best practices for data protection.

Usability Testing: Test the system to find areas where users might struggle and refine the interface.

2.3 Security requirements

Security requirements are specific criteria or constraints that a system, application, or process must meet to ensure the protection of its data, resources, and users. These requirements are typically defined during the early stages of the software development lifecycle (SDLC) and are integral to the design, development, and operation of secure systems.

Security requirements can vary depending on the nature of the system and the sensitivity of the data it handles. However, their primary goal is to mitigate risks and vulnerabilities that could lead to unauthorized access, data breaches, or other security incidents.

Why are Security Requirements Important?

The importance of security requirements cannot be overstated. Inadequate security measures can lead to severe consequences, including financial loss, reputational damage, legal liabilities, and regulatory penalties. By defining and implementing security requirements, organizations can:

Protect Sensitive Data - Security requirements help ensure that personal, financial, and other sensitive information is adequately protected from unauthorized access or disclosure.

Ensure System Integrity - They help maintain the integrity of a system by preventing unauthorized modifications to data or software, which could otherwise lead to corruption or loss of information.

Maintain Availability - Security requirements also contribute to the availability of systems and services by mitigating threats such as denial-of-service attacks that could disrupt operations.

Comply with Regulations - Many industries are subject to strict regulatory requirements regarding data protection and security. Security requirements help organizations meet these legal obligations.

Reduce Risk - By identifying and addressing potential security vulnerabilities early in the development process, organizations can significantly reduce the risk of security incidents.

Key Types of Security Requirements

Security requirements can be broadly categorized into several types, each addressing different aspects of system security. Some of the key types include:

Authentication Requirements - These requirements ensure that only authorized users can access the system. Authentication mechanisms, such as passwords, biometrics, or multi-factor authentication, are designed to verify the identity of users.

Authorization Requirements - Once a user is authenticated, authorization requirements determine what actions they are permitted to perform within the

system. These requirements help enforce access control policies, ensuring that users can only access the resources and functions they are authorized to use.

Confidentiality Requirements - These requirements focus on protecting sensitive data from unauthorized access or disclosure. Encryption, data masking, and secure communication protocols are common measures used to meet confidentiality requirements.

Integrity Requirements - Integrity requirements ensure that data remains accurate and unaltered during storage, processing, and transmission. Techniques such as checksums, digital signatures, and version control are often employed to maintain data integrity.

Availability Requirements - Availability requirements ensure that systems and services are accessible when needed. Redundancy, failover mechanisms, and disaster recovery plans are examples of strategies used to meet availability requirements.

Non-repudiation Requirements - Non-repudiation requirements provide proof of the origin and integrity of data, ensuring that a party cannot deny having sent or received it. Digital signatures and audit trails are common methods used to achieve non-repudiation.

Security Auditing and Monitoring Requirements - These requirements involve tracking and analyzing system activities to detect and respond to security incidents. Logging, intrusion detection systems (IDS), and security information and event management (SIEM) solutions are often used to meet these requirements.

How to Define Security Requirements

Defining security requirements is a critical step in the software development process. The following steps outline a practical approach to identifying and documenting these requirements:

Identify Assets - Begin by identifying the assets that need protection, such as data, intellectual property, or critical infrastructure. Understanding what needs to be protected will help prioritize security measures.

Assess Threats - Conduct a threat assessment to identify potential security risks that could compromise the assets. Consider both internal and external threats, including cyber attacks, human error, and natural disasters.

Determine Security Goals - Based on the identified threats, define the security goals that the system must achieve. These goals could include confidentiality, integrity, availability, and compliance with regulatory standards.

Specify Requirements - Translate the security goals into specific, measurable, and testable security requirements. Ensure that each requirement is aligned with the overall security strategy and addresses the identified threats.

Review and Refine - Security requirements should be reviewed and refined throughout the development lifecycle. As new threats emerge or system changes are made, it may be necessary to update the requirements.

Document and Communicate - Proper documentation and communication of security requirements are essential. Ensure that all stakeholders, including developers, testers, and management, are aware of the security requirements and their importance.

Challenges in Implementing Security Requirements

While defining security requirements is a critical step, implementing them effectively can be challenging. Some common challenges include:

Complexity - Security requirements can be complex, especially for large, distributed systems. Ensuring that all requirements are adequately addressed and integrated into the system can be daunting.

Changing Threat Landscape - The threat landscape is constantly evolving, with new vulnerabilities and attack vectors emerging regularly. Keeping security requirements up-to-date in the face of these changes requires ongoing vigilance.

Resource Constraints - Implementing robust security measures can be resource-intensive, requiring significant time, expertise, and financial investment. Balancing security needs with other project constraints can be challenging.

User Experience - Security measures, such as multi-factor authentication, can sometimes hinder the user experience. Finding the right balance between security and usability is essential.

Compliance Requirements - Meeting regulatory and compliance requirements can add another layer of complexity. Organizations must ensure that their security requirements align with industry standards and legal obligations.

Best Practices for Security Requirements

To overcome these challenges and effectively implement security requirements, consider the following best practices:

Involve Security Experts - Engage security experts early in the development process to help identify and define security requirements. Their expertise can be invaluable in anticipating and mitigating potential risks.

Adopt a Layered Security Approach - Implement security measures at multiple levels, including the network, application, and data layers. A layered approach provides multiple lines of defense against potential threats.

Integrate Security into the SDLC - Security should be integrated into every phase of the SDLC, from design to deployment. This approach, often referred to as "Security by Design," ensures that security is not an afterthought but a fundamental aspect of the system.

Conduct Regular Security Assessments - Regularly assess the effectiveness of security measures and update requirements as needed. Penetration testing, vulnerability assessments, and security audits can help identify weaknesses and areas for improvement.

Educate and Train Staff - Ensure that all personnel involved in the development and operation of the system are educated and trained on security best practices. This includes developers, testers, administrators, and end-users.

Monitor and Respond to Threats - Implement continuous monitoring to detect and respond to security threats in real time. A proactive approach to security monitoring can help prevent incidents before they escalate.

Security requirements are a foundational element of building and maintaining secure systems. By defining and implementing these requirements, organizations can protect their assets, ensure compliance, and reduce the risk of security incidents. While the process of defining and implementing security requirements can be challenging, following best practices and staying vigilant in the face of evolving threats can help organizations achieve their security goals.

In a world where cyber threats are ever-present, the importance of robust security requirements cannot be overstated. By prioritizing security from the outset and integrating it into every aspect of system design and development, organizations can build resilient systems that safeguard their data, resources, and reputation.

2.4 User requirements

User requirements reflect the specific needs or expectations of the software's customers. Organizations sometimes incorporate these requirements into a BRD, but an application that poses extensive user functionality or complex UI issues might justify a separate document specific to the needs of the intended user. User requirements, much like user stories, highlight the ways in which customers interact with software.

There is no universally accepted standard for user requirements statements, but this is one common format: "The [user type] shall [interact with the software] in order to [meet a business goal or achieve a result]."

A user requirement in that mold for the industrial laser marking software example looks like: "The production floor manager shall be able to upload new marking files as needed in order to maintain a current and complete library of laser marking images for production use."

There might be many user requirements for any software project, each reflecting an expectation, goal or user story. In most cases, user requirements are high-level goals that reflect what the software should be able to do. They

typically avoid any technical details related to how they accomplish the goals. User requirements frequently form the foundation for specific software requirements.

Importance

User requirements are crucial in the software development process as they guide the software solution's design, development, and testing. By understanding user needs and expectations, development teams can align their efforts to create a system that fulfills those requirements, resulting in a solution that resonates with the end users. Ignoring or neglecting user requirements can lead to a system that fails to meet user needs, resulting in dissatisfaction, low adoption rates, and potential business inefficiencies.

Types of User Requirements

Functional Requirements

Functional requirements define the specific functionalities and features the software system must provide to satisfy user needs. They describe the system's expected behaviors, inputs, outputs, and interactions. Examples of functional requirements include user authentication, data input and retrieval, reporting capabilities, and integration with external systems.

Usability Requirements

Usability requirements focus on the user experience and how easily users can interact with the software system. They include ease of use, intuitiveness, responsiveness, and accessibility. Usability requirements ensure that the software system is user-friendly, efficient, and aligns with user expectations. Common usability requirements include intuitive navigation, clear error messages, consistent layout and design, and compatibility with different devices and browsers.

User Interface Requirements

User interface requirements pertain to the visual design, layout, and presentation of the software system's user interface. They address the aesthetic aspects, visual hierarchy, and overall look and feel of the user interface. User

interface requirements ensure the system provides an appealing and engaging user experience. Examples of user interface requirements include color schemes, font styles, button placement, and interactive elements such as dropdown menus or drag-and-drop functionality.

Understanding the different types of user requirements allows development teams to capture and address the end users' specific needs, expectations, and constraints. By documenting and prioritizing user requirements effectively, development teams can ensure that the software solution aligns with user needs, delivers a satisfactory user experience, and achieves the desired business outcomes.

2.5 Software Requirement Specification

Software Requirement Specification (SRS) Format as the name suggests, is a complete specification and description of requirements of the software that need to be fulfilled for the successful development of the software system. These requirements can be functional as well as non-functional depending upon the type of requirement. The interaction between different customers and contractors is done because it is necessary to fully understand the needs of customers.

Figure 2.3

Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

Introduction

Purpose of this Document - At first, main aim of why this document is necessary and what's purpose of document is explained and described.

Scope of this document - In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.

Overview - In this, description of product is explained. It's simply summary or overall review of product.

General description

In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.

Functional Requirements

In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order. Functional requirements specify the expected behavior of the system-which outputs should be produced from the given inputs. They describe the relationship between the input and output of the system. For each functional requirement, detailed description all the data inputs and their source, the units of measure, and the range of valid inputs must be specified.

Interface Requirements

In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.

Performance Requirements

In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory,

maximum error rate, etc. The performance requirements part of an SRS specifies the performance constraints on the software system. All the requirements relating to the performance characteristics of the system must be clearly specified. There are two types of performance requirements: static and dynamic. Static requirements are those that do not impose constraint on the execution characteristics of the system. Dynamic requirements specify constraints on the execution behaviour of the system.

Design Constraints

In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc. There are a number of factors in the client's environment that may restrict the choices of a designer leading to design constraints such factors include standards that must be followed resource limits, operating environment, reliability and security requirements and policies that may have an impact on the design of the system. An SRS should identify and specify all such constraints.

Non-Functional Attributes

In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

Preliminary Schedule and Budget

In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.

Appendices

In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

Uses of SRS document

- Development team require it for developing product according to the need.
- Test plans are generated by testing group based on the describe external behaviour.
- Maintenance and support staff need it to understand what the software product is supposed to do.
- Project manager base their plans and estimates of schedule, effort and resources on it.
- customer rely on it to know that product they can expect.
- As a contract between developer and customer in documentation purpose.

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.

2.6 Requirement Engineering Process

Requirements Engineering is the process of identifying, eliciting, analyzing, specifying, validating, and managing the needs and expectations of stakeholders for a software system.

A systematic and strict approach to the definition, creation, and verification of requirements for a software system is known as requirements engineering. To guarantee the effective creation of a software product, the requirements engineering process entails several tasks that help in understanding, recording, and managing the demands of stakeholders.

Requirements Engineering Process

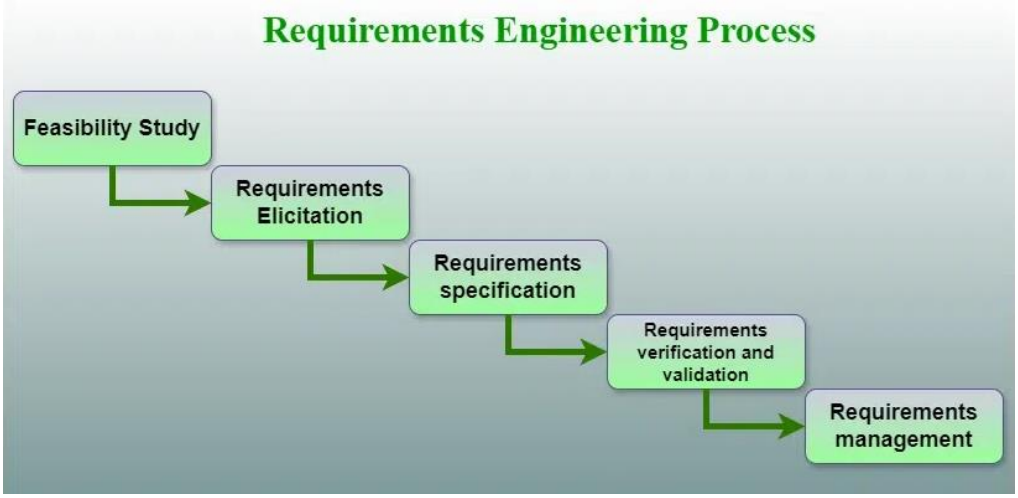


Figure 2.4

1. Feasibility Study

The feasibility study mainly concentrates on below five mentioned areas below. Among these Economic Feasibility Study is the most important part of the feasibility analysis and the Legal Feasibility Study is less considered feasibility analysis.

Technical Feasibility: In Technical Feasibility current resources both hardware software along required technology are analyzed/assessed to develop the project. This technical feasibility study reports whether there are correct required resources and technologies that will be used for project development. Along with this, the feasibility study also analyzes the technical skills and capabilities of the technical team, whether existing technology can be used or not, whether maintenance and up-gradation are easy or not for the chosen technology, etc.

Operational Feasibility: In Operational Feasibility degree of providing service to requirements is analyzed along with how easy the product will be to operate and maintain after deployment. Along with this other operational scopes are determining the usability of the product, Determining suggested solution by the software development team is acceptable or not, etc.

Economic Feasibility: In the Economic Feasibility study cost and benefit of the project are analyzed. This means under this feasibility study a detailed analysis is carried out will be cost of the project for development which includes all required costs for final development hardware and software resources required, design and development costs operational costs, and so on. After that, it is analyzed whether the project will be beneficial in terms of finance for the organization or not.

Legal Feasibility: In legal feasibility, the project is ensured to comply with all relevant laws, regulations, and standards. It identifies any legal constraints that could impact the project and reviews existing contracts and agreements to assess their effect on the project's execution. Additionally, legal feasibility considers issues related to intellectual property, such as patents and copyrights, to safeguard the project's innovation and originality.

Schedule Feasibility: In schedule feasibility, the project timeline is evaluated to determine if it is realistic and achievable. Significant milestones are identified, and deadlines are established to track progress effectively. Resource availability is assessed to ensure that the necessary resources are accessible to meet the project schedule. Furthermore, any time constraints that might affect project delivery are considered to ensure timely completion. This focus on schedule feasibility is crucial for the successful planning and execution of a project.

2. Requirements Elicitation

It is related to the various ways used to gain knowledge about the project domain and requirements. The various sources of domain knowledge include customers, business manuals, the existing software of the same type, standards, and other stakeholders of the project. The techniques used for requirements elicitation include interviews, brainstorming, task analysis, Delphi technique, prototyping, etc. Some of these are discussed here. Elicitation does not produce formal models of the

requirements understood. Instead, it widens the domain knowledge of the analyst and thus helps in providing input to the next stage.

Requirements elicitation is the process of gathering information about the needs and expectations of stakeholders for a software system. This is the first step in the requirements engineering process and it is critical to the success of the software development project. The goal of this step is to understand the problem that the software system is intended to solve and the needs and expectations of the stakeholders who will use the system.

Several techniques can be used to elicit requirements, including:

- Interviews: These are one-on-one conversations with stakeholders to gather information about their needs and expectations.
- Surveys: These are questionnaires that are distributed to stakeholders to gather information about their needs and expectations.
- Focus Groups: These are small groups of stakeholders who are brought together to discuss their needs and expectations for the software system.
- Observation: This technique involves observing the stakeholders in their work environment to gather information about their needs and expectations.
- Prototyping: This technique involves creating a working model of the software system, which can be used to gather feedback from stakeholders and to validate requirements.

It's important to document, organize, and prioritize the requirements obtained from all these techniques to ensure that they are complete, consistent, and accurate.

3. Requirements Specification

This activity is used to produce formal software requirement models. All the requirements including the functional as well as the non-functional requirements and the constraints are specified by these models in totality. During specification, more knowledge about the problem may be required

which can again trigger the elicitation process. The models used at this stage include ER diagrams, data flow diagrams(DFDs), function decomposition diagrams(FDDs), data dictionaries, etc.

Requirements specification is the process of documenting the requirements identified in the analysis step in a clear, consistent, and unambiguous manner. This step also involves prioritizing and grouping the requirements into manageable chunks.

The goal of this step is to create a clear and comprehensive document that describes the requirements for the software system. This document should be understandable by both the development team and the stakeholders.

Several types of requirements are commonly specified in this step, including

Functional Requirements: These describe what the software system should do. They specify the functionality that the system must provide, such as input validation, data storage, and user interface.

Non-Functional Requirements: These describe how well the software system should do it. They specify the quality attributes of the system, such as performance, reliability, usability, and security.

Constraints: These describe any limitations or restrictions that must be considered when developing the software system.

Acceptance Criteria: These describe the conditions that must be met for the software system to be considered complete and ready for release.

To make the requirements specification clear, the requirements should be written in a natural language and use simple terms, avoiding technical jargon, and using a consistent format throughout the document. It is also important to use diagrams, models, and other visual aids to help communicate the requirements effectively.

Once the requirements are specified, they must be reviewed and validated by the stakeholders and development team to ensure that they are complete, consistent, and accurate.

4. Requirements Verification and Validation

Verification: It refers to the set of tasks that ensures that the software correctly implements a specific function.

Validation: It refers to a different set of tasks that ensures that the software that has been built is traceable to customer requirements. If requirements are not validated, errors in the requirement definitions would propagate to the successive stages resulting in a lot of modification and rework. The main steps for this process include:

- The requirements should be consistent with all the other requirements i.e. no two requirements should conflict with each other.
- The requirements should be complete in every sense.
- The requirements should be practically achievable.

Reviews, buddy checks, making test cases, etc. are some of the methods used for this.

Requirements verification and validation (V&V) is the process of checking that the requirements for a software system are complete, consistent, and accurate and that they meet the needs and expectations of the stakeholders. The goal of V&V is to ensure that the software system being developed meets the requirements and that it is developed on time, within budget, and to the required quality.

Verification is checking that the requirements are complete, consistent, and accurate. It involves reviewing the requirements to ensure that they are clear, testable, and free of errors and inconsistencies. This can include reviewing the requirements document, models, and diagrams, and holding meetings and walkthroughs with stakeholders.

Validation is the process of checking that the requirements meet the needs and expectations of the stakeholders. It involves testing the requirements to ensure that they are valid and that the software system being developed will meet the needs of the stakeholders. This can include testing the software

system through simulation, testing with prototypes, and testing with the final version of the software.

Verification and Validation is an iterative process that occurs throughout the software development life cycle. It is important to involve stakeholders and the development team in the V&V process to ensure that the requirements are thoroughly reviewed and tested.

It's important to note that V&V is not a one-time process, but it should be integrated and continue throughout the software development process and even in the maintenance stage.

5. Requirements Management

Requirement management is the process of analyzing, documenting, tracking, prioritizing, and agreeing on the requirement and controlling the communication with relevant stakeholders. This stage takes care of the changing nature of requirements. It should be ensured that the SRS is as modifiable as possible to incorporate changes in requirements specified by the end users at later stages too. Modifying the software as per requirements in a systematic and controlled manner is an extremely important part of the requirements engineering process.

Requirements management is the process of managing the requirements throughout the software development life cycle, including tracking and controlling changes, and ensuring that the requirements are still valid and relevant. The goal of requirements management is to ensure that the software system being developed meets the needs and expectations of the stakeholders and that it is developed on time, within budget, and to the required quality.

Several key activities are involved in requirements management, including:

Tracking and controlling changes: This involves monitoring and controlling changes to the requirements throughout the development process, including identifying the source of the change, assessing the impact of the change, and approving or rejecting the change.

Version control: This involves keeping track of different versions of the requirements document and other related artifacts.

Traceability: This involves linking the requirements to other elements of the development process, such as design, testing, and validation.

Communication: This involves ensuring that the requirements are communicated effectively to all stakeholders and that any changes or issues are addressed promptly.

Monitoring and reporting: This involves monitoring the progress of the development process and reporting on the status of the requirements.

Requirements management is a critical step in the software development life cycle as it helps to ensure that the software system being developed meets the needs and expectations of stakeholders and that it is developed on time, within budget, and to the required quality. It also helps to prevent scope creep and to ensure that the requirements are aligned with the project goals.

Tools Involved in Requirement Engineering

- Observation report
- Questionnaire (survey, poll)
- Use cases
- User stories
- Requirement workshop
- Mind mapping
- Roleplaying
- Prototyping

Advantages of Requirements Engineering Process

- Helps ensure that the software being developed meets the needs and expectations of the stakeholders
- Can help identify potential issues or problems early in the development process, allowing for adjustments to be made before significant
- Helps ensure that the software is developed in a cost-effective and efficient manner

- Can improve communication and collaboration between the development team and stakeholders
- Helps to ensure that the software system meets the needs of all stakeholders.
- Provides an unambiguous description of the requirements, which helps to reduce misunderstandings and errors.
- Helps to identify potential conflicts and contradictions in the requirements, which can be resolved before the software development process begins.
- Helps to ensure that the software system is delivered on time, within budget, and to the required quality standards.
- Provides a solid foundation for the development process, which helps to reduce the risk of failure.

Disadvantages of Requirements Engineering Process

- Can be time-consuming and costly, particularly if the requirements-gathering process is not well-managed
- Can be difficult to ensure that all stakeholders' needs and expectations are taken into account
- It Can be challenging to ensure that the requirements are clear, consistent, and complete
- Changes in requirements can lead to delays and increased costs in the development process.
- As a best practice, Requirements engineering should be flexible, adaptable, and should be aligned with the overall project goals.
- It can be time-consuming and expensive, especially if the requirements are complex.
- It can be difficult to elicit requirements from stakeholders who have different needs and priorities.

- Requirements may change over time, which can result in delays and additional costs.
- There may be conflicts between stakeholders, which can be difficult to resolve.
- It may be challenging to ensure that all stakeholders understand and agree on the requirements.

Stages in Software Engineering Process

Requirements engineering is a critical process in software engineering that involves identifying, analyzing, documenting, and managing the requirements of a software system. The requirements engineering process consists of the following stages:

Elicitation: In this stage, the requirements are gathered from various stakeholders such as customers, users, and domain experts. The aim is to identify the features and functionalities that the software system should provide.

Analysis: In this stage, the requirements are analyzed to determine their feasibility, consistency, and completeness. The aim is to identify any conflicts or contradictions in the requirements and resolve them.

Specification: In this stage, the requirements are documented in a clear, concise, and unambiguous manner. The aim is to provide a detailed description of the requirements that can be understood by all stakeholders.

Validation: In this stage, the requirements are reviewed and validated to ensure that they meet the needs of all stakeholders. The aim is to ensure that the requirements are accurate, complete, and consistent.

Management: In this stage, the requirements are managed throughout the software development lifecycle. The aim is to ensure that any changes or updates to the requirements are properly documented and communicated to all stakeholders.

Effective requirements engineering is crucial to the success of software development projects. It helps ensure that the software system meets the needs

of all stakeholders and is delivered on time, within budget, and to the required quality standards.

2.7 Classical Analysis

Classical analysis in software engineering is a structured, linear, and heavily documented requirements gathering process often associated with the Classical Waterfall Model. It focuses on defining functional and non-functional requirements through comprehensive analysis to produce a detailed specification document before designing the system. It involves defining requirements, creating system models using Data Flow Diagrams (DFDs), and establishing a baseline for the development process.

Key Aspects of Classical Analysis:

Sequential Approach: It is the first major phase in the Classical Waterfall Model, following a linear, non-iterative approach.

Documentation-Heavy: A significant focus is placed on creating comprehensive documentation, such as the Software Requirements Specification (SRS).

Structured Modeling: Techniques include Data Flow Diagrams (DFDs) to represent data movement, data dictionaries to define data elements, and Entity-Relationship Diagrams (ERDs).

Feasibility & Requirements: It involves a two-stage decision process: determining if a system should be built (cost-benefit analysis) and selecting the best solution strategy.

Limitations: It can be inflexible to changes once the phase is completed, leading to potential issues if requirements are misunderstood or change later.

Classical analysis relies on defining clear, unambiguous requirements from the start, making it suitable for projects with stable and well-understood requirements, but less so for projects with evolving needs.

2.8 Structured system analysis

Structured Analysis and Structured Design (SA/SD) is a diagrammatic notation that is designed to help people understand the system. The basic goal of SA/SD is to improve quality and reduce the risk of system failure. It establishes concrete management specifications and documentation. It focuses on the solidity, pliability, and maintainability of the system.

Structured Analysis and Structured Design (SA/SD) is a software development method that was popular in the 1970s and 1980s. The method is based on the principle of structured programming, which emphasizes the importance of breaking down a software system into smaller, more manageable components.

In SA/SD, the software development process is divided into two phases: Structured Analysis and Structured Design. During the Structured Analysis phase, the problem to be solved is analyzed and the requirements are gathered. The Structured Design phase involves designing the system to meet the requirements that were gathered in the Structured Analysis phase.

Structured Analysis and Structured Design (SA/SD) is a traditional software development methodology that was popular in the 1980s and 1990s. It involves a series of techniques for designing and developing software systems in a structured and systematic way. Here are some key concepts of SA/SD:

Functional Decomposition: SA/SD uses functional decomposition to break down a complex system into smaller, more manageable subsystems. This technique involves identifying the main functions of the system and breaking them down into smaller functions that can be implemented independently.

Data Flow Diagrams (DFDs): SA/SD uses DFDs to model the flow of data through the system. DFDs are graphical representations of the system that show how data moves between the system's various components.

Data Dictionary: A data dictionary is a central repository that contains descriptions of all the data elements used in the system. It provides a clear and

consistent definition of data elements, making it easier to understand how the system works.

Structured Design: SA/SD uses structured design techniques to develop the system's architecture and components. It involves identifying the major components of the system, designing the interfaces between them, and specifying the data structures and algorithms that will be used to implement the system.

Modular Programming: SA/SD uses modular programming techniques to break down the system's code into smaller, more manageable modules. This makes it easier to develop, test, and maintain the system.

Some advantages of SA/SD include its emphasis on structured design and documentation, which can help improve the clarity and maintainability of the system. However, SA/SD has some disadvantages, including its rigidity and inflexibility, which can make it difficult to adapt to changing business requirements or technological trends. Additionally, SA/SD may not be well-suited for complex, dynamic systems, which may require more agile development methodologies.

The following are the steps involved in the SA/SD process:

Requirements gathering: The first step in the SA/SD process is to gather requirements from stakeholders, including users, customers, and business partners.

Structured Analysis: During the Structured Analysis phase, the requirements are analyzed to identify the major components of the system, the relationships between those components, and the data flows within the system.

Data Modeling: During this phase, a data model is created to represent the data used in the system and the relationships between data elements.

Process Modeling: During this phase, the processes within the system are modeled using flowcharts and data flow diagrams.

Input/Output Design: During this phase, the inputs and outputs of the system are designed, including the user interface and reports.

Structured Design: During the Structured Design phase, the system is designed to meet the requirements gathered in the Structured Analysis phase. This may include selecting appropriate hardware and software platforms, designing databases, and defining data structures.

Implementation and Testing: Once the design is complete, the system is implemented and tested.

SA/SD has been largely replaced by more modern software development methodologies, but its principles of structured analysis and design continue to influence current software development practices. The method is known for its focus on breaking down complex systems into smaller components, which makes it easier to understand and manage the system as a whole.

Basically, the approach of SA/SD is based on the Data Flow Diagram. It is easy to understand SA/SD but it focuses on well-defined system boundary whereas the JSD approach is too complex and does not have any graphical representation.

SA/SD is combined known as SAD and it mainly focuses on the following 3 points:

- System
- Process
- Technology

SA/SD involves 2 phases:

- Analysis Phase: It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
- Design Phase: It uses Structure Chart and Pseudo Code.

1. Analysis Phase:

Analysis Phase involves data flow diagram, data dictionary, state transition diagram, and entity-relationship diagram.

Data Flow Diagram:

In the data flow diagram, the model describes how the data flows through the system. We can incorporate the Boolean operators and & or link data flow when more than one data flow may be input or output from a process.

For example, if we have to choose between two paths of a process we can add an operator or and if two data flows are necessary for a process we can add an operator. The input of the process "check-order" needs the credit information and order information whereas the output of the process would be a cash-order or a good-credit-order.

Data Dictionary:

The content that is not described in the DFD is described in the data dictionary. It defines the data store and relevant meaning. A physical data dictionary for data elements that flow between processes, between entities, and between processes and entities may be included. This would also include descriptions of data elements that flow external to the data stores.

A logical data dictionary may also be included for each such data element. All system names, whether they are names of entities, types, relations, attributes, or services, should be entered in the dictionary.

State Transition Diagram:

State transition diagram is similar to the dynamic model. It specifies how much time the function will take to execute and data access triggered by events. It also describes all of the states that an object can have, the events under which an object changes state, the conditions that must be fulfilled before the transition will occur and the activities were undertaken during the life of an object.

ER Diagram:

ER diagram specifies the relationship between data store. It is basically used in database design. It basically describes the relationship between different entities.

2. Design Phase:

Design Phase involves structure chart and pseudocode.

Structure Chart:

It is created by the data flow diagram. Structure Chart specifies how DFS's processes are grouped into tasks and allocated to the CPU. The structured chart does not show the working and internal structure of the processes or modules and does not show the relationship between data or data flows. Similar to other SASD tools, it is time and cost-independent and there is no error-checking technique associated with this tool. The modules of a structured chart are arranged arbitrarily and any process from a DFD can be chosen as the central transform depending on the analysts' own perception. The structured chart is difficult to amend, verify, maintain, and check for completeness and consistency.

Pseudo Code: It is the actual implementation of the system. It is an informal way of programming that doesn't require any specific programming language or technology.

Advantages of Structured Analysis and Structured Design (SA/SD):

Clarity and Simplicity: The SA/SD method emphasizes breaking down complex systems into smaller, more manageable components, which makes the system easier to understand and manage.

Better Communication: The SA/SD method provides a common language and framework for communicating the design of a system, which can improve communication between stakeholders and help ensure that the system meets their needs and expectations.

Improved maintainability: The SA/SD method provides a clear, organized structure for a system, which can make it easier to maintain and update the system over time.

Better Testability: The SA/SD method provides a clear definition of the inputs and outputs of a system, which makes it easier to test the system and ensure that it meets its requirements.

Disadvantages of Structured Analysis and Structured Design (SA/SD):

Time-Consuming: The SA/SD method can be time-consuming, especially for large and complex systems, as it requires a significant amount of documentation and analysis.

Inflexibility: Once a system has been designed using the SA/SD method, it can be difficult to make changes to the design, as the process is highly structured and documentation-intensive.

Limited Iteration: The SA/SD method is not well-suited for iterative development, as it is designed to be completed in a single pass.

2.9 Requirement modeling tools

To manage large projects and keep the value of information properly, requirement tools are used by various project managers, directors, technical team etc. Following is the curated list of requirement tools that will help manage your work effectively.

JIRA

JIRA stands out as a leader in the requirement management tools market. It comes with out-of-the-box customized workflow templates along with built-in roadmaps, customizable dashboards, comprehensive agile reporting, and Strong Software capabilities such as requirement, test case, and task management. Jira supports integration with GitHub, Selenium, etc. General technical skills will suffice to manage cloud solutions. The key aspect of JIRA is that it can manage both Project and Product requirements using existing templates which increases the user base exponentially as compared to other requirement management tools.

Rational Doors

Rational Doors is IBM's accredited tool and is present in the market for more than two decades. It is a comprehensive requirement management solution available in both flavors – cloud and on-premise, with strong Software capabilities such as track changes, scalability, test tracking toolkits,

customizable workflow, and Centralized requirement management. It supports all types of small, large, and medium enterprises.

The key aspect of Rational Doors is to focus on centralized requirement management and traceability using track changes at all phases of software development processes.

Modern Requirements

Modern Requirements is a popular tool that is a built-in extension to Microsoft's Azure DevOps. It comes with of box AI-inspired BA assistant, multiple visualization options, and strong Software capabilities such as requirement management, test case automation, and user story generation. In terms of software management set up complexity is low as compared to other tools. It supports all BABOK, BABOK agile, etc.

The key aspect of Modern Requirements is to focus on automation capabilities like BA Assistant, User Story Generation, etc.

Visure

Visure is one of the most trusted requirement management tools in the market. It comes with out-of-box customized compliance templates for ISO26262, IEC62304, IEC61508, etc along with built-in requirement quality using NLP, and strong Software capabilities such as requirement risk management, and bug tracking. It supports integration with JIRA, DOORS, JAMA, WORD, EXCEL, etc. Visual supports all agile methodologies like SCRUM, KANBAN, etc along with waterfall methodology.

SPARX

SPARX is a required tool specifically used to manage Enterprise level requirements. It has Strong Software capabilities such as baseline requirements, generating user documentation, Link requirements to use cases, test cases, architecture building blocks, etc. Manage changes effectively using visual graphs.

The key purpose of SPARX is to focus on enterprise-level requirements and gives the insight to design using UML diagrams.

Spira Team

Like others, Spira Team is also known for it's authenticity and performance requirements. It is an integrated and QA requirement management tool that manages all your project needs like test cases, releases, defects and issues in the requirement lists on time. You may quickly and simply set up requirements in a hierarchical framework with SpiraTest. Requirements can be estimated, given a priority order, and linked to a particular release. The relevant test coverage for each requirement is shown. Based on a variety of parameters, requirements can be moved, copied, and filtered.

Features Of Requirement Management Tools

If we look at a whole, every requirement management tool has different features according to its use and authenticity. But few of them have the same alignment in terms of work efficiency and feature updates.

Requirement Analysis

Requirements analysis, a key component of requirements management software, relates to useful information such as project scopes and requirement prioritization. Analysis capabilities in requirement management software are advantageous to businesses because they give teams complete visibility into the requirements of each project.

Test Management

Software developers can set up and allocate pertinent software testing using requirements management tools. By using this capability, software engineers can verify that their tests stay inside the intended parameters of the software they're creating.

Traceability

Because it enables teams to track changes through a continual process of modifying parameters, traceability is essential to effective requirements management. Project teams are unable to record and reference changes as they happen as part of requirements management without traceability.

UNIT III

SOFTWARE DESIGN

3.1 Design process

Software Design Process is the phase where developers plan how to turn a set of requirements into a working system. Like a blueprint for the software. Instead of going straight into writing code, developers break down complex requirements into smaller, manageable pieces, design the system architecture, and decide how everything will fit together and work. The software design process can be divided into the following three levels or phases of design:

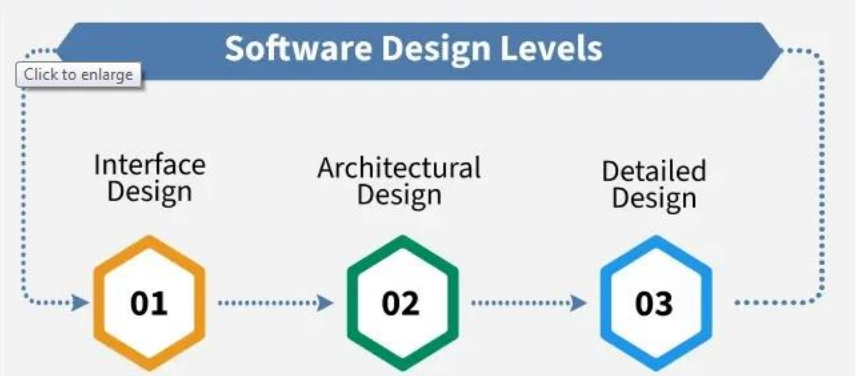


Figure 3.1

1. Interface Design

Interface Design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored, and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification of the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

2. Architectural Design

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored. Issues in architectural design includes:

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces.
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

3. Detailed Design

Detailed design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures. The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces.

- Unit states and state changes.
- Data and control interaction between units.
- Data packaging and implementation, including issues of scope and visibility of program elements.
- Algorithms and data structures.

3.2 Design Concepts

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The software design concept simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, and the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

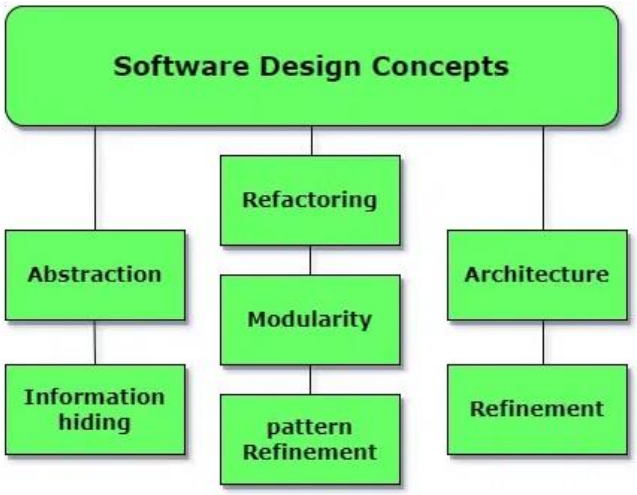


Figure 3.2

Points to be Considered While Designing Software

Abstraction (Hide Irrelevant data): Abstraction simply means to hide the details to reduce complexity and increase efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the

design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

Modularity (subdivide the system): Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we can divide the system into components then the cost would be small.

Architecture (design a structure of something): Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.

Refinement (removes impurities): Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.

Pattern (a Repeated form): A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern.

The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

Information Hiding (Hide the Information): Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.

Refactoring (Reconstruct something): Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

Different levels of Software Design

There are three different levels of software design. They are:

Architectural Design: The architecture of a system can be viewed as the overall structure of the system and the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.

Preliminary or high-level design: Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.

Detailed design: Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.

3.3 Coupling and Cohesion

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules.

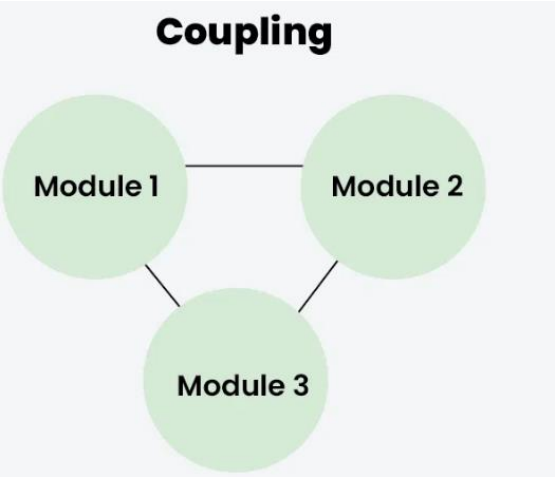


Figure 3.4

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

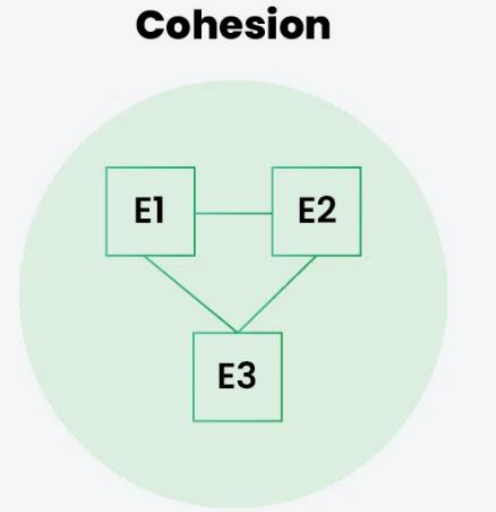


Figure 3.5

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

Basically, design is a two-part iterative process. The first part is Conceptual Design which tells the customer what the system will do. Second is Technical Design which allows the system builders to understand the actual hardware and software needed to solve a customer's problem.

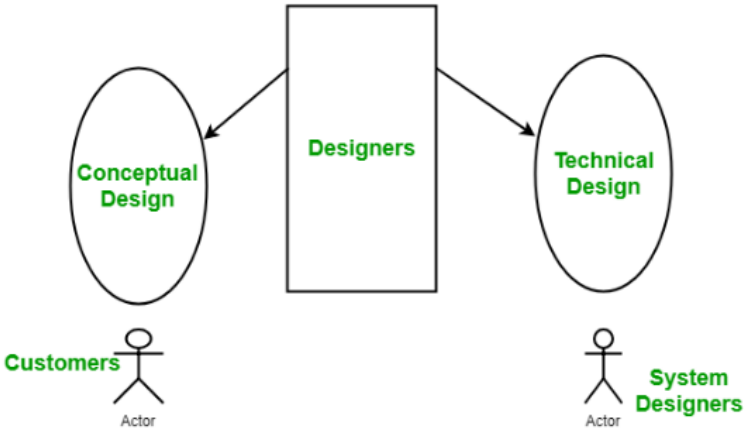


Figure 3.5

Conceptual design of the system:

- Written in simple language i.e. customer understandable language.
- Detailed explanation about system characteristics.
- Describes the functionality of the system.
- It is independent of implementation.
- Linked with requirement document.

Technical Design of the System:

- Hardware component and design.
- Functionality and hierarchy of software components.
- Software architecture
- Network architecture
- Data structure and flow of data.

- I/O component of the system.
- Shows interface.

Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again.

Types of Coupling

Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

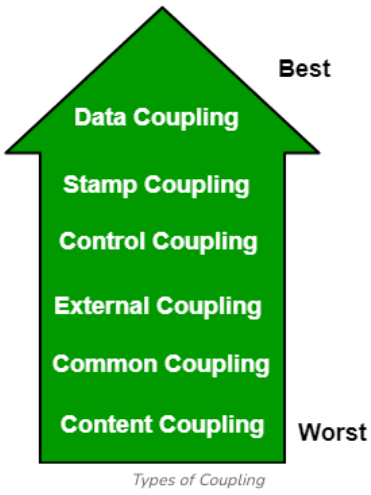


Figure 3.6

Following are the types of Coupling:

Data Coupling: If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.

Stamp Coupling: In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.

Control Coupling: If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.

External Coupling: In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

Common Coupling: The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.

Content Coupling: In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Temporal Coupling: Temporal coupling occurs when two modules depend on the timing or order of events, such as one module needing to execute before another. This type of coupling can result in design issues and difficulties in testing and maintenance.

Sequential Coupling: Sequential coupling occurs when the output of one module is used as the input of another module, creating a chain or sequence of dependencies. This type of coupling can be difficult to maintain and modify.

Communicational Coupling: Communicational coupling occurs when two or more modules share a common communication mechanism, such as a shared

message queue or database. This type of coupling can lead to performance issues and difficulty in debugging.

Functional Coupling: Functional coupling occurs when two modules depend on each other's functionality, such as one module calling a function from another module. This type of coupling can result in tightly-coupled code that is difficult to modify and maintain.

Data-Structured Coupling: Data-structured coupling occurs when two or more modules share a common data structure, such as a database table or data file. This type of coupling can lead to difficulty in maintaining the integrity of the data structure and can result in performance issues.

Interaction Coupling: Interaction coupling occurs due to the methods of a class invoking methods of other classes. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods. Coupling is lowest if methods communicate directly through parameters.

Component Coupling: Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist as to how this can happen. A class C can be component coupled with another class C1, if C has an instance variable of type C1, or C has a method whose parameter is of type C1,or if C has a method which has a local variable of type C1. It should be clear that whenever there is component coupling, there is likely to be interaction coupling.

Types of Cohesion

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.

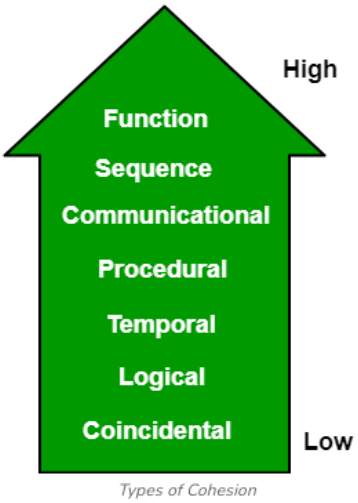


Figure 3.7

Following are the Types of Cohesion:

Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.

Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.

Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.

Procedural Cohesion: Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

Logical Cohesion: The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for

these functions is in the same component. Operations are related, but the functions are significantly different.

Coincidental Cohesion: The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component.

Procedural Cohesion: This type of cohesion occurs when elements or tasks are grouped together in a module based on their sequence of execution, such as a module that performs a set of related procedures in a specific order. Procedural cohesion can be found in structured programming languages.

Communicational Cohesion: Communicational cohesion occurs when elements or tasks are grouped together in a module based on their interactions with each other, such as a module that handles all interactions with a specific external system or module. This type of cohesion can be found in object-oriented programming languages.

Temporal Cohesion: Temporal cohesion occurs when elements or tasks are grouped together in a module based on their timing or frequency of execution, such as a module that handles all periodic or scheduled tasks in a system. Temporal cohesion is commonly used in real-time and embedded systems.

Informational Cohesion: Informational cohesion occurs when elements or tasks are grouped together in a module based on their relationship to a specific data structure or object, such as a module that operates on a specific data type or object. Informational cohesion is commonly used in object-oriented programming.

Functional Cohesion: This type of cohesion occurs when all elements or tasks in a module contribute to a single well-defined function or purpose, and there is little or no coupling between the elements. Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code.

Layer Cohesion: Layer cohesion occurs when elements or tasks in a module are grouped together based on their level of abstraction or responsibility, such as a module that handles only low-level hardware interactions or a module that handles only high-level business logic. Layer cohesion is commonly used in large-scale software systems to organize code into manageable layers.

Advantages of Low coupling

Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.

Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.

Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

Advantages of High cohesion

Improved readability and understandability: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.

Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to

Improved reliability: High cohesion leads to modules that are less prone to errors and that function more consistently, leading to an overall improvement in the reliability of the system.

Disadvantages of High coupling

Increased complexity: High coupling increases the interdependence between modules, making the system more complex and difficult to understand.

Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.

Decreased modularity: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

Disadvantages of Low cohesion

Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.

Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.

Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

In conclusion, it's good for software to have low coupling and high cohesion. Low coupling means the different parts of the software don't rely too much on each other, which makes it safer to make changes without causing unexpected problems. High cohesion means each part of the software has a clear purpose and sticks to it, making the code easier to work with and reuse. Following these principles helps make software stronger, more adaptable, and easier to grow.

3.4 Design model

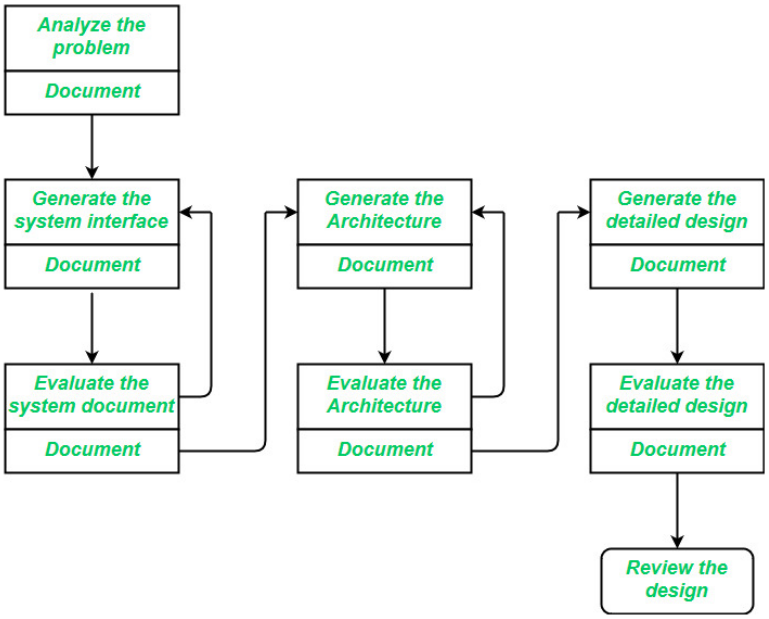


Figure 3.8

Good software design is built on several core element that work together to create effective system.

1. Architecture: This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture.

A solid architecture verify the system is flexible, stable, and easy to maintain over time.

2. Modules: Modules as the building blocks of the system. Each one handles a specific task or feature. Breaking a system into smaller modules makes it easier to develop, test, and maintain the system.

These are components that handle one specific task in a system. A combination of the modules makes up the system.

3. Components: This provides a particular function or group of related functions. They are made up of modules. Organising the system into components helps keep the code clean and makes the system more adaptable.

4. Interfaces: These are smaller units within modules that focus on specific functions. This is the shared boundary across which the components of a system exchange information and relate.

5. Data: Data is at the heart of any system. It's all about how information is stored, accessed, and shared. This is the management of the information and data flow.

How Software Design Fits into the SDLC?

Software Design comes when the project requirement are done and when we are about to start the Development process, He.re is How Software Design fits into the Software Development Life Cycle.

After planning, the designing of software phase starts. This is where the team uses tools like wireframes, data flow diagrams, and UI designs to map out how the system will work. It's like drawing the roadmap for how everything will come together and how we can use each component well with

proper functionality. After the Designing phase the Actual process of Software Development begins.

Software Design Principles

To make sure software is easy to manage, maintain, and update, there are a few important principles that should guide the design process:

Modularity: Think of modularity as breaking down the software into smaller, independent sections or "modules". Each module handles a specific task, which makes it much easier to test, maintain, and update without affecting the entire system.

Coupling: Coupling refers to how much one part of the software depends on others. The goal is to keep the connections between modules to a minimum, so that changing one module won't impact others. This makes the software more flexible and easier to update in the future.

Abstraction: Abstraction is about simplifying the software by hiding its complexity. It only shows users the essential features they need, making the software easier to use and understand, without overwhelming them with unnecessary details.

Anticipation of Change: It's important to design software with future changes in mind. This means building it in a way that allows for easy updates or adjustments, whether it's adding new features or adapting to new technologies.

Simplicity: Simple designs are the best. Keeping things simple means less room for errors, and it's easier to maintain and improve over time. The goal is to avoid overcomplicating things, keeping the design clean and efficient.

Sufficiency & Completeness: The design should make sure the software meets all the required functions without unnecessary extras. It's about striking a balance making sure everything needed is there, but avoiding unnecessary features that can slow things down or create confusion.

Tools for Software Design

There are many tools that make the process easier and more efficient, whether you're working on wireframes, prototypes, or documentation. Here are some popular tools:

- Figma
- Balsamiq
- Axure RP
- Sketch
- InVision Studio

3.5 Architectural Design

The software needs an architectural design to represent the design of the software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles.

System Category Consists of

- A set of components(eg: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that defines how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.

The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural Styles

Data centered architectures:

- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete, or modify the data present within the store.
- The figure illustrates a typical data-centered style. The client software accesses a central repository. Variations of this approach are used to transform the repository into a blackboard when data related to the client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using the blackboard mechanism.

Advantages of Data centered architecture:

- Repository of data is independent of clients
- Client work independent of each other
- It may be simple to add additional clients.
- Modification can be very easy

Figure 3.9

Data flow architectures:

- This kind of architecture is used when input data is transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by lines.
- Pipes are used to transmitting data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

Advantages of Data Flow architecture:

- It encourages upkeep, repurposing, and modification.
- With this design, concurrent execution is supported.

Disadvantage of Data Flow architecture:

- It frequently degenerates to batch sequential system
- Data flow architecture does not allow applications that require greater user engagement.
- It is not easy to coordinate two different but related streams

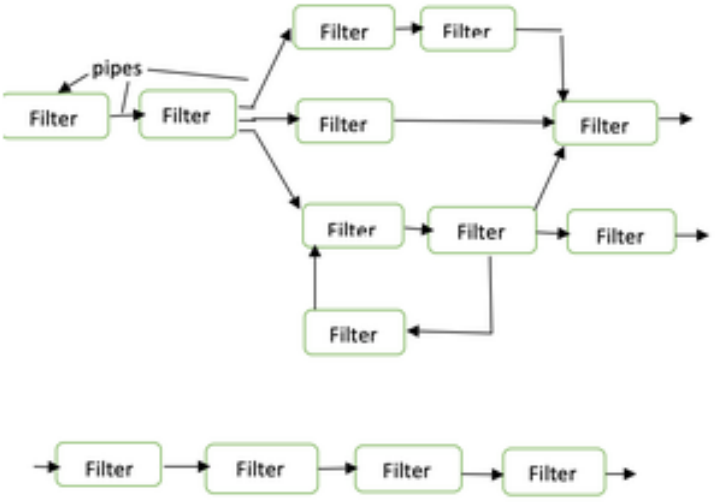


Figure 3.10

Call and Return architectures

- It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.
- Remote procedure call architecture: This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- Main program or Subprogram architectures: The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

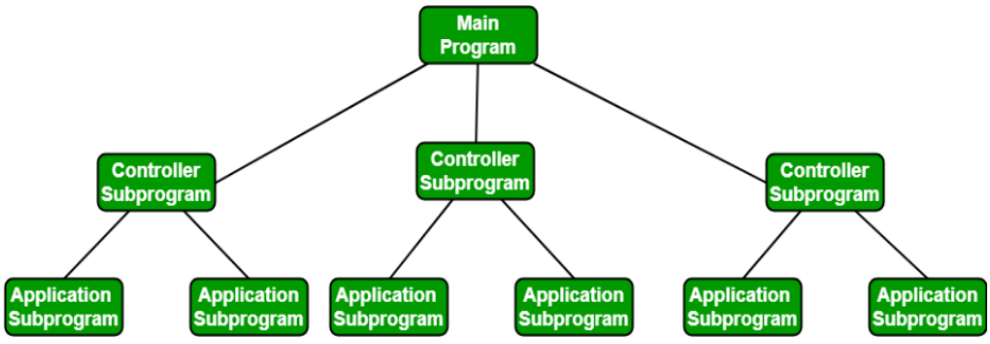


Figure 3.11

Object Oriented architecture

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

Characteristics of Object Oriented architecture:

- Object protect the system's integrity.
- An object is unaware of the depiction of other items.

Advantage of Object Oriented architecture:

- It enables the designer to separate a challenge into a collection of autonomous objects.
- Other objects are aware of the implementation details of the object, allowing changes to be made without having an impact on other objects.

Layered architecture

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
- Intermediate layers to utility services and application software functions.
- One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.

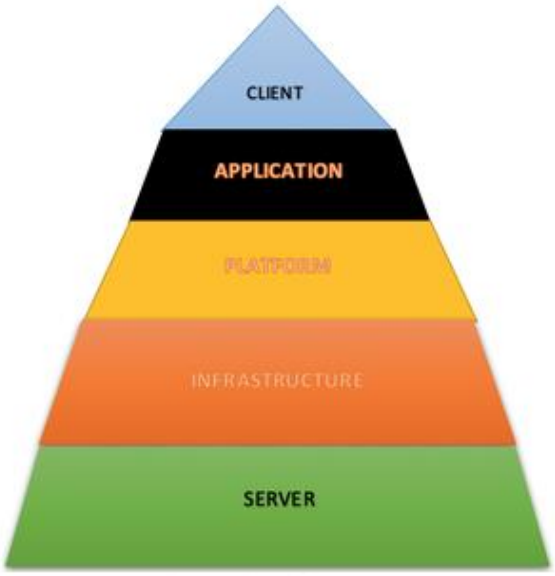


Figure 3.12

3.6 Architectural styles

Software architecture is like the blueprint for building software, showing how different parts fit together and interact. It helps the development team understand how to build the software according to customer requirements. There are many ways to organize these parts, called software architecture patterns. These patterns have been tested and proven to solve different problems by arranging components in specific ways.

1. Layered Architecture Pattern

As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another. Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others. It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.

- Presentation layer: The user interface layer where we see and enter data into an application.)

- Business layer: This layer is responsible for executing business logic as per the request.)
- Application layer: This layer acts as a medium for communication between the 'presentation layer' and 'data layer'.
- Data layer: This layer has a database for managing data.)

Advantages:

- Scalability: Individual layers in the architecture can be scaled independently to meet performance needs.
- Flexibility: Different technologies can be used within each layer without affecting others.
- Maintainability: Changes in one layer do not necessarily impact other layers, thus simplifying the maintenance.

Disadvantages:

- Complexity: Adding more layers to the architecture can make the system more complex and difficult to manage.
- Performance Overhead: Multiple layers can introduce latency due to additional communication between the layers.
- Strict Layer Separation: Strict layer separation can sometimes lead to inefficiencies and increased development effort.

Use Cases:

- Enterprise Applications like Customer Relationship Management (CRM).
- Web Applications like E-commerce platforms.
- Desktop Applications such as Financial Software.
- Mobile Applications like Banking applications.
- Content Management Systems like Wordpress.

2. Client-Server Architecture Pattern

The client-server pattern has two major entities. They are a server and multiple clients. Here the server has resources(data, files or services) and a client

requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Advantages:

- Centralized Management: Servers can centrally manage resources, data, and security policies, thus simplifying the maintenance.
- Scalability: Servers can be scaled up to handle increased client requests.
- Security: Security measures such as access controls, data encryption can be implemented in a better way due to centralized controls.

Disadvantages:

- Single Point of Failure: Due to centralized server, if server fails clients lose access to services, leading loss of productivity.
- Costly: Setting up and maintaining servers can be expensive due to hardware, software, and administrative costs.
- Complexity: Designing and managing a client-server architecture can be complex.

Use Cases:

- Web Applications like Amazon.
- Email Services like Gmail, Outlook.
- File Sharing Services like Dropbox, Google Drive.
- Media Streaming Services like Netflix.
- Education Platforms like Moodle.

3. Event-Driven Architecture Pattern

Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events. When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.

Example:

A new user fills the signup form and clicks the signup button on Facebook and then a FB account is created for him, which is an event.

Advantages:

- Scalability: System can scale horizontally by adding more consumers.
- Real-time Processing: This enables real-time processing and immediate response to events.
- Flexibility: New event consumers can be added without modifying existing components.

Disadvantages:

- Complexity: The architecture can be complex to design, implement, and debug.
- Complex Testing: Testing event-driven systems can be complicated compared to synchronous systems.
- Reliability: Ensuring reliability requires additional mechanisms to handle failed events.

Use Cases:

- Real-Time Analytics like stock market analysis systems.
- IoT Applications like smart home systems.
- Financial Systems like fraud detection systems monitor transactions in real-time.
- Online multiplayer games.
- Customer Support Systems like Chatbots.

4. Microkernel Architecture Pattern

- Microkernel pattern has two major components. They are a core system and plug-in modules.
- The core system handles the fundamental and minimal operations of the application.
- The plug-in modules handle the extended functionalities (like extra features) and customized processing.

Let's imagine, you have successfully built a chat application. And the basic functionality of the app is that you can text with people across the world

without an internet connection. After some time, you would like to add a voice messaging feature to the application, then you are adding the feature successfully. You can add that feature to the already developed application because the microkernel pattern facilitates you to add features as plug-ins.

Advantages:

- Flexibility: New functionalities can be added easily through plug-ins.
- Scalability: The system can scale by adding more plug-ins to handle more tasks.
- Maintainability: Plug-ins are developed and tested independently which makes maintenance easier.

Disadvantages:

- Complex Communication: Managing communication between the core systems and plug-ins can be complex.
- Lack Built-in Functionalities: Due to minimalistic design the basic functionalities are absent that are common in monolithic architectures.
- Complex Design: Designing the microkernel and its communication mechanisms can be challenging.

Use Cases:

- Operating Systems like Windows NT and macOS.
- Embedded Systems like Automotive Software Systems.
- Plugin-based Applications like Eclipse IDE.

5. Microservices Architecture Pattern

The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application. So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern. Modules in the

application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

Advantages:

- Scalability: Each service can be scaled independently based on demand.
- Faster Delivery: Independent services allows teams to develop, test, and deploy features faster.
- Easier Maintenance: Services can be updated and maintained independently.

Disadvantages:

- Complex Management: Managing multiple services requires robust monitoring and management tools.
- Network Congestion: Increased network traffic between services can lead to congestion and overhead.
- Security: Securing multiple services and their communication increases the probability of attack.

Use Cases:

- E-commerce Platforms like Amazon and eBay.
- Streaming services like Netflix and Spotify.
- Online Banking Platforms.
- Electronic Health Record (EHR) Systems.
- Social Media Platforms like Twitter and Facebook.

6. Space-Based Architecture Pattern

Space-Based Architecture Pattern is also known as Cloud-Based or Grid-Based Architecture Pattern. It is designed to address the scalability issues associated with large-scale and high-traffic applications. This pattern is built around the concept of shared memory space that is accessed by multiple nodes.

Advantages:

- Scalability: The system can be easily scaled horizontally by adding more processing units.
- Performance: In-memory data grids reduces the data access latency.
- Flexibility: Modular components allow for flexible deployment.

Disadvantages:

- Complexity: Designing and managing distributed system is complex.
- Cost: The infrastructure for space-based architecture pattern requires multiple servers and advanced middleware, which can be expensive.
- Network Latency: Communication between distributed components can introduce network latency.

Use Cases:

- E-commerce Platforms like Amazon.
- Telecom Service Providers.
- Multiplayer Online Games.

7. Master-Slave Architecture Pattern

The Master-Slave Architecture Pattern is also known as Primary-Secondary Architecture. It involves a single master component and that controls multiple slave components. The master components assign tasks to slave components and the slave components report the results of task execution back to the master. This is often used for parallel processing and load distribution.

Advantages:

- Scalability: The system can scale horizontally by adding more slave units to handle increased load.
- Fault Tolerance: If slave fails, then the master can reassign the tasks to some another slave. thus enhancing the fault tolerance.
- Performance: Parallel execution of tasks can improve the performance of the system.

Disadvantages:

- Single Point of Failure: The master component is a single point of failure. If the master fails then the entire system can collapse.
- Complex Communication: The communication overhead between master and slave can be significant especially in large systems.
- Latency: Systems' responsiveness can be affected by the latency introduced by master-slave communication.

Use Cases:

- Database Replication.
- Load Balancing.
- Sensor Networks.
- Backup and Recovery Systems.

8. Pipe-Filter Architecture Pattern

Pipe-Filter Architecture Pattern structures a system around a series of processing elements called filters that are connected by pipes. Each filter processes data and passes it to the next filter via pipe.

Advantages:

- Reusability: Filters can be reused in different pipelines or applications.
- Scalability: Additional filters can be added to extend the functionality to the pipeline.
- Parallelism: Filters can be executed in parallel if they are stateless, thus improving performance.

Disadvantages:

- Debugging Difficulty: Identifying and debugging issues are difficult in long pipelines.
- Data Format constraints: Filters must agree on the data format, requiring careful design and standardization.
- Latency: Data must be passed through multiple filters, which can introduce latency.

Use Cases:

- Data Processing Pipelines like Extract, Transform, Load (ETL) processes in data warehousing.
- Compilers.
- Stream-Processing like Apache Flink.
- Image and Signal Processing.

9. Broker Architecture Pattern

The Broker architecture pattern is designed to manage and facilitate communication between decoupled components in a distributed system. It involves a broker that acts as a intermediary to route the requests to the appropriate server.

Advantages:

- Scalability: Brokers support horizontal scaling by adding more servers to handle increased load.
- Flexibility: New servers can be added and the existing ones can be removed or modified without impacting the entire system.
- Fault Tolerance: If a server fails, then broker can route the request to another server.

Disadvantages:

- Complex Implementation: Implementing a broker requires robust management of routing and load balancing, thus make system more complex.
- Single Point of Failure: If broker is not designed with failover mechanisms then it can become a single point of failure.
- Security Risks: Securing broker component is important to prevent potential vulnerabilities.

Use Cases:

- Integration of various enterprise applications like CRM, ERP, and HR systems.
- Systems using message brokers like RabbitMQ or Apache Kafka.

- Sensor networks in IoT applications.

10. Peer-to-Peer Architecture Pattern

The Peer-to-Peer (P2P) architecture pattern is a decentralized network model where each node, known as a peer, acts as both a client and a server. There is no central authority or single point of control in P2P architecture pattern. Peers can share resources, data, and services directly with each other.

Advantages:

- Scalability: The network can scale easily as more peers join.
- Fault Tolerance: As data is replicated across multiple peers, this results in system being resilient to failures.
- Cost Efficiency: There is no need for centralized servers, thus reducing the infrastructure cost.

Disadvantages:

- Security Risks: Decentralized nature of the architecture makes it difficult to enforce security policies.
- Data Consistency: Ensuring data consistency across peers can be challenging.
- Complex Management: Managing a decentralized network with numerous independent peers can be complex.

Use Cases:

- File Sharing like BitTorrent Protocol.
- Blockchain and Cryptocurrencies such as Bitcoin and Ethereum.
- VoIP and Communication like Skype.

In conclusion, software architecture patterns are essential for designing software that meets specific needs and challenges. The Layered Pattern is great for e-commerce sites with its clear separation of tasks. The Client-Server Pattern works well for centralized resources like email and banking systems. The Event-Driven Pattern is perfect for applications that react to user actions. The Microkernel Pattern allows easy addition of new features. Finally, the Microservices Pattern helps build scalable and flexible applications, like

Netflix. Choosing the right pattern is key to making the software adaptable, maintainable, and successful.

3.7 Architecture for Network based Applications

Network Architecture is the design of a computer or communication network that shows how devices, software, protocols, and media work together. It defines the physical and logical structure, task allocation, and connectivity between clients like laptops and servers. Simply put, it provides the rules and tools for smooth communication, managed by network architects.

Network Architect vs Network Administrator vs Network Engineer

Network Administrator: They take care of a network that's already set up. They handle day-to-day tasks like fixing problems, adding users, and making sure everything runs smoothly.

Network Engineer: They're like the builders and fixers. They create networks based on designs, make changes when needed, and troubleshoot any issues that pop up.

Network Architect: They're the big planners. They design how a network should look and work. They create the blueprint that the engineers follow to build the network

Types of Network Architecture

Computer networks can be classified based on architecture into two primary types:

- Peer-to-Peer Architecture
- Client/Server Architecture
- Hybrid Networks

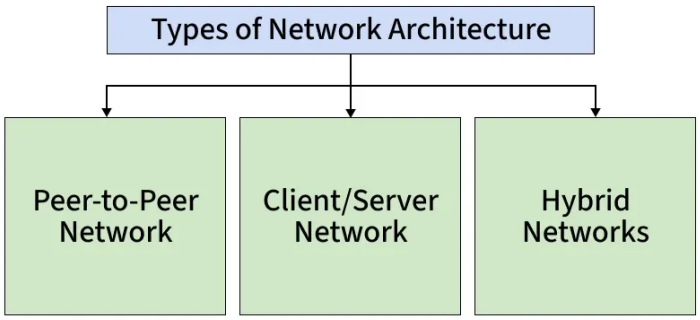


Figure 3.13

1. Peer-to-Peer Architecture

In a P2P network, computers (peers) are directly connected, usually via the Internet, to share files and resources without a central server. Each computer acts as both client and server, with equal roles and responsibilities. Tasks are distributed across all devices, making P2P suitable for small setups (up to about 10 computers). Since there's no strict client-server division, peers can both send and receive data directly. P2P networks are commonly used in business, education, and military applications.

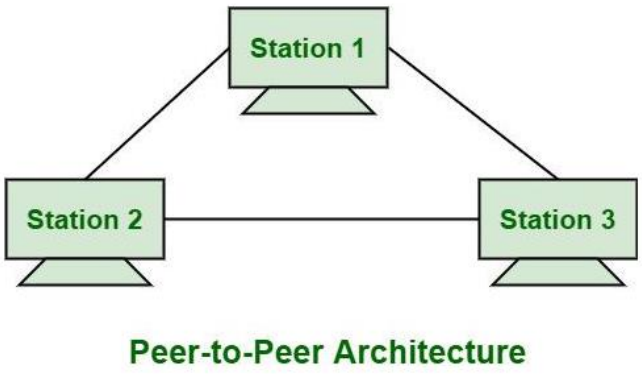


Figure 3.14

Advantages of Peer-to-Peer Architecture

- P2P network is less costly and cheaper. It is affordable.
- P2P is very simple and not complex. This is because all computers that are connected in network communication in an efficient and well-mannered with each other.

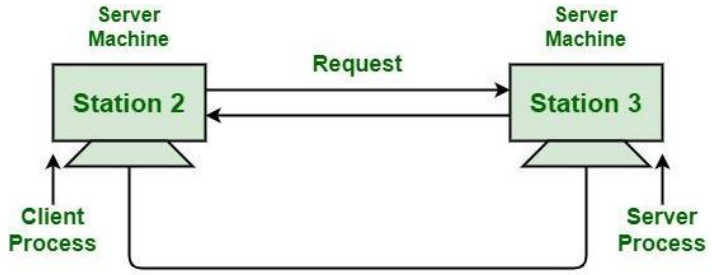
- It is very easy and simple to set up and manage as installation and setup is less painless and computer manages itself. This is because of built-in support in modern operating systems.

Disadvantages of Peer-to-Peer Architecture

- It is more difficult to manage security policies consistently.
- Each peer demands individual care and control.
- As the network expands in size, it may become inefficient.

2. Client/Server Architecture

CSN (Client/Server Network) is type of computer network in which one of centralized and powerful computers (commonly called as server) is hub to which many of personal computers that are less powerful or workstations (commonly known as clients) are connected. It is type of system where clients are connected to server to just share or use resources. These servers are generally considered as heart of system. This type of network is more stable and scalable as compared to P2P network. In this architecture, system is generally decomposed into client and server processor or processes.



Client/Server Architecture

Figure 3.15

Advantages of Client/Server Architecture

- A special Network Operating System (NOS) is provided by server to provide resources to many users that request them.
- It is also very easy and simple to set up and manage data updates. This is because data is generally stored in centralized manner on server.
- The server usually controls resources and data security.

Disadvantages of Client/Server Architecture

- If the server fails, clients may lose access to services.
- Setting up servers requires a higher investment in hardware and software.
- Managing servers requires skilled personnel.

3. Hybrid Networks

Hybrid networks combine elements of both client-server and peer-to-peer architectures. They leverage the centralized control of client-server networks and the decentralized resource sharing of P2P networks. An example is a torrent network, where a central tracker (server) coordinates peers, but file sharing occurs directly between peers. Hybrid networks are used in applications requiring both centralized management and distributed resource sharing.

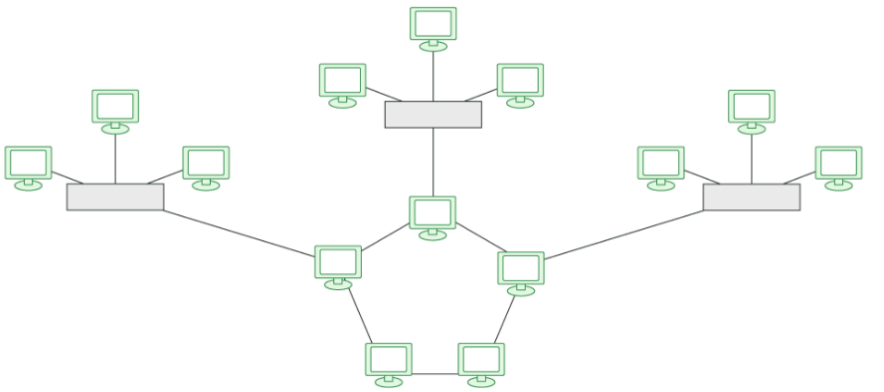


Figure 3.15

Advantages of Hybrid Networks:

- Combines the benefits of centralized control and decentralized resource sharing.
- Flexible, allowing adaptation to different use cases and requirements.
- Can provide redundancy and fault tolerance by distributing tasks across peers and servers.

Disadvantages of Hybrid Networks:

- Complex to design and maintain due to the combination of two architectures.
- May face security challenges from both client-server and P2P components.
- Higher costs compared to pure P2P networks due to the need for server infrastructure.

What Does a Computer Network Architect Do?

A computer network architect is responsible for designing and building communication networks for organizations. Their main tasks include:

Designing, Modeling, Testing, and Troubleshooting Networks: This involves creating plans for networks, testing them out, and solving any problems that arise.

Testing and Inspecting Existing Networks: You'll need to examine current networks to ensure they're working correctly and identify any issues that need fixing.

Upgrading Networks (Hardware and Software): Keeping networks up-to-date by improving both the physical equipment and the software they run on.

Analyzing and Fixing Security Weaknesses: Identifying and resolving any vulnerabilities in networks to keep them safe from cyber threats.

Developing Technical Documentation: Creating detailed guides and instructions for building and maintaining networks.

Installing and Maintaining Hardware Components: Setting up and looking after the physical parts of networks, like routers, cables, and adapters.

What Skills Does a Computer Network Architect Need?

To be successful as a computer network architect, you need a mix of technical and interpersonal skills. Here's what you should have:

Design and Modeling Skills: You'll be creating and visualizing network systems, so being adept at designing and modeling is crucial.

Cybersecurity Knowledge: Ensuring network security is paramount. You must be alert against threats and know how to support the network against potential attacks.

Technical Equipment Proficiency: Understanding the hardware components of networks like servers, routers, and modems is essential for implementing effective designs.

Soft Skills: In addition to technical prowess, you'll need soft skills to excel. Problem-solving skills are vital for troubleshooting issues that arise in complex networks. Effective communication skills are necessary for collaborating with other IT professionals and sometimes providing training to users.

3.8 Decentralized Architectures

Decentralized architecture means a conceptual design of a system's components as elements that are mutually integrated and act according to the general principles of a system without a specific coordinative center.

In such systems, the control and data are present in different nodes and each node is independent and can make decisions independently. This architecture improves the system's capability on scalability, fault tolerance and robustness in eliminating a single centralized point of failure and integrating peers to work together. Distribution is a usual practice in the blockchain networks, P2P systems, and distributed ledger technology field.

Key Concepts in Decentralized Systems

Below are the key concepts of distributed systems:

Peer-to-Peer (P2P) Networks: the distributed network in which most of the communicating entities have equal capabilities, and each peer is on an equal level, using the resources of other peers directly without any interference from the hub. These include; the Bit Torrent and the Block chain networks.

Consensus Mechanisms: A procedure applied to solve a consistency problem in distributed computing, to have a unique value or state among the

participating processes or systems. They also apply decentralization procedures to make sure they are not only uniform and safe as well as trustworthy.

Decentralized Autonomous Organizations (DAOs): Organizations depicted by rules written in a program that is, clear; open for modification by organization members, and not dominated by central government. A smart contract is a digital contract that exists on a blockchain and DAOs are fully automated.

Data Replication: The write operation of creating backups of the data, which is committed to several nodes to warrant its accessibility and integrity. The consensus helps in providing a certain level of repetitiveness, and error tolerance within the decentralized applications.

Benefits of Decentralized Architecture

Below are the benefits of decentralized architecture in distributed systems:

Scalability: Decentralized systems can easily scale up by adding more nodes and, therefore, do not have significant alterations to their structures. This enables the system to serve larger amounts of loads and users without making any considerable impact.

Fault Tolerance and Resilience: Another crucial advantage of these distributed systems is that they are less vulnerable to node failures than centralized systems. If one node of the framework fails at that particular point of time then others can work, thus making the application highly available and reliable.

Enhanced Security: Such structures are characterized by the use of very strong encryption and consensus mechanisms; it will not be easy for an attacker to penetrate the system. The total distribution of data and control works well in creating and maintaining security.

Resource Optimization: Decentralized systems take advantage of multiple nodes and hence result in optimal utilization of resources such as computational power, storage, and bandwidth. This can lead to efficiency and effectiveness in the organization, meaning that costs are cut and performance is optimised.

Reduced Latency: Compared to centralized systems, decentralized systems may be able to decrease response time by sending out data and services closer to the customers and use since it decreases latency.

Challenges of Decentralized Architecture

Below are the challenges of decentralized architecture in distributed systems:

Complexity in Design and Implementation: Decentralized systems call for coordinated, reliable, and consistent design of distributed nodes, and therefore the designs implied by their working entail such complexity. Assessing the actualization of these systems entails dealing' with algorithms and protocols and this is not easy.

Data Consistency: It is often challenging to ensure that data created and stored in nodes at different locations will remain consistent. The fact that all the nodes have to keep the same view of the data, let alone in real-time, typically entails complicated consensus work.

Network Latency and Bandwidth: The communication between nodes can add a certain amount of latency and consume a lot of bandwidth. This can have an impact on its performance particularly for applications that are heavily dependent on real-time processing.

Resource Management: Coordinating the use of resources including storage, processing power and energy in such distributed nodes may be difficult. Managing the load and preventing one node from overloading is a very challenging process that requires special approaches.

Governance and Coordination: Decentralized systems do not feature a centralized body of control in some and or any usually it becomes very hard to implement certain policies, manage updates and surmount certain conflicts relating to the system. This may cause issues in the struggle to sustain the integrity as well as the consistency of the system.

Use Cases of Decentralized Architecture

Below are the use cases of decentralized architecture in distributed system:

Decentralized Autonomous Organizations (DAOs): Smart contracts on the blockchain networks help DAOs for decentralised control and management. It is easy to control and facilitate since the participants would make decisions and also control resources on their own.

Supply Chain Management: Decentralized systems can increase transparency and versioning in supply chains together with increasing the immunity to failure. Through the use of blockchain, it is shown that; recording of products' movements in the blockchain helps in the verification of the products.

Content Distribution Networks (CDNs): Decentralized CDNs take the content, and divide it among various nodes, to increase service time and decrease server strain. Theta and Filecoin are examples of Decentralized CDN to some degree.

IoT Networks: Decentralization approaches can improve the robustness and the accommodating capacity of IoT networks through the elimination of centralized architectures for data processing and storage.

Decentralized Identity Management: Services such as uPort and Sovrin provide decentralized approaches to identity assurance where the user retains control over their identifiers and associated information without the need for a central entity.

Gaming and Virtual Worlds: Other games like Decentraland and Axie Infinity provide digital assets for in-game use, property, and commerce without the interference of a central hub, giving power to the players of the game's economies.

Important Decentralized Algorithms and Protocols

Below are some important decentralized algorithms and protocols in distributed systems:

1. Consensus Algorithms:

Proof of Work (PoW): Designed and implemented in cryptocurrencies like Bitcoin, PoW necessitates miners, who are nodes, to solve hard computational problems to confirm transactions and integrate new blocks in the blockchain

network. This process on the other hand guarantees security and eliminates the probability of double spending but it is costly when it comes to energy consumption.

Proof of Stake (PoS): Used by Ethereum 2. 0 and the other networks, PoS chooses the validators depending on their token amount and what they are willing to put up for staking. PoS also turned out to be less energy-consuming than PoW although it offers the same level of security.

Delegated Proof of Stake (DPoS): It is a form of PoS where ordinary token owners elect just a few individuals who perform transactions' validation and blockchain maintenance. This approach enhances the scalability and optimality of the system since it does not call for extensive use of databases.

Practical Byzantine Fault Tolerance (PBFT): An approach intended for consensus in distributed systems in the presence of malicious nodes. It is applied in systems such as Hyperledger Fabric to offer high throughput and low latency.

Raft: An algorithm used in distributed systems for the replication of a log. It is supposed to be comprehendible and is employed in etcd and Consul systems.

2. Routing Protocols:

Chord: A DHT is a decentralized lookup protocol that enables one to find other nodes and data in a P2P network based on CH. This depends on the number of instructions implemented, and its simplicity is admired by most people.

Kademlia: A DHT protocol employed in peer-to-peer systems like that of BitTorrent. It allows for the efficient forwarding/routing searching and access of a particular datum since nodes are organized in the structure of an overlay network.

Pastry: A large-scale, distributed object name and address space for Internet applications. It is a DHT that offers the routing of the distributed data in a highly efficient manner.

3. Consensus Mechanisms for Distributed Databases:

Paxos: Aerap is a family of protocols aimed at the agreement construction in a network of processors which can be considered unreliable. Paxos algorithm is commonly applied in distributed databases and systems to guarantee convergence and correctness.

Raft: Developed as a competitor to Paxos, but being markedly more comprehensible and extensible, Raft is a consensus algorithm. It is utilized for consensus serving such as in etcd and Consul for leader selection together with log synchronization.

4. Blockchain Protocols:

Bitcoin Protocol: The first kind of blockchain implementation where through PoW the consensus in the network is reached.

Ethereum Protocol: Expands the concept of blockchain with smart contracts and Decentralised applications (dApps). Ethereum at the start employed PoW, however, it plans to change over to PoS with Ethereum 2. 0.

Ripple Protocol: It implements the Ripple Protocol Consensus Algorithm (RPCA) that enables quick and secure transaction processing, especially in the financial sectors.

5. Communication Protocols:

Gossip Protocols: Used for broadcasting information across the decentralized network. Each node works independently to broadcast information with other nodes to enhance information transfer while minimizing error.

SCP (Stellar Consensus Protocol): Applicable in the Stellar network, SCP allows for direct, leaderless consensus with modular trust, which is suitable for financial projects.

Design Principles for Decentralized Architecture

Below are the design principles for decentralized architecture:

Decentralization: Ensure no centralized points are created because all the data, control, and processing are crucial and should be distributed across nodes. Every node is to be able to work on its own.

Scalability: Make sure data is presented in such a way that as the number of nodes and transactions increases the architecture is ready to accommodate them. Use a type of scaling known as horizontal scaling by incorporating more nodes into the system.

Consistency and Consensus: Use consensus algorithms such as Paxos, Raft or consensus mechanisms based on blockchain technology which is PoW or PoS to make sure all nodes are in consensus about the state of the system.

Security: Implement cryptography for protection of the messages being sent or received, storing of information and performing of a transaction. Incorporate measures to minimize the occurrence of negative actions.

Transparency and Auditability: They want all happenings associated with transactions and changes of state to be visible and traceable by all nodes. Record every transaction with an impenetrable ledger in digital technology such as blockchain.

Best Practices for Decentralized Architecture

Below are the best practices for decentralized architecture:

Node Distribution: Use nodes at different geographical areas to increase the level of redundancy as well as decrease the meantime for response. Use a cloud provider and/or a combination of local/on-premise and cloud nodes.

Data Partitioning and Replication: Shard data to distribute the load between nodes, and make replicates of the same data set for availability and fault tolerance purposes. Sharding and Distributed Hash Tables (DHTs) should be used.

Consensus Mechanism Choice: Select the agreement protocol that you want to implement based on your requirements of usage. For instance, PoW for extended security, PoS for low energy consumption and Raft or Paxos as the practical implementations of distributed databases.

Regular Audits and Monitoring: Periodically check the system's effectiveness, its vulnerability, and whether it meets the standards of the organization.

Employ logging, alerting, and auditing tools for monitoring the health status and for identifying signs of any error.

Smart Contract Security: For blockchain-based systems, perform robust testing of smart contracts to identify susceptibilities. Apply mathematical reasoning and get the service of auditors to improve the quality of contracts.

Tools and Frameworks for Decentralized Architecture

Below are some tools and framework for decentralized architecture:

Blockchain Platforms:

Ethereum: A decentralized digital marketplace for launching and running smart stories as well as decentralized applications. Complies with Solidity for contract development.

Hyperledger Fabric: An enterprise-grade permissioned blockchain platform for building applications. Support pluggable consensus and chain code, now clever contracts are also alluded to as chain code.

Corda: A blockchain solution for enterprises that focuses on providing privacy and the ability to scale.

Consensus Libraries:

Raft: A reasonable consensus algorithm for the replicated log. Examples of implementations are etched and HashiCorp Consul.

Distributed Storage:

IPFS (InterPlanetary File System): Protocol and computer communication network that stores and further shares hypermedia in a Decentralized File System.

Apache Cassandra: A primarily externally persisted, high-scale NoSQL database for distributed data management. It supports data duplication with the other nodes of the cluster.

Decentralized Communication:

libp2p: A system for constructing peer-to-peer applications where the latter is divided into a stack of modules. Implemented in IPFS and generally utilized in other decentralized projects.

Whisper: It should also be noted that Whisper is one of the protocols actively used in Ethereum to solve problems related to messaging while ensuring the anonymity and security of the participants.

Real-World Examples of Decentralized Architecture

Below are some real-world examples of decentralized architecture:

Bitcoin Blockchain:

- Bitcoin is an autonomous virtual currency that works on a distributed system that lacks a formal controlling body.
- The process of the transaction is confirmed and executed by the different nodes of the computer network and has a digital record known as a blockchain.

Filecoin:

- Filecoin can be considered a decentralized storage infrastructure based on the IPFS that pays users for storage.
- In this way, there is no single point of failure of storage as Filecoin implements a decentralized network of storage providers.
- To encourage storage and retrieval services, users compensate other clients based on FILs market price to make it market-driven rather than through a centralized system.

Hyperledger Fabric:

- Hyperledger Fabric is an enterprise-level freely available permissioned blockchain platform.
- Fabric, intended for business applications, enables organizations to construct and release applications based on blockchain technologies using Configurable Consensus and Membership Services.
- Ideally, places for its use are supply chain, finance, and healthcare where it can ensure secure, private transactions and data sharing among a consortium of known participants.

In conclusion, Decentralized architecture means a complete overhaul of how systems are designed and run, the distribution of components, structure and

function, fault tolerance and self-organizing capabilities. Here, consensus mechanisms increase the scalability of participating nodes while decentralised networks, and integrating elements such as peer-to-peer network protocols and complex cryptographic technologies improve the security and the networks the system's ability to withstand challenges.

UNIT IV

SOFTWARE TESTING

4.1 Software Testing Fundamentals

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. The complete testing includes identifying errors and bugs that cause future problems for the performance of an application.

Software Testing Can be Divided into Two Steps:

Software testing mainly divides into the two parts, which is used in the Software Development Process:

- Verification: This step involves checking if the software is doing what is supposed to do. Its like asking, "Are we building the product the right way?"
- Validation: This step verifies that the software actually meets the customer's needs and requirements. Its like asking, "Are we building the right product?"

Need for Software Testing

Software bugs can cause potential monetary and human loss. There are many examples in history that clearly depicts that without the testing phase in software development lot of damage was incurred. Below are some examples:

1985: Canada's Therac-25 radiation therapy malfunctioned due to a software bug and resulted in lethal radiation doses to patients leaving 3 injured and 3 people dead.

1994: China Airlines Airbus A300 crashed due to a software bug killing 264 people.

1996: A software bug caused U.S. bank accounts of 823 customers to be credited with 920 million US dollars.

1999: A software bug caused the failure of a $1.2 billion military satellite launch.

2015: A software bug in fighter plane F-35 resulted in making it unable to detect targets correctly.

2015: Bloomberg terminal in London crashed due to a software bug affecting 300,000 traders on the financial market and forcing the government to postpone the 3bn pound debt sale.

Starbucks was forced to close more than 60% of its outlet in the U.S. and Canada due to a software failure in its POS system.

Nissan cars were forced to recall 1 million cars from the market due to a software failure in the car's airbag sensory detectors.

Key Fundamentals of Software Testing

Objectives: The primary goals are to identify defects (bugs), improve software quality, verify that the system satisfies requirements, and ensure reliability before deployment.

Verification vs. Validation:

Verification: "Are we building the product right?" (Documents, design, code reviews).

Validation: "Are we building the right product?" (Executing the software to check if it works as intended).

Core Testing Methods:

Black Box Testing: Focuses on input/output without knowing the internal code structure.

White Box Testing: Involves testing the internal structure, logic, and code.

Levels of Testing:

Unit Testing: Testing individual components or code modules.

Integration Testing: Verifying that different modules work together.

System Testing: Testing the complete, integrated system to ensure compliance with requirements.

Acceptance Testing: Validating the system against user needs, often done before release.

Software Testing Lifecycle (STLC):

Test Planning: Defining the scope and strategy.

Test Case Development: Creating test cases based on requirements.

Test Execution: Running tests and recording results.

Defect Tracking: Identifying, logging, and fixing bugs.

Testing Approaches:

Manual Testing: Human testers execute tests without tools.

Automated Testing: Using tools (e.g., Selenium, JUnit) to run repetitive tests.

Key Testing Types:

Functional Testing: Validates what the system does.

Non-Functional Testing: Evaluates performance, security, and usability.

Regression Testing: Ensures new code changes do not break existing functionality.

Common Seven Principles of Testing

- Testing shows the presence of defects, not their absence.
- Exhaustive testing is impossible.
- Early testing saves time and money.
- Defect clustering (most bugs are in a few modules).
- Pesticide paradox (repeating tests won't find new bugs).
- Testing is context-dependent.
- Absence-of-errors fallacy (a bug-free system is useless if it doesn't meet user needs).

Best Practices for Software Testing

Here are the best practices for software testing that help to verify the testing process:

Continuous Testing: Project teams test each build as it becomes available thus it enables software to be validated in real environments earlier in the development cycle, reducing risks and improving the functionality and design.

Involve Users: It is very important for the developers to involve users in the process and open-ended questions about the functionality required in the application. This will help to develop and test the software from the customer's perspective.

Divide Tests into Smaller Parts: Dividing tests into smaller fractions save time and other resources in environments where frequent testing needs to be conducted. This also helps teams to make better analyses of the tests and the test results.

Metrics and Reporting: Reporting enables the team members to share goals and test results. Advanced tools integrate the project metrics and present an integrated report in the dashboard that can be easily reviewed by the team members to see the overall health of the project.

Don't Skip Regression Testing: Regression testing is one of the most important steps as it encourages the validation of the application. Thus, it should not be skipped.

Programmers Should Avoid Writing Tests: Test cases are usually written before the start of the coding phase so it is considered a best practice for programmers to avoid writing test cases as they can be biased towards their code and the application.

Service Virtualization: Service virtualization simulates the systems and services that are not yet developed or are missing. Thus, enabling teams to reduce dependency and start the testing process sooner. They can modify, and reuse the configuration to test different scenarios without having to alter the original environment.

4.2 Internal and External Views of Testing

Here are the Types of Software Testing mainly categorized into the two domain, which are below.

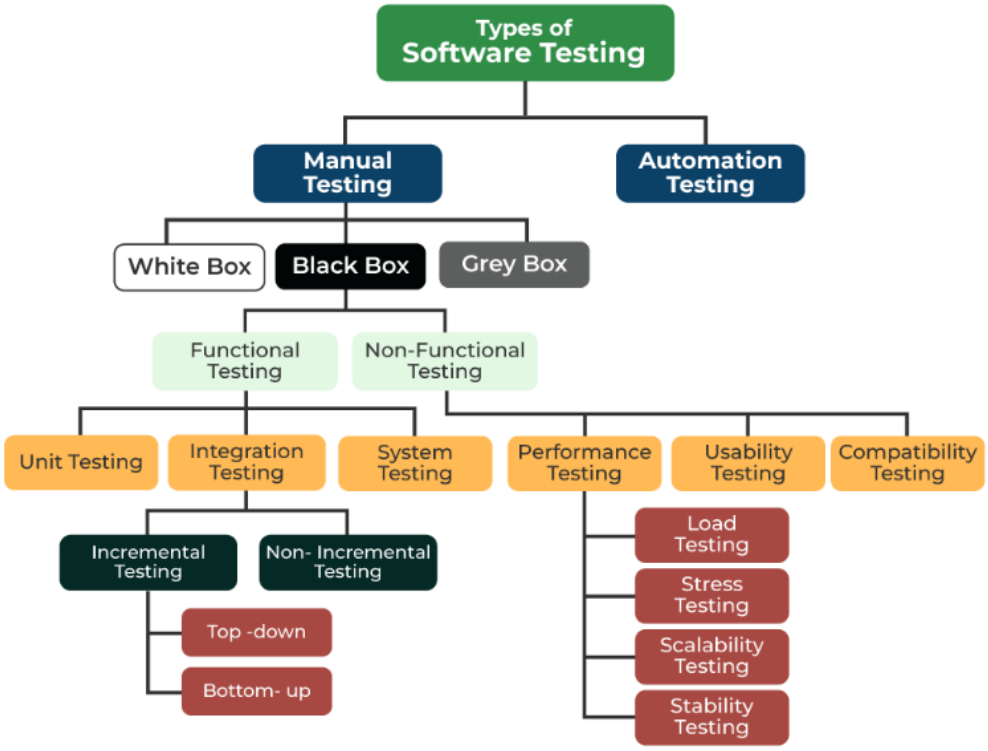


Figure 4.1

1. Manual Testing

Manual Testing is a technique to test the software that is carried out using the functions and features of an application. Which means manual testing will check the defect manually with trying one by one function is working as expected.

2. Automation Testing

Automation Testing It is a technique where the Tester writes scripts independently and uses suitable Software or Automation Tools to test the software. It is an Automation Process of a Manual Process. It allows for executing repetitive tasks without the use of a Manual Tester.

Types of Functional Testing

Functional Testing will be divided into further types which is following:

1. Unit Testing

Unit Testing is a method of testing individual units or components of a software application. It is typically done by developers and is used to ensure that the individual units of the software are working as intended.

2. Integration Testing

Integration Testing is a method of testing how different units or components of a software application interact with each other. It is used to identify and resolve any issues that may arise when different units of the software are combined.

3. System Testing

System Testing is a type of software testing that evaluates the overall functionality and performance of a complete and fully integrated software solution. It tests if the system meets the specified requirements and if it is suitable for delivery to the end-users.

Types of Non-functional Testing

Here are the Types of Non-Functional Testing

1. Performance Testing

Performance Testing is a type of software testing that ensures software applications perform properly under their expected workload. It is a testing technique carried out to determine system performance in terms of sensitivity, reactivity, and stability under a particular workload.

2. Usability Testing

Usability Testing in software testing is a type of testing, that is done from an end user's perspective to determine if the system is easily usable. Usability testing is generally the practice of testing how easy a design is to use on a group of representative users.

3. Compatibility Testing

Compatibility Testing is software testing that comes under the non functional testing category, and it is performed on an application to check its

compatibility (running capability) on different platforms/environments. This testing is done only when the application becomes stable.

Different Levels of Software Testing

In Software testing there are Different Levels of Testing can be majorly classified into 4 levels:

Unit Testing: In this type of testing, errors are detected individually from every component or unit by individually testing the components or units of software to ensure that they are fit for use by the developers. It is the smallest testable part of the software.

Integration Testing: In this testing, two or more modules which are unit tested are integrated to test i.e., technique interacting components, and are then verified if these integrated modules work as per the expectation or not, and interface errors are also detected.

System Testing: In system testing, complete and integrated Software are tested i.e., all the system elements forming the system are tested as a whole to meet the requirements of the system.

Acceptance Testing: This is a kind of testing conducted to ensure that the requirements of the users are fulfilled before its delivery and that the software works correctly in the user's working environment.

4.3 White box testing

White box testing is a Software Testing Technique that involves testing the internal structure and workings of a Software Application. The tester has access to the source code and uses this knowledge to design test cases that can verify the correctness of the software at the code level.

White box testing is also known as Structural Testing or Code-based Testing, and it is used to test the software's internal logic, flow, and structure. The tester creates test cases to examine the code paths and logic flows to ensure they meet the specified requirements.

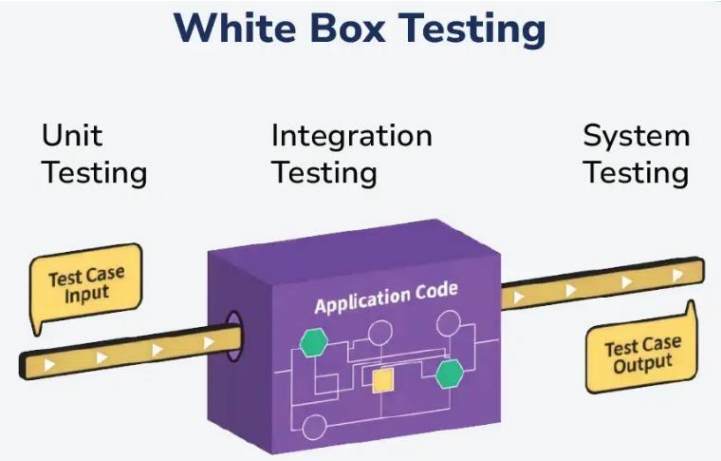


Figure 4.2

What Does White Box Testing Focus On?

White-box Testing focuses on the internal workings of an application, ensuring that its logic, structure, and flow operate as intended. Unlike black-box testing, which focuses on user interactions without knowledge of the underlying code, white-box testing involves examining the software's source code directly. Below are the key areas of focus in white-box testing:

1. Code Logic and Flow

Checks if the program's logic works as intended. This means verifying that code modules (like functions or classes) interact correctly and that control structures such as if-else statements, loops, or switches execute properly. For example, ensuring a login function redirects users correctly based on valid or invalid credentials.

2. Code Coverage

Ensures tests exercise as much of the code as possible. This includes:

- Statement coverage: Every line of code runs at least once.
- Branch coverage: All decision paths (e.g., true/false conditions) are tested.
- Path coverage: Every possible route through the code is checked. This helps find untested or "dead" code that could hide bugs.

3. Data Flow and Variables

Verifies that data is passed and manipulated correctly through the application. This includes ensuring variables are properly initialized, updated, and used without causing any errors or unexpected behavior. Monitoring the flow of data ensures that the system remains stable and reliable as it processes inputs, calculations, and outputs.

4. Internal Functions and Methods

Tests the individual functions or methods to ensure they perform their intended tasks accurately and return the expected results. This part of white-box testing focuses on validating business logic, mathematical computations, and other operations within the software. Ensuring these internal processes are correct helps catch potential issues early and improves the reliability of the overall system.

5. Boundary Conditions

Examines how the code handles edge cases, like the maximum or minimum values for inputs (e.g., a loop running 0 or 100 times, or an input field accepting a 255-character string). This ensures the app doesn't crash at its limits.

6. Error Handling and Exception Management

Confirms the program manages errors smoothly, catching exceptions (e.g., invalid inputs) and providing clear feedback. For example, testing if a file upload function handles a missing file gracefully.

Types of White Box Testing

White box testing can be done for different purposes at different places. There are three main types of White Box testing which is follows:-

Path Testing: White box testing will be checks all possible execution paths in the program to sure about the each one of the function behaves as expected. It helps verify that all logical conditions in the code are functioning correctly and efficiently with as properly manner, avoiding unnecessary steps with better code reusability.

Loop testing: It will be check that loops (for or while loops) in the program operate correctly and efficiently. It checks that the loop handles variables correctly and doesn't cause errors like infinite loops or logic flaws.

Unit Testing: Unit Testing checks if each part or function of the application works correctly. It will check the application meets design requirements during development.

Mutation Testing: It is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests. Mutation testing is related to modification a program in small ways.

Integration Testing: Integration Testing Examines how different parts of the application work together. After unit testing to make sure components work well both alone and together.

Penetration testing: Penetration testing, or pen testing, is like a practice cyber attack conducted on your computer systems to find and fix any weak spots before real attackers can exploit them. It focuses on web application security, where testers try to breach parts like APIs and servers to uncover vulnerabilities such as code injection risks from unfiltered inputs.

Process of White Box Testing

White box testing include the verify the internal workings of a software application. It checks that every aspect of the code is tested, basically is focusing on the logic, structure, and flow of the software.

Here's a breakdown of how this process works:

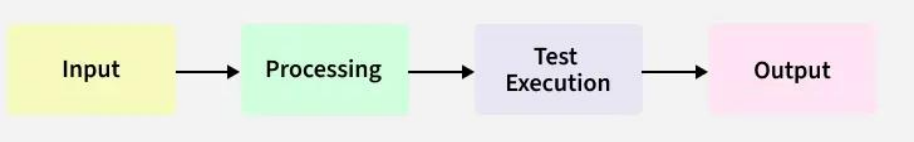


Figure 4.3

1. Input: Gathering Essential Documents

The process begins by collecting key documents to understand the application:

Requirements: These outline what the application is supposed to do and its expected behavior.

Functional Specifications: These describe how the software should perform under specific conditions.

Design Documents: These provide detailed insights into the architecture, components, and flow of the system.

Source Code: This is the actual code written for the application. It is where the logic and functionality are defined and is the primary focus during white box testing.

2. Processing: Planning and Prioritizing

With inputs gathered, testers prepare for thorough testing:

Risk Analysis: This step identifies potential risks in the code. By analyzing the application's functionality and dependencies, testers can identify areas where errors are more likely to occur and prioritize testing those areas. This helps in making the testing process more focused and efficient for the further process.

Test Planning: In this stage, testers design detailed test cases that cover all aspects of the code. The aim is to check all paths, conditions, loops, and functions within the code. Test planning ensures that no part of the application is left untested.

3. Test Execution: Running and Refining

Tests are executed to validate the code's behavior:

Execute the Tests: The test cases are run to check the behavior of the application. During execution, the application's internal logic is verified during the same. This includes testing individual functions, loops, and conditions to check that they work as expected.

Error Identification and Fixing: If errors or bugs are found, they are reported to the development team. The development team fixes the errors, and the tests are again run to verify the fixes. This cycle continues until the software is free from critical issues.

Results Communication: within the process of testing, the results are documented and communicated to all stakeholders to re-sure everyone is informed of the software's progress.

4. Output: Delivering Results

The process concludes with a comprehensive summary:

Final Report: A detailed report is prepared that includes all findings, test case results, error logs, and improvements made in the proper format which is easily understandable. This report documented as a record of the testing process and provides an complete overview of the software's quality. It is typically shared with the development team and other related stakeholders and members.

In white-box testing, the tester must to understand the application's code and write test cases to validate specific parts of it with checking all the function of the software. Then they can execute these tests, identify any issues, and check the software works correctly as expected.

White Box Testing Techniques

One of the main benefits of white box testing is that it allows for testing every part of an application. To achieve complete code coverage, white box testing uses the following techniques:

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

If we see in the case of a flowchart, every node must be traversed at least once. Since all lines of code are covered, it helps in pointing out faulty code while detecting.

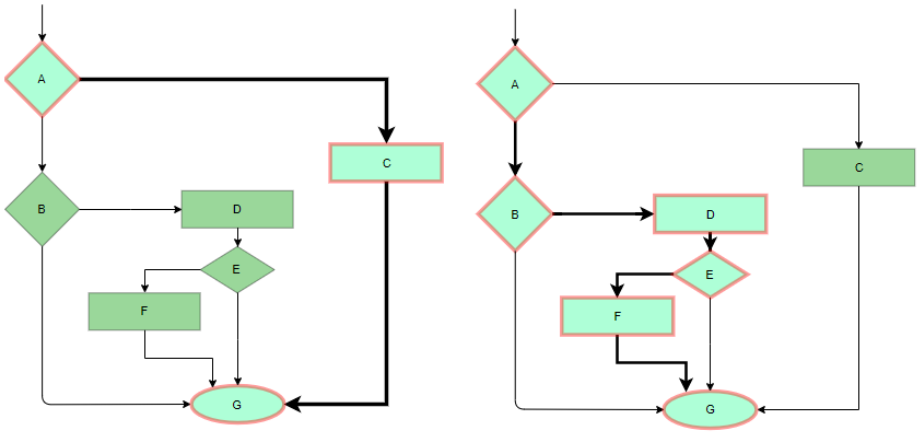


Figure 4.4

2. Branch Coverage: Branch coverage focuses on testing the decision points or conditional branches in the code. It checks whether both possible outcomes (true and false) of each conditional statement are tested. In this technique, test cases are designed so that each branch from all decision points is traversed at least once. In a flowchart, all edges must be traversed at least once.

In a flowchart, all edges must be traversed at least once.

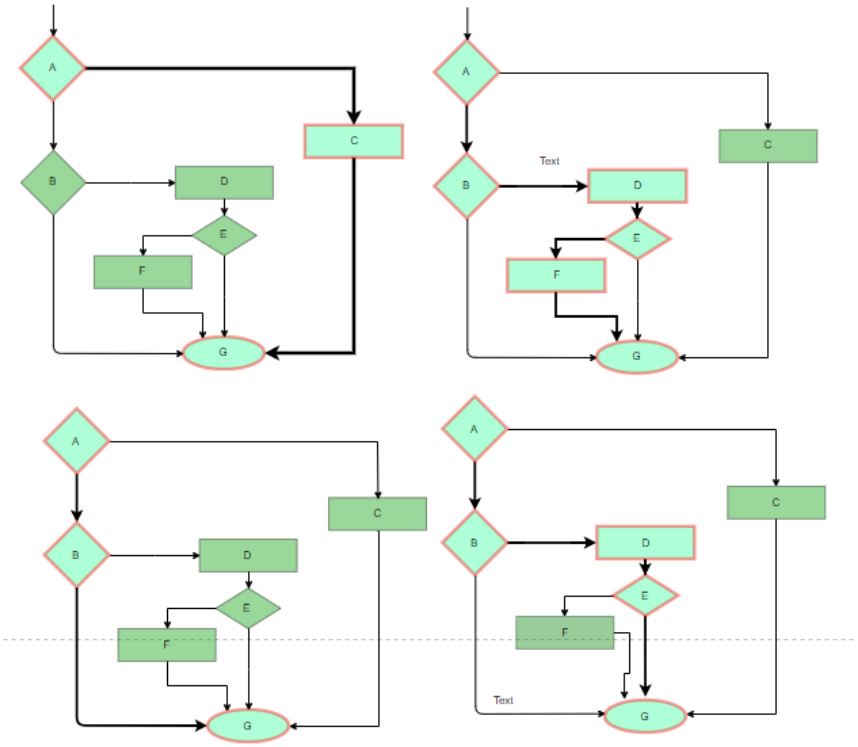


Figure 4.5

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

READ X, Y

IF(X == 0 || Y == 0)

PRINT '0'

#TC1 – X = 0, Y = 55

#TC2 – X = 5, Y = 0

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

READ X, Y

IF(X == 0 || Y == 0)

PRINT '0'

#TC1: X = 0, Y = 0

#TC2: X = 0, Y = 5

#TC3: X = 55, Y = 0

#TC4: X = 55, Y = 5

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

- Make the corresponding control flow graph
- Calculate the cyclomatic complexity
- Find the independent paths
- Design test cases corresponding to each independent path

V(G) = P + 1, where P is the number of predicate nodes in the flow graph

V(G) = E – N + 2, where E is the number of edges and N is the total number of nodes

V(G) = Number of non-overlapping regions in the graph

#P1: 1 – 2 – 4 – 7 – 8

#P2: 1 – 2 – 3 – 5 – 7 – 8

#P3: 1 – 2 – 3 – 6 – 7 – 8

#P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8

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

Simple loops: For simple loops of size n, test cases are designed that:

- Skip the loop entirely
- Only one pass through the loop
- 2 passes
- m passes, where m < n
- n-1 ans n+1 passes

Nested loops: For nested loops, all the loops are set to their minimum count, and we start from the innermost loop. Simple loop tests are conducted for the innermost loop and this is worked outwards till all the loops have been tested.

Concatenated loops: Independent loops, one after another. Simple loop tests are applied for each. If they're not independent, treat them like nesting.

Tools of White box testing

White box testing, also known as clear box or structural testing, involves examining the internal workings of an application to ensure its functionality and security. In 2025, several tools have used for this process. Here are few white box testing tools:

- SonarQube
- Veracode
- OWASP Code Pulse
- JaCoCo
- PVS-Studio
- Checkmarx
- Coverity
- Klocwork

- CodeClimate
- Codacy

These tools help developers to perform detailed analysis on the source code, checking that all potential issues are detected and addressed early in the development cycle.

Advantages of White Box Testing

Here are the Advantages of White Box Testing:

Thorough Testing: White box testing is thorough as the entire code and structures are tested.

Code Optimization: It results in the optimization of code removing errors and helps in removing extra lines of code.

Early Detection of Defects: It can start at an earlier stage as it doesn't require any interface as in the case of black box testing.

Integration with SDLC: White box testing can be easily started in Software Development Life Cycle.

Detection of Complex Defects: Testers can identify defects that cannot be detected through other testing techniques.

Comprehensive Test Cases: Testers can create more comprehensive and effective test cases that cover all code paths.

Disadvantages of White Box Testing

Here are the Disadvantages of White Box Testing:

Programming Knowledge and Source Code Access: Testers need to have programming knowledge and access to the source code to perform tests.

Overemphasis on Internal Workings: Testers may focus too much on the internal workings of the software and may miss external issues.

Bias in Testing: Testers may have a biased view of the software since they are familiar with its internal workings.

Test Case Overhead: Redesigning code and rewriting code needs test cases to be written again.

Dependency on Tester Expertise: Testers are required to have in-depth knowledge of the code and programming language as opposed to black-box testing.

Inability to Detect Missing Functionalities: Missing functionalities cannot be detected as the code that exists is tested.

Increased Production Errors: High chances of errors in production.

White box testing checks the internal code and logic of software to re-sure it to works properly and is optimized. It includes unit testing, integration testing, and regression testing, using methods like statement and branch coverage. While it helps catch defects early and improve performance, it requires knowledge of programming to understand issues related to the software's external behavior.

4.4 Path Testing

Path Testing is a method that is used to design the test cases. In the path testing method, the control flow graph of a program is designed to find a set of linearly independent paths of execution. In this method, Cyclomatic Complexity is used to determine the number of linearly independent paths and then test cases are generated for each path.

It gives complete branch coverage but achieves that without covering all possible paths of the control flow graph. McCabe's Cyclomatic Complexity is used in path testing. It is a structural testing method that uses the source code of a program to find every possible executable path.

Path Testing Process

Control Flow Graph:

Draw the corresponding control flow graph of the program in which all the executable paths are to be discovered.

Cyclomatic Complexity:

After the generation of the control flow graph, calculate the cyclomatic complexity of the program using the following formula.

```
McCabe's Cyclomatic Complexity = E - N + 2P

Where,
E = Number of edges in the control flow graph
N = Number of vertices in the control flow graph
P = Program factor
```

Make Set:

Make a set of all the paths according to the control flow graph and calculate cyclomatic complexity. The cardinality of the set is equal to the calculated cyclomatic complexity.

Create Test Cases:

Create a test case for each path of the set obtained in the above step.

Path Testing Techniques

Control Flow Graph: The program is converted into a control flow graph by representing the code into nodes and edges.

Decision to Decision path: The control flow graph can be broken into various Decision to Decision paths and then collapsed into individual nodes.

Independent paths: An Independent path is a path through a Decision to Decision path graph that cannot be reproduced from other paths by other methods.

Advantages of Path Testing

- The path testing method reduces the redundant tests.
- Path testing focuses on the logic of the programs.
- Path testing is used in test case design.

Disadvantages of Path Testing

- A tester needs to have a good understanding of programming knowledge or code knowledge to execute the tests.
- The test case increases when the code complexity is increased.
- It will be difficult to create a test path if the application has a high complexity of code.

- Some test paths may skip some of the conditions in the code. It may not cover some conditions or scenarios if there is an error in the specific paths.

4.5 Control structure testing

Control structure testing is used to increase the coverage area by testing various control structures present in the program. The different types of testing performed under control structure testing are as follows-

1. Condition Testing
2. Data Flow Testing
3. Loop Testing

1. Condition Testing: Condition testing is a test cased design method, which ensures that the logical condition and decision statements are free from errors. The errors present in logical conditions can be incorrect boolean operators, missing parenthesis in a booleans expression, error in relational operators, arithmetic expressions, and so on. The common types of logical conditions that are tested using condition testing are-

- A relation expression, like E1 op E2 where 'E1' and 'E2' are arithmetic expressions and 'OP' is an operator.
- A simple condition like any relational expression preceded by a NOT (~) operator. For example, (~E1) where 'E1' is an arithmetic expression and 'a' denotes NOT operator.
- A compound condition consists of two or more simple conditions, Boolean operator, and parenthesis. For example, (E1 & E2)|(E2 & E3) where E1, E2, E3 denote arithmetic expression and '&' and '|' denote AND or OR operators.
- A Boolean expression consists of operands and a Boolean operator like 'AND', OR, NOT. For example, 'A|B' is a Boolean expression where 'A' and 'B' denote operands and | denotes OR operator.

2. Data Flow Testing: The data flow test method chooses the test path of a program based on the locations of the definitions and uses all the variables in the program. The data flow test approach is depicted as follows suppose each statement in a program is assigned a unique statement number and that theme function cannot modify its parameters or global variables. For example, with S as its statement number.

```
DEF (S) = {X | Statement S has a definition of X}
USE (S) = {X | Statement S has a use of X}
```

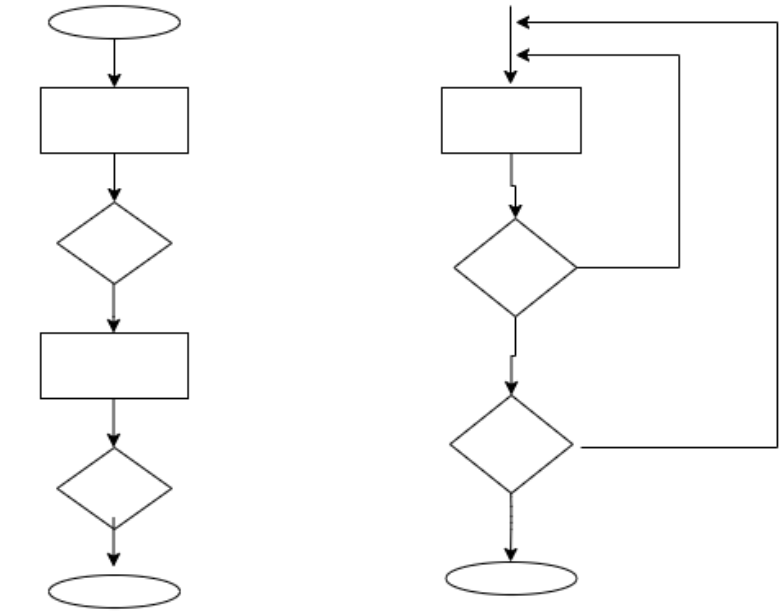
If statement S is an if loop statement, them its DEF set is empty and its USE set depends on the state of statement S. The definition of the variable X at statement S is called the line of statement S' if the statement is any way from S to statement S' then there is no other definition of X. A definition use (DU) chain of variable X has the form [X, S, S'], where S and S' denote statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is line at statement S'. A simple data flow test approach requires that each DU chain be covered at least once. This approach is known as the DU test approach. The DU testing does not ensure coverage of all branches of a program. However, a branch is not guaranteed to be covered by DU testing only in rare cases such as then in which the other construct does not have any certainty of any variable in its later part and the other part is not present. Data flow testing strategies are appropriate for choosing test paths of a program containing nested if and loop statements. 3. Loop Testing : Loop testing is actually a white box testing technique. It specifically focuses on the validity of loop construction. Following are the types of loops.

Simple Loop - The following set of test can be applied to simple loops, where the maximum allowable number through the loop is n.

- Skip the entire loop.
- Traverse the loop only once.
- Traverse the loop two times.
- Make p passes through the loop where p<n.

- Traverse the loop n-1, n, n+1 times.

Concatenated Loops - If loops are not dependent on each other, contact loops can be tested using the approach used in simple loops. if the loops are interdependent, the steps are followed in nested loops.



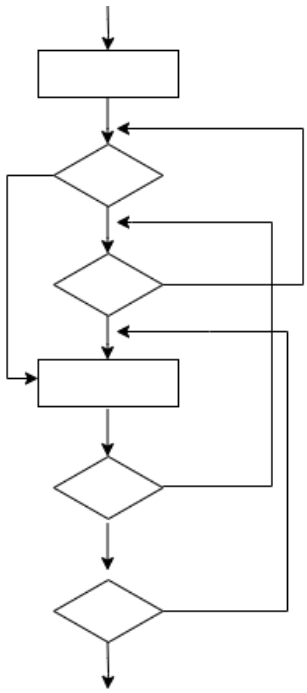
Concatenated Loops

Figure 4.6

Nested Loops - Loops within loops are called as nested loops. when testing nested loops, the number of tested increases as level nesting increases. The following steps for testing nested loops are as follows-

- Start with inner loop. set all other loops to minimum values.
- Conduct simple loop testing on inner loop.
- Work outwards.
- Continue until all loops tested.

Unstructured loops - This type of loops should be redesigned, whenever possible, to reflect the use of unstructured the structured programming constructs.



Unstructured Loops

Figure 4.7

4.6 Black box testing

Black-box testing is a Type of Software Testing in which the tester is not concerned with the software's internal knowledge or implementation details but rather focuses on validating the functionality based on the provided specifications or requirements.

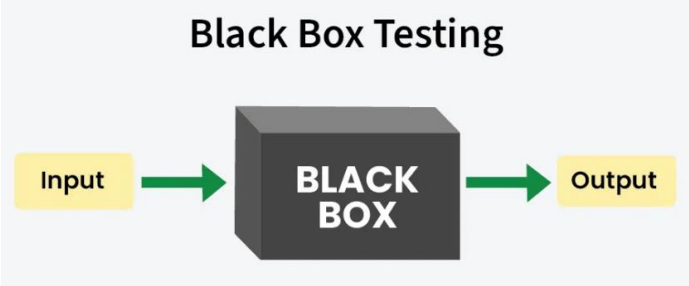


Figure 4.8

Types Of Black Box Testing

The testing of application without knowing the internal code or structure, following are the various Types of Black Box Testing:

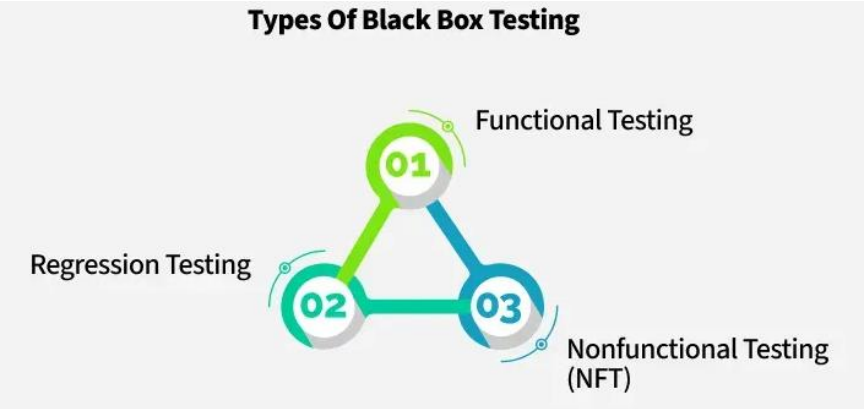


Figure 4.9

1. Functional Testing

Functional Testing is a type of Software Testing in which the system is tested against the functional requirements and specifications. Functional testing ensures that the requirements or specifications are properly satisfied by the application.

This testing is not concerned with the source code of the application. Each functionality of the software application is tested by providing appropriate test input, expecting the output, and comparing the actual output with the expected output.

This testing focuses on checking the user interface, APIs, database, security, client or server application, and functionality of the Application Under Test. Functional testing can be manual or automated. It determines the system's software functional requirements.

2. Regression Testing

Regression Testing is like a Software Quality checkup after any changes are made. It involves running tests to make sure that everything still works as it should, even after updates or tweaks to the code. This ensures that the software remains reliable and functions properly, maintaining its integrity throughout its development lifecycle.

Regression means the return of something and in the software field, it refers to the return of a bug. It ensures that the newly added code is compatible with the existing code.

In other words, a new software update has no impact on the functionality of the software. This is carried out after a system maintenance operation and upgrades.

3. Nonfunctional Testing

Non-functional Testing is a type of Software Testing that is performed to verify the non-functional requirements of the application. It verifies whether the behavior of the system is as per the requirement or not. It tests all the aspects that are not tested in functional testing.

- It is designed to test the readiness of a system as per nonfunctional parameters which are never addressed by functional testing.
- It is as important as functional testing.
- It is also known as NFT. This testing is not functional testing of software. It focuses on the software's performance, usability, and scalability.

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.

Ways of Black Box Testing Done

1. Syntax-Driven Testing

Syntax-Driven Testing is a Software Engineering technique or approach that is used in functional automation testing that's why called a type of functional automation testing.

- This type of testing is applied to systems that can be syntactically represented by some language.
- For example, language can be represented by context-free grammar.
- In this, the test cases are generated so that each grammar rule is used at least once.

2. Equivalence partitioning

Equivalence Partitioning Methods is also known as Equivalence Class partitioning (ECP). It is a software testing technique or black-box testing that divides input domain into classes of data, and with the help of these classes of data, test cases can be derived.

- The idea is to partition the input domain of the system into several Equivalence Classes such that each member of the class works similarly.
- If a test case in one class results in some error, other members of the class would also result in the same error.

The technique involves two steps:

Identification of equivalence class – Partition any input domain into a minimum of two sets: valid values and invalid values . For example, if the

valid range is 0 to 100 then select one valid input like 49 and one invalid like 104.

Generating test cases- To each valid and invalid class of input assign a unique identification number. Write a test case covering all valid and invalid test cases considering that no two invalid inputs mask each other. The whole number which is a perfect square-output will be an integer. The entire number which is not a perfect square-output will be a decimal number. Positive decimals Negative numbers(integer or decimal). Characters other than numbers like “a”,”!”,”;”, etc.

3. Boundary value analysis

Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.

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

4. Cause effect graphing

This technique establishes a relationship between logical input called causes with corresponding actions called the effect. The causes and effects are represented using Boolean graphs. The following steps are followed:

- Identify inputs (causes) and outputs (effect).
- Develop a cause-effect graph.
- Transform the graph into a decision table.
- Convert decision table rules to test cases.

For example, in the following cause-effect graph:

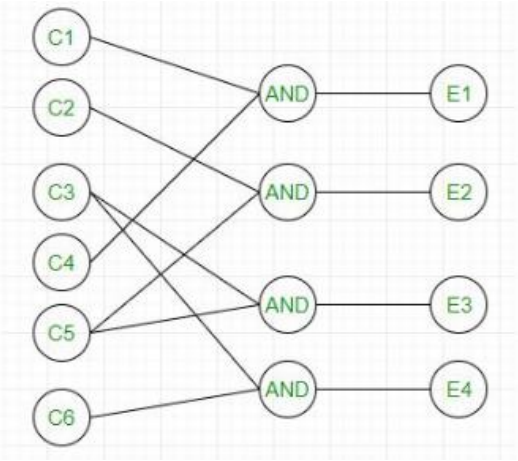


Figure 4.10

It can be converted into a decision table like:

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

Figure 4.11

Each column corresponds to a rule which will become a test case for testing. So there will be 4 test cases.

5. Requirement-based testing

Requirement-Based Testing in Software Development refers to a crucial process that involves validating a software system based on its specified criteria. This approach guarantees that the software aligns with documented specifications and meets the anticipated outcomes outlined during the initial phases of the project.

Principles of Requirement-Based Testing

Traceability: The approach aims to establish clear links between each test and its respective requirements, ensuring easy tracking.

Early Engagement: Early involvement in testing allows teams to comprehend, validate, and clarify requirements, minimizing misinterpretation risks.

Validation and Verification: This methodology focuses on both aspects to ensure software compliance with specified requirements, boosting testing reliability.

6. Compatibility testing

Compatibility Testing is the test case results not only depends on the product but is also on the infrastructure for delivering functionality. When the infrastructure parameters are changed it is still expected to work properly. Some parameters that generally affect the compatibility of software are:

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

Tools Used for Black Box Testing

Black box testing focuses on verifying the functionality of a software application by evaluating its inputs and outputs without any knowledge of its internal workings. The tester interacts with the system as an end-user to ensure the software meets its requirements and performs tasks as expected.

- QA Wolf
- Mobot
- Selendroid
- Watir
- Katalon
- IBM Rational Functional Tester (RFT)
- AutoHotkey
- Ranorex
- Selenium IDE
- TestComplete

Features of Black Box Testing

Independent testing: Black box testing is performed by testers who are not involved in the development of the application, which helps to ensure that testing is unbiased and impartial.

Testing from a user's perspective: Black box testing is conducted from the perspective of an end user, which helps to ensure that the application meets user requirements and is easy to use.

No knowledge of internal code: Testers performing black box testing do not have access to the application's internal code, which allows them to focus on testing the application's external behavior and functionality.

Requirements-based testing: Black box testing is typically based on the application's requirements, which helps to ensure that the application meets the required specifications.

Different testing techniques: Black box testing can be performed using various testing techniques, such as functional testing, usability testing, acceptance testing, and regression testing.

Easy to automate: Black box testing is easy to automate using various automation tools, which helps to reduce the overall testing time and effort.

Scalability: Black box testing can be scaled up or down depending on the size and complexity of the application being tested.

Limited knowledge of application: Testers performing black box testing have limited knowledge of the application being tested, which helps to ensure that testing is more representative of how the end users will interact with the application.

4.7 Unit testing

Unit testing is the process of testing the smallest parts of your code, like it is a method in which we verify the code's correctness by running one by one. It's a key part of software development that improves code quality by testing each unit in isolation.

You write unit tests for these code units and run them automatically every time you make changes. If a test fails, it helps you quickly find and fix the issue. Unit testing promotes modular code, ensures better test coverage, and saves time by allowing developers to focus more on coding than manual testing.

What is a Unit Test?

A unit test is a small piece of code that checks if a specific function or method in is an application works correctly. It will work as the function inputs and verifying the outputs. These tests check that the code work as expected based on the logic the developer intended.

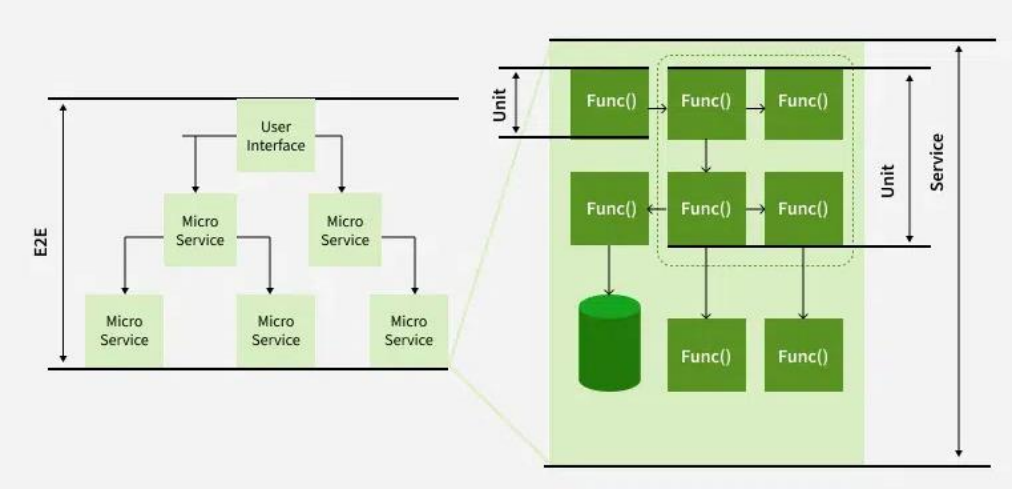


Figure 4.12

Unit testing strategies

To create effective unit tests, follow these basic techniques to ensure all scenarios are covered:

Logic checks: Verify if the system performs correct calculations and follows the expected path with valid inputs. Check all possible paths through the code are tested.

Boundary checks: Test how the system handles typical, edge case, and invalid inputs. For example, if an integer between 3 and 7 is expected, check how the system reacts to a 5 (normal), a 3 (edge case), and a 9 (invalid input).

Error handling: Check the system properly handles errors. Does it prompt for a new input, or does it crash when something goes wrong?

Object-oriented checks: If the code modifies objects, confirm that the object's state is correctly updated after running the code.

Benefits of Unit Testing

Here are the Unit testing benefits which used in the software development with many ways:

Early Detection of Issues: Unit testing allows developers to detect and fix issues early in the development process before they become larger and more difficult to fix.

Improved Code Quality: Unit testing helps to ensure that each unit of code works as intended and meets the requirements, improving the overall quality of the software.

Increased Confidence: Unit testing provides developers with confidence in their code, as they can validate that each unit of the software is functioning as expected.

Faster Development: Unit testing enables developers to work faster and more efficiently, as they can validate changes to the code without having to wait for the full system to be tested.

Better Documentation: Unit testing provides clear and concise documentation of the code and its behavior, making it easier for other developers to understand and maintain the software.

Facilitation of Refactoring: Unit testing enables developers to safely make changes to the code, as they can validate that their changes do not break existing functionality.

Reduced Time and Cost: Unit testing can reduce the time and cost required for later testing, as it helps to identify and fix issues early in the development process.

Disadvantages of Unit Testing

Here are the Unit testing dis-advantages which are follows:

Time and Effort: Unit testing requires a significant investment of time and effort to create and maintain the test cases, especially for complex systems.

Dependence on Developers: The success of unit testing depends on the developers, who must write clear, concise, and comprehensive test cases to validate the code.

Difficulty in Testing Complex Units: Unit testing can be challenging when dealing with complex units, as it can be difficult to isolate and test individual units in isolation from the rest of the system.

Difficulty in Testing Interactions: Unit testing may not be sufficient for testing interactions between units, as it only focuses on individual units.

Difficulty in Testing User Interfaces: Unit testing may not be suitable for testing user interfaces, as it typically focuses on the functionality of individual units.

Over-reliance on Automation: Over-reliance on automated unit tests can lead to a false sense of security, as automated tests may not uncover all possible issues or bugs.

Maintenance Overhead: Unit testing requires ongoing maintenance and updates, as the code and test cases must be kept up-to-date with changes to the software.

How do developers use unit tests?

Unit testing plays an important role throughout the software development process:

Test-Driven Development (TDD): In TDD, developers write tests before writing the actual code. This ensures that once the code is completed, it instantly meets the functional requirements when tested, saving time on debugging.

After Completing Code Blocks: After a section of code is finished, unit tests are created (if not already done through TDD). These tests are then run to verify that the code works as expected. Unit tests are rarely the first set of tests run during broader system testing.

DevOps and CI/CD: In DevOps environments, Continuous Integration/Continuous Delivery (CI/CD) automatically runs unit tests whenever new code is added. This ensures that changes are integrated smoothly, tested thoroughly, and deployed efficiently, maintaining overall code quality.

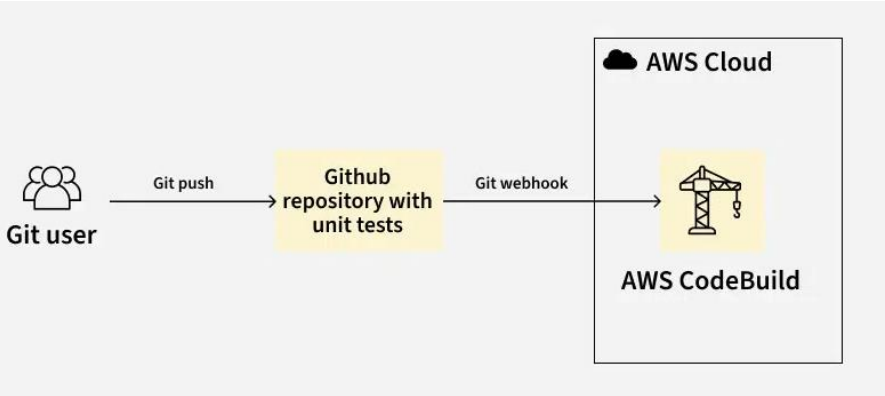


Figure 4.13

Types of Unit Testing

Unit testing can be performed manually or automatically:

1. Manual unit testing

Manual Testing is like checking each part of a project by hand, without using any special tools. People, like developers, do each step of the testing themselves. But manual unit testing isn't used much because there are better ways to do it and it has some problems:

- It costs more because workers have to be paid for the time they spend testing, especially if they're not permanent staff.
- It takes a lot of time because tests have to be done every time the code changes.
- It is hard to find and fix problems because it is tricky to test each part separately.
- Developers often do manual testing themselves to see if their code works correctly.

Figure 4.14

2. Automated unit testing

Automation Unit Testing is a way of checking if software works correctly without needing lots of human effort. We use special tools made by people to run these tests automatically. These are part of the process of building the software. Here's how it works:

- Developers write a small piece of code to test a function in the software. This code is like a little experiment to see if everything works as it should.
- Before the software is finished and sent out to users, these test codes are taken out. They're only there to make sure everything is working properly during development.
- Automated testing can help us check each part of the software on its own. This helps us find out if one part depends too much on another. It's like putting each piece of a puzzle under a magnifying glass to see if they fit together just right.
- We usually use special tools or frameworks to do this testing automatically. These tools can tell us if any of our code doesn't pass the tests we set up.
- The tests we write should be really small and focus on one specific thing at a time. They should also run on the computer's memory and not need internet connection.

So, automated unit testing is like having a helper that checks our work as we build software, making sure everything is in its right place and works as expected.

Unit Testing Tools

Choosing the correct unit testing tool is important for the following reasons:

- Ensures that every system component is benefitted from achieving better product quality.
- The debugging process is simplified.
- Code Refactoring and debugging of the code becomes simpler.
- If bugs are present, they would be defined earlier in the SDLC process.
- Bug fixing costs would be reduced significantly

Following are some of the unit testing tools:

LambdaTest: A cloud-based platform for cross-browser testing, supporting unit tests in various frameworks.

JUnit: A widely-used Java testing framework for creating and running unit tests.

NUnit: A .NET framework for unit testing C# applications.

TestNG: An advanced Java testing framework with features like parallel testing.

PHPUnit: A PHP testing framework for unit and integration tests.

Mockito: A Java mocking framework for simulating dependencies.

Cantata: A tool for unit and integration testing in C/C++.

Karma: A JavaScript test runner for browser-based testing.

Mocha: A flexible JavaScript testing framework for Node.js and browsers.

TypeMock: A .NET mocking framework for isolating dependencies.

Unit testing will validate individual units of software in a proper manner, checking if the function works correctly and meets the requirements of the project. While it may offer benefits such as early issue detection and improved code quality, it requires significant time and effort, which depends on the developers' skills.

Checking the challenges, such as the difficulty in testing complex units and UI elements, unit testing is crucial for ensuring software quality and the longevity of the software.

4.8 Integration testing

Integration Testing is a Software Testing Technique that focuses on verifying the interactions and data exchange between different components or modules of a Software Application. The goal of Integration Testing is to identify any problems or bugs that arise when different components are combined and interact with each other.

It mainly tests interface between two software units or modules. It focuses on determining the correctness of the interface. Once all the modules have been unit-tested, integration testing is performed.

Integration testing can be done by picking module by module. This can be done so that there is a proper sequence to be followed. Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units.

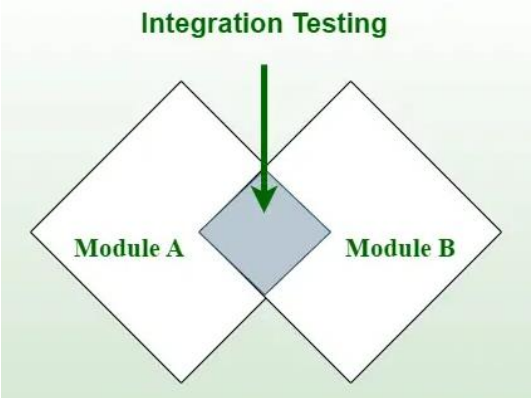


Figure 4.15

Why is Integration Testing Important?

Integration testing is important because it verifies that individual software modules or components work together correctly as a whole system. This ensures that the integrated software functions as intended and helps identify any compatibility or communication issues between different parts of the

system. By detecting and resolving integration problems early, integration testing contributes to the overall reliability, performance, and quality of the software product.

How to Write Integration Tests?

Designing integration test cases is a key part of ensuring that the different components of your software work well together. Here's a simplified approach to designing these tests:

Identify the components to be tested: Start by pinpointing which parts of your software need to be tested together. These are usually modules that interact or depend on each other.

Determine the test objectives: Define what you want to achieve with the test. Are you testing if data flows correctly between modules? Or perhaps checking if the system behaves as expected when components interact?

Define the test data: Decide what data you'll use to test the integration. Make sure the data represents real-world scenarios so that your tests are relevant and meaningful.

Design the test cases: Plan out the specific steps for each test. Think about what actions the test will take and what results you expect.

Develop test scripts: Write the code that will automate your tests. If your tests are manual, ensure the steps are clearly documented and easy to follow.

Set up the testing environment: Make sure the environment where the tests will run mimics the real-world setup as closely as possible. This will give you more accurate results.

Execute the tests: Run your tests, paying close attention to how the components interact and whether they perform as expected.

Evaluate the results: Finally, review the test outcomes. Did the components work as intended? Were there any errors or unexpected behaviors?

Types of Integration Testing

There are four main strategies for executing integration testing: big-bang, top-down, bottom-up, and sandwich (or hybrid) testing. Each of these methods

comes with its own set of advantages and challenges, so it's important to choose the right one based on the specific needs of your project. Those approaches are the following:

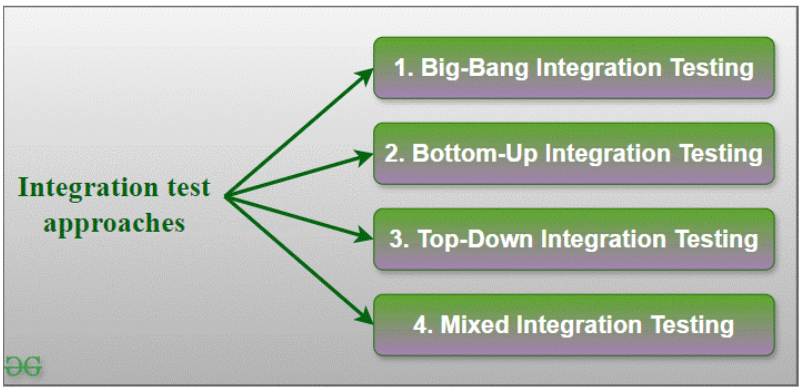


Figure 4.16

1. Big-Bang Integration Testing

It is the simplest integration testing approach, where all the modules are combined and the functionality is verified after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated.

- In debugging errors reported during Big Bang integration testing is very expensive to fix.
- Big-bang integration testing is a software testing approach in which all components or modules of a software application are combined and tested at once.
- This approach is typically used when the software components have a low degree of interdependence or when there are constraints in the development environment that prevent testing individual components.

- The goal of big-bang integration testing is to verify the overall functionality of the system and to identify any integration problems that arise when the components are combined.

While big-bang integration testing can be useful in some situations, it can also be a high-risk approach, as the complexity of the system and the number of interactions between components can make it difficult to identify and diagnose problems.

Advantages of Big-Bang Integration Testing

- It is convenient for small systems.
- Simple and straightforward approach.
- Can be completed quickly.
- Does not require a lot of planning or coordination.
- May be suitable for small systems or projects with a low degree of interdependence between components.

Disadvantages of Big-Bang Integration Testing

- There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
- High-risk critical modules are not isolated and tested on priority since all modules are tested at once.
- Not Good for long projects.
- High risk of integration problems that are difficult to identify and diagnose.
- This can result in long and complex debugging and troubleshooting efforts.
- This can lead to system downtime and increased development costs.
- May not provide enough visibility into the interactions and data exchange between components.
- This can result in a lack of confidence in the system's stability and reliability.

- This can lead to decreased efficiency and productivity.
- This may result in a lack of confidence in the development team.
- This can lead to system failure and decreased user satisfaction.

2. Bottom-Up Integration Testing

In bottom-up testing, each module at lower levels are tested with higher modules until all modules are tested. The primary purpose of this integration testing is that each subsystem tests the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules.

Advantages of Bottom-Up Integration Testing

- In bottom-up testing, no stubs are required.
- A principal advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- It is easy to create the test conditions.
- Best for applications that uses bottom up design approach.
- It is Easy to observe the test results.

Disadvantages of Bottom-Up Integration Testing

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystems.
- As Far modules have been created, there is no working model can be represented.

3. Top-Down Integration Testing

Top-down integration testing technique is used in order to simulate the behaviour of the lower-level modules that are not yet integrated. In this integration testing, testing takes place from top to bottom. First, high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

Advantages of Top-Down Integration Testing

- Separately debugged module.

- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.
- Easier isolation of interface errors.
- In this, design defects can be found in the early stages.

Disadvantages of Top-Down Integration Testing

- Needs many Stubs.
- Modules at lower level are tested inadequately.
- It is difficult to observe the test output.
- It is difficult to stub design.

4. Mixed Integration Testing

A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. It is also called the hybrid integration testing. also, stubs and drivers are used in mixed integration testing.

Advantages of Mixed Integration Testing

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.
- Parallel test can be performed in top and bottom layer tests.

Disadvantages of Mixed Integration Testing

- For mixed integration testing, it requires very high cost because one part has a Top-down approach while another part has a bottom-up approach.
- This integration testing cannot be used for smaller systems with huge interdependence between different modules.

Applications of Integration Testing

Integration testing is all about making sure that different parts of a software application work well together. While unit testing checks individual components, integration testing focuses on how those components interact with each other. Here are the few application of Integration Testing:

Identify the components: Identify the individual components of your application that need to be integrated. This could include the frontend, backend, database, and any third-party services.

Create a test plan: Develop a test plan that outlines the scenarios and test cases that need to be executed to validate the integration points between the different components. This could include testing data flow, communication protocols, and error handling.

Set up test environment: Set up a test environment that mirrors the production environment as closely as possible. This will help ensure that the results of your integration tests are accurate and reliable.

Execute the tests: Execute the tests outlined in your test plan, starting with the most critical and complex scenarios. Be sure to log any defects or issues that you encounter during testing.

Analyze the results: Analyze the results of your integration tests to identify any defects or issues that need to be addressed. This may involve working with developers to fix bugs or make changes to the application architecture.

Repeat testing: Once defects have been fixed, repeat the integration testing process to ensure that the changes have been successful and that the application still works as expected.

4.9 Regression testing

Regression Testing involves re-executing a previously created test suite to verify that recent code changes haven't caused new issues. This verifies that updates, bug fixes, or enhancements do not break the functionality of the application.

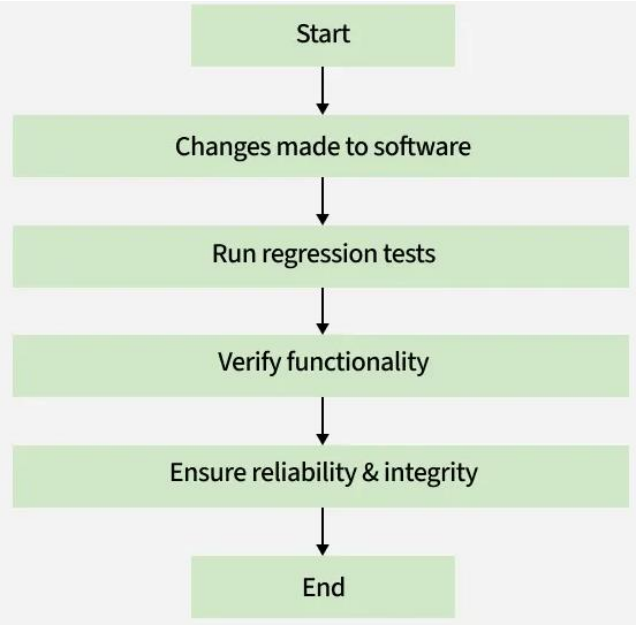


Figure 4.17

When to do Regression Testing?

Regression testing is necessary in several scenarios to maintain software quality:

- When new functionality is added to the system and the code has been modified to absorb and integrate that functionality with the existing code.
- When some defect has been identified in the software and the code is debugged to fix it.
- When the code is modified to optimize its working.

Process of Regression Testing

Here is the step-by-step process of the regression testing:

Identify Code Changes: Analyze the source code to determine which areas have been modified, such as new features, bug fixes, or optimizations.

Debug and Fix Failures: If existing test cases fail due to changes, debug the code to identify and resolve defects.

Modify Code: Apply necessary updates to the code to incorporate changes or fixes.

Select Test Cases: Choose relevant test cases from the existing test suite that cover modified and affected areas. Add new test cases if needed to address new functionality.

Execute Regression Tests: Run the selected test cases, either manually or using automated tools, to verify system behavior.

Analyze Results: Review test outcomes to identify regressions, document issues, and recommend fixes.

Retest as Needed: If defects are found, fix them and re-run tests to confirm resolution.

Techniques for Selecting Test Cases for Regression Testing

Selecting the right test cases is critical for efficient regression testing. Common techniques include:

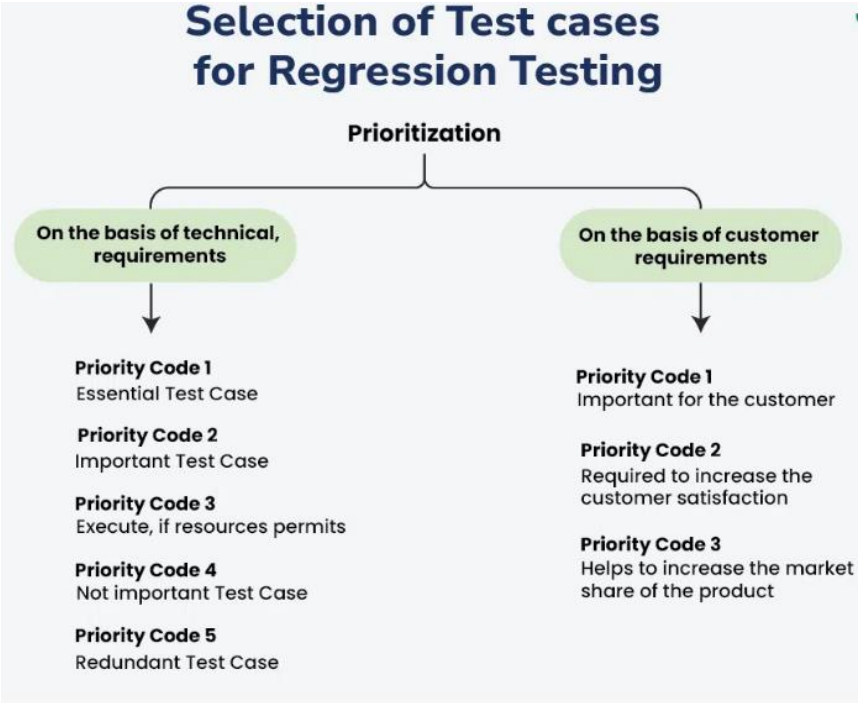


Figure 4.18

Select all test cases: In this technique, all the test cases are selected from the already existing test suite. It is the simplest and safest technique but not very efficient.

Select test cases randomly: In this technique, test cases are selected randomly from the existing test suite, but it is only useful if all the test cases are equally good in their fault detection capability which is very rare. Hence, it is not used in most of the cases.

Select modification traversing test cases: In this technique, only those test cases are selected that cover and test the modified portions of the source code and the parts that are affected by these modifications.

Select higher priority test cases: In this technique, priority codes are assigned to each test case of the test suite based upon their bug detection capability, customer requirements, etc. After assigning the priority codes, test cases with the highest priorities are selected for the process of regression testing. The test case with the highest priority has the highest rank. For example, a test case with priority code 2 is less important than a test case with priority code.

Regression Testing Tools

In regression testing, we generally select the Test Cases from the existing test suite itself and hence, we need not compute their expected output, and it can be easily automated due to this reason. Automating the process of regression testing will be very effective and time-saving.

The most commonly used tools for regression testing are:

Selenium: Open-source, supports multiple browsers and programming languages, and is widely used for automating web application tests.

Ranorex Studio: A comprehensive solution for testing web, desktop, and mobile applications with both codeless and coded automation options.

testRigor: AI-powered test automation tool that simplifies test creation with natural language processing and requires no coding skills.

Sahi Pro: A user-friendly tool that supports cross-browser testing and integrates well with continuous integration systems like Jenkins.

Testlio: A cloud-based solution with a global network of testers, ideal for on-demand testing, especially for regression testing in real-world scenarios.

Regression testing is like a safety net for software changes, making sure they do not bring in new problems. It picks test cases from existing suites, sometimes focusing on modified parts. Popular tools like Selenium and QTP automate this process. While it ensures thorough testing, doing it manually can up time and encounter data management issues.

4.10 Validation testing

Validation testing is the process of evaluating a software product to ensure that it meets the intended use and satisfies business requirements. This type of testing ensures that the final product meets the expectations of end users, confirming that the software operates as designed.

In the software development process, validation involves testing the product after it has been built, typically during the later stages, such as user acceptance testing (UAT) or system testing. It answers the critical question: "Are we building the right product?" This is different from verification testing, which focuses on ensuring that the software is built correctly according to design specifications.

Validation testing, along with verification and validation testing practices, is fundamental to the software engineering discipline. While verification testing ensures the software is built according to design specifications, validation testing ensures the product works correctly and fulfills the needs of the users. This distinction highlights the importance of conducting both forms of testing throughout the software development life cycle to ensure software quality.

Key Objectives of Validation Testing:

- Confirm that the software meets business requirements.
- Ensure the software fulfills user needs and expectations.
- Validate that the product operates under real-world conditions.

The Difference Between Verification and Validation

One common area of confusion is the distinction between verification and validation in software testing. Both processes are critical to the overall software development life cycle (SDLC), but they serve different purposes:

Verification checks if the product is built correctly by ensuring that the design specifications and coding meet required standards. It involves testing techniques like unit testing, integration testing, and functional testing. Verification ensures that the software functions as intended and adheres to the set guidelines.

Validation checks if the right product has been built, verifying whether the software meets user expectations and business requirements. This process typically involves higher-level testing, including system testing, user acceptance testing, and beta testing.

In simpler terms, verification testing ensures the software is correctly built, while validation testing ensures the software is fit for use.

Types of Validation Testing

There are various types of validation testing that software teams can perform throughout the development process to ensure the software meets user needs:

Functional Testing: Ensures the software's features work according to the specified requirements. It involves testing individual functionalities of the application to ensure the software behaves as expected.

Non-Functional Testing: Assesses aspects like performance, security, and usability to ensure the software can handle real-world conditions. Accessibility testing, for example, ensures the product is accessible to all users, including those with disabilities.

User Acceptance Testing (UAT): In this phase, real users test the software to verify that it meets their needs and expectations before release.

System Testing: Involves evaluating the software as a whole to ensure all components work together correctly. This includes end-to-end testing of the entire system.

Regression Testing: Ensures that new updates or changes to the software don't negatively impact existing functionality. Validation can help confirm that the software update doesn't introduce new bugs.

Beta Testing: Performed in a real-world environment by end users, beta testing helps identify any issues that may have been overlooked during earlier testing phases.

Validation Testing Techniques

Various testing techniques can be applied to validation testing in software development. These include:

Black Box Testing: Focuses on testing the software without looking at the underlying code. The tester inputs data and examines the outputs, verifying whether the software behaves as expected.

White Box Testing: Involves testing the internal structures of the application, with a focus on understanding the code and how it functions.

Test Automation: Tools can be employed to automate repetitive tasks in validation testing, ensuring consistency and efficiency. Automation testing is particularly useful for running large sets of test cases during regression testing.

Example of Validation Testing

Let's consider an e-commerce website that allows users to purchase products online. Validation testing would involve checking whether the checkout process works smoothly, whether payment methods are functioning, and whether the final product meets the user's needs. This might include UAT, where real customers simulate purchases to verify the system is user-friendly and performs as expected.

In the software development lifecycle, validation testing is essential for ensuring the final product meets user needs, business requirements, and design specifications. It is the process of evaluating a software system to verify that it operates as expected and delivers the desired results. Validation testing involves various testing phases such as system testing, user acceptance testing, and regression testing.

By incorporating validation testing techniques throughout the development process, you can ensure that the software is built correctly and performs well in real-world scenarios. This ensures a high-quality software application that meets the requirements of both developers and end users, driving confidence in the final product.

4.11 Security testing

Security Testing is a type of Software Testing that uncovers vulnerabilities in the system and determines that the data and resources of the system are protected from possible intruders. It ensures that the software system and application are free from any threats or risks that can cause a loss.

Security testing of any system is focused on finding all possible loopholes and weaknesses of the system that might result in the loss of information of the organization.

Types of Security Testing

Security testing is important to check and sure that applications and systems are protected from various threats. There are several types of security testing, each targeting specific vulnerabilities and aspects of security. Security testing includes various methods, each targeting specific vulnerabilities:

Figure 4.19

1. Vulnerability Scanning

It is a type of testing that uses automated tools to scan the system for known vulnerabilities and weaknesses. It aims to detect patterns of vulnerabilities that are commonly exploited by attackers. By performing vulnerability scans regularly, organizations can proactively address these vulnerabilities before they become security risks.

2. Security Scanning

It involves identifying weaknesses in the network or system and then providing solutions to mitigate these risks. It can be performed either manually or automatically, depending on the complexity of the system. This process helps uncover potential weak points that could be exploited by attackers, allowing for early intervention to secure the system.

3. Penetration Testing

Penetration Testing simulates an attack from a malicious hacker to identify vulnerabilities in the system. This type of testing helps organizations understand how an attacker might exploit weaknesses in the system. By performing penetration testing, organizations can see their system from an attacker's perspective and fix vulnerabilities before they are exploited in a real-world attack.

4. Risk Assessment

It involves analyzing the security risks that could affect the organization. Risks are categorized as low, medium, or high, and this testing suggests controls and measures to minimize those risks. Risk assessment helps prioritize actions by identifying the most critical threats and focusing efforts on addressing them first, ultimately improving the overall security posture of the system.

5. Security Auditing

These is an internal inspection of the system to identify security defects. This can involve reviewing system configurations, checking for weaknesses in the code, or conducting a line-by-line inspection of the application's source code.

Security audits ensure that all security standards and protocols are being followed and identify any gaps that need to be addressed.

6. Ethical Hacking

It also known as white-hat hacking, is when security professionals are hired to simulate attacks on the system to identify vulnerabilities. Unlike malicious hacking, ethical hacking is done with the organization's consent to help improve system security. Ethical hackers use the same techniques as malicious hackers to uncover weaknesses in the system, but their goal is to fix those flaws before they can be exploited by actual attackers.

7. Posture Assessment

These will combines security scanning, ethical hacking, and risk assessments to provide an overall view of the system's security. It gives a comprehensive evaluation of the system's security by integrating multiple testing methods, ensuring that no part of the security infrastructure is overlooked. This assessment helps organizations understand their security readiness and take necessary actions to strengthen their defenses.

8. Application Security Testing

These Testing focuses specifically on identifying vulnerabilities within the application itself. This includes examining the application's code, configurations, and dependencies to identify flaws that could lead to security breaches. Regular application security testing ensures that the software does not contain any weaknesses that could be exploited by attackers.

9. Network Security Testing

In these testing targets the vulnerabilities in the network infrastructure, such as firewalls, routers, and other network devices. This testing is crucial for identifying weaknesses that could allow unauthorized access to the system. Network security testing helps ensure that the communication pathways between devices are secure and that sensitive data is protected from cyber threats.

10. Social Engineering Testing

Social Engineering Testing simulates phishing, baiting, or other manipulative techniques used to exploit human behavior to gain unauthorized access. This type of testing focuses on the human element of security, ensuring that employees are aware of potential threats and know how to protect themselves from such attacks. By testing employees with simulated social engineering attacks, organizations can gauge the effectiveness of their security awareness programs and make necessary improvements.

In addition to manual methods, tools like Nessus, OpenVAS, and Metasploit can automate and simplify the process of security testing. These tools help speed up the identification of vulnerabilities and reduce the risk of human error, making the testing process more efficient.

Types of Security Testing Tools

Security Testing includes specialized tools enhance the efficiency and accuracy of security testing. Key tools include the following mentioned bellow:

SAST (Static Application Security Testing): Analyzes the source code to identify security flaws without executing the program. It helps developers identify and fix vulnerabilities early in the development process.

DAST (Dynamic Application Security Testing): Tests running applications to identify security vulnerabilities. It simulates real-world attacks like SQL injection or cross-site scripting (XSS) and is typically used for web applications.

IAST (Interactive Application Security Testing): Combines both static and dynamic testing to provide real-time feedback during the application's runtime. It offers deeper insights into the security of the application by continuously monitoring code flow.

SCA (Software Composition Analysis): Scans third-party libraries and dependencies used in the application for known vulnerabilities, license issues, and outdated components.

MAST (Mobile Application Security Testing): Focuses on identifying vulnerabilities in mobile applications, including platform-specific security risks, session handling, and insecure data storage.

RASP (Runtime Application Self-Protection): Embeds security controls within an application during runtime to detect and mitigate attacks in real-time. RASP tools protect applications by automatically responding to security threats as they occur.

Goal of Security Testing

The Security Testing Goals which are mentioned bellow:

- To identify the threats in the system and measure the potential vulnerabilities of the system.
- To help in detecting every possible security risk in the system and help developers fix security problems through coding.
- The goal of security testing is to identify vulnerabilities and potential threats in a system or application and to ensure that the system is protected against unauthorized access, data breaches, and other security-related issues. The main objectives of security testing are to:
- Identify vulnerabilities: Security testing helps identify vulnerabilities in the system, such as weak passwords, unpatched software, and misconfigured systems, that could be exploited by attackers.
- Evaluate the system's ability to withstand an attack: Security testing evaluates the system's ability to withstand different types of attacks, such as network attacks, social engineering attacks, and application-level attacks.
- Ensure compliance: Security testing helps ensure that the system meets relevant security standards and regulations, such as HIPAA, PCI DSS, and SOC2.
- Provide a comprehensive security assessment: Security testing provides a comprehensive assessment of the system's security posture, including the identification of vulnerabilities, the evaluation of the

system's ability to withstand an attack, and compliance with relevant security standards.

Help organizations prepare for potential security incidents: Security testing helps organizations understand the potential risks and vulnerabilities that they face, enabling them to prepare for and respond to potential security incidents.

Identify and fix potential security issues before deployment to production: Security testing helps identify and fix security issues before the system is deployed to production. This helps reduce the risk of a security incident occurring in a production environment.

Principle of Security Testing

Security testing follows seven core principles, often referred to as the CIA triad (Confidentiality, Integrity, Availability) with seven basic principles of security testing:

Confidentiality: verifies that sensitive data is only accessible to authorized users, often through encryption and access control mechanisms.

Integrity: Verifies that data remains unchanged and unaltered during storage or transmission. Hash functions and checksums are commonly used to guarantee integrity.

Authentication: Ensures that only authorized users can access the system. This involves testing password policies, multi-factor authentication (MFA), and identity verification mechanisms.

Authorization: Verifies that authenticated users can only access the resources and data they are authorized to use, through mechanisms such as role-based or attribute-based access control (RBAC and ABAC).

Availability: Ensures that the system remains functional and accessible, even under heavy traffic or during a cyberattack, such as a Distributed Denial of Service (DDoS) attack.

Non-Repudiation: Ensures that users cannot deny their actions in the system. Digital signatures, audit logs, and transaction records are commonly used to guarantee non-repudiation.

Resilience: Verifies the system's ability to recover from incidents, such as system crashes or attacks, by evaluating backup systems and response protocols.

Advantages of Security Testing

Security testing offers significant benefits that enhance system protection and user trust:

Identifying vulnerabilities: Security testing helps identify vulnerabilities in the system that could be exploited by attackers, such as weak passwords, unpatched software, and misconfigured systems.

Improving system security: Security testing helps improve the overall security of the system by identifying and fixing vulnerabilities and potential threats.

Ensuring compliance: Security testing helps ensure that the system meets relevant security standards and regulations, such as HIPAA, PCI DSS, and SOC2.

Reducing risk: By identifying and fixing vulnerabilities and potential threats before the system is deployed to production, security testing helps reduce the risk of a security incident occurring in a production environment.

Improving incident response: Security testing helps organizations understand the potential risks and vulnerabilities that they face, enabling them to prepare for and respond to potential security incidents.

Disadvantages of Security Testing

Here are the Security testing challenges which are mentioned bellow:

Resource-intensive: Security testing can be resource-intensive, requiring significant hardware and software resources to simulate different types of attacks.

Complexity: Security testing can be complex, requiring specialized knowledge and expertise to set up and execute effectively.

Limited testing scope: Security testing may not be able to identify all types of vulnerabilities and threats.

False positives and negatives: Security testing may produce false positives or false negatives, which can lead to confusion and wasted effort.

Time-consuming: Security testing can be time-consuming, especially if the system is large and complex.

Difficulty in simulating real-world attacks: It's difficult to simulate real-world attacks, and it's hard to predict how attackers will interact with the system.

With cyberattacks becoming more easily happens, it’s important to perform thorough security tests throughout the development process. This helps find vulnerabilities early on, preventing them from being exploited later.

By following best practices, using the right tools, and working closely with security experts, companies can create secure software that protects user data, meets regulatory standards, and builds trust with customers.

4.12 Testing Tools

Software Testing tools are the tools that are used for the testing of software. Software testing tools are often used to assure firmness, thoroughness, and performance in testing software products. Unit testing and subsequent integration testing can be performed by software testing tools. These tools are used to fulfill all the requirements of planned testing activities. These tools also work as commercial software testing tools. The quality of the software is evaluated by software testers with the help of various testing tools.

Types of Testing Tools

Software testing is of two types, static testing, and dynamic testing. Also, the tools used during these testing are named accordingly on these testings. Testing tools can be categorized into two types which are as follows:

1. Static Test Tools: Static test tools are used to work on the static testing processes. In the testing through these tools, the typical approach is taken. These tools do not test the real execution of the software. Certain input and output are not required in these tools. Static test tools consist of the following:

- Flow analyzers: Flow analyzers provides flexibility in the data flow from input to output.
- Path Tests: It finds the not used code and code with inconsistency in the software.
- Coverage Analyzers: All rationale paths in the software are assured by the coverage analyzers.
- Interface Analyzers: They check out the consequences of passing variables and data in the modules.

2. Dynamic Test Tools: Dynamic testing process is performed by the dynamic test tools. These tools test the software with existing or current data. Dynamic test tools comprise the following:

- Test driver: The test driver provides the input data to a module-under-test (MUT).
- Test Beds: It displays source code along with the program under execution at the same time.
- Emulators: Emulators provide the response facilities which are used to imitate parts of the system not yet developed.
- Mutation Analyzers: They are used for testing the fault tolerance of the system by knowingly providing the errors in the code of the software.

There is one more categorization of software testing tools. According to this classification, software testing tools are of 10 types:

Test Management Tools : Test management tools are used to store information on how testing is to be done, help to plan test activities, and report the status of quality assurance activities. For example, JIRA, Redmine, Selenium, etc.

Automated Testing Tools : Automated testing tools helps to conduct testing activities without human intervention with more accuracy and less time and effort. For example, Appium, Cucumber, Ranorex, etc.

Performance Testing Tools : Performance testing tools helps to perform effectively and efficiently performance testing which is a type of non-functional testing that checks the application for parameters like stability,

scalability, performance, speed, etc. For example, WebLOAD, Apache JMeter, Neo Load, etc.

Cross-browser Testing Tools : Cross-browser testing tools helps to perform cross-browser testing that lets the tester check whether the website works as intended when accessed through different browser-OS combinations. For example, Testsigma, Testim, Perfecto, etc.

Integration Testing Tools : Integration testing tools are used to test the interface between the modules and detect the bugs. The main purpose here is to check whether the specific modules are working as per the client's needs or not. For example, Citrus, FitNesse, TESSY, etc.

Unit Testing Tools : Unit testing tools are used to check the functionality of individual modules and to make sure that all independent modules works as expected. For example, Jenkins, PHPUnit, JUnit, etc.

Mobile Testing Tools : Mobile testing tools are used to test the application for compatibility on different mobile devices. For example, Appium, Robotium, Test IO, etc.

GUI Testing Tools : GUI testing tools are used to test the graphical user interface of the software. For example, EggPlant, Squish, AutoIT, etc.

Bug Tracking Tools : Bug tracking tool helps to keep track of various bugs that come up during the application lifecycle management. It helps to monitor and log all the bugs that are detected during software testing. For example, Trello, JIRA, GitHub, etc.

Security Testing Tools : Security testing is used to detect the vulnerabilities and safeguard the application against the malicious attacks. For example, NetSparker, Vega, ImmuniWeb, etc.

Top 10 Software Testing Tools

1. BrowserStack Test Management

BrowserStack Test management is the latest software test management platform that offers a centralized test case repository with the best-in-class

UI/UX. Integrates with other BrowserStack software testing tools such as Live, Test Observability, Automate & App Automate.

Features :

- Facilitates two-way integration with Jira, enhancing traceability for test cases and runs.
- Provides a rich dashboard for real-time reports & insights.
- Users can import data from existing tools using APIs or CSVs, with smart parsing for CSV fields.
- Test results can be uploaded from Test Observability or report formats like JUnit-XML/BDD-JSON.
- Supports test automation frameworks such as TestNG, WebdriverIO, Nightwatch.js, Appium, Playwright, etc.
- Integrates with CI/CD tools such as Jenkins, Azure Pipelines, Bamboo & CircleCI.

2. LambdaTest

LambdaTest is an AI-powered test orchestration and execution platform that allows developers and testers to perform manual and automated software testing at scale across different permutations of real browsers, devices, and operating systems.

Features:

- Run your test scripts on a cloud grid using popular test automation frameworks like Selenium, Playwright, Cypress, Appium, and more.
- Accelerate your software release cycles by multiple fold folds with parallel test execution.
- Test locally hosted projects with LambdaTest Tunnel and UnderPass before going live with your websites.
- Leverage the HyperExecute platform to perform end-to-end test orchestration and get high-test execution speed up to 70% faster than traditional cloud grids.

- Integrate LambdaTest with third-party tools like Jira, Asana, Jenkins, GitHub Actions, and more as per your project requirements.

3. TestGrid

TestGrid is a leading cloud-based end-to-end testing and test infrastructure platform designed to streamline and enhance the automated testing of web and mobile applications. The platform integrates seamlessly with leading test automation frameworks like Selenium, Appium, and Cypress, allowing for the automated execution of test scripts and enhancing testing efficiency and reliability.

Features:

- It supports integration with popular CI/CD tools such as Jenkins, CircleCI, and GitLab.
- Offers true scriptless testing for test case generation & execution
- It allows remote access to testers and developers for manual testing and debugging.
- It offers detailed reporting and analytics features for testing results.
- It enables cross-browser and cross-device testing.
- Both private and on-premise browser and mobile cloud infrastructure are available

4. QA Wolf

QA Wolf revolutionizes software testing by offering a powerful fusion of automated testing tools and professional QA services, enabling web application teams to attain 80% end-to-end test coverage at an unprecedented pace.

Features:

- Accelerates end-to-end user workflow test coverage to 80% in a matter of weeks
- Runs complete test suites in minutes via parallel execution.
- Completely eliminates flaky tests through proactive maintenance
- Charges are based on achieved test coverage, not hourly

- Frees up developers to focus on feature development
- Speeds up releases and minimizes production bugs
- Tests end-to-end web and native mobile applications including performance, accessibility, Salesforce, SAP, and more.

5. aqua cloud:

aqua cloud is an AI-powered test management solution designed to streamline and enhance the efficiency of QA teams. Offering 100% traceability, aqua helps you manage all testing activities with unparalleled ease and precision. The platform seamlessly integrates with leading test automation frameworks like Selenium, JMeter, Ranorex, and SoapUI, as well as any other tool via REST API, empowering teams without limiting their options.

Features:

- AI-Driven Test Management: You can use AI Copilot to create comprehensive requirements and entire test cases from a few tips or spoken instructions, significantly speeding up the documentation process.
- Automated Test Case Generation : aqua's AI helps you generate complete test cases from requirements in a few seconds, with 42% of generated tests requiring no tweaks.
- QA-Tuned Chat Bot: With a chatbot, you can interact with a custom algorithm for QA suggestions, test draft validations, and inspiration, making the test creation process up to 10 times faster.
- Enhanced Test Coverage Control: aqua allows you to easily update QA scope and identify gaps with aqua's requirements view.
- Efficient Bug Reporting: aqua's Capture extension records full desktop captures, window recordings, and in-app screencasts, providing developers with timestamped videos and transcripts for quick defect reproduction.
- Productivity-Boosting Dashboards and Custom Reports: You can also visualize testing and development data, track trends, set KPI alerts, and

create detailed reports with charts, external data, custom scripts, and more.

6. TestLodge

TestLodge is a lightweight test management tool designed to help QA teams plan, organize, and track their software testing efforts efficiently. It is aimed at teams who are executing manual test cases, rather than automated and, it integrates with popular bug-tracking tools like Jira, Trello, and GitHub, allowing teams to log defects seamlessly. With its easy-to-use interface and flexible test planning features, TestLodge simplifies test case management, ensuring a structured approach to software testing.

Features:

- It organizes test cases and execution cycles in a structured way to enhance team collaboration.
- It integrates with leading bug-tracking tools like Jira, Trello, and GitHub, ensuring a smooth defect management workflow.
- It improves test execution visibility by offering real-time progress tracking and detailed reporting.
- It enables unlimited user collaboration without extra cost, making it ideal for teams of all sizes.
- It provides flexible test planning with support for Agile, Waterfall, and hybrid testing methodologies.
- It generates detailed test reports and analytics to help teams track and optimize their testing efforts.

7. Selenium

Selenium provides a playback tool for authoring tests across most web browsers without the need to learn a test scripting language.

Features:

- It provides multi-browser support.
- It makes it easy to identify web elements on the web apps with the help of its several locators.

- It is able to execute test cases quicker than the other tools.

8. Ranorex

Ranorex Studio is a GUI test automation framework used for testing web-based, desktop, and mobile applications. It does not have its own scripting language to automate application.

Features:

- It helps to automate tests on Windows desktop, then execute locally or remotely on real or virtual machines.
- It runs tests in parallel to accelerate cross-browser testing for Chrome, Firefox, Safari, etc.
- It tests on real iOS or Android devices, simulators, emulators, etc.

9. TestProject:

TestProject is a test automation tool that allows users to create automated tests for mobile and web applications. It is built on top of popular frameworks like Selenium and Appium.

Features:

- It is a free end-to-end test automation platform for web, mobile, and API testing.
- Tests are saved as local files directly on your machine with no cloud-footprint to get a complete offline experience.
- It helps to create reliable codeless tests powered by self-healing, adaptive wait, and community add-ons.
- It provides insights about release quality, step-by-step detailed report with screenshots and logs.

10. Katalon Platform:

Katalon Platform is a comprehensive quality management platform that enables team to easily and efficiently test, launch, and optimize the best digital experiences.

Features:

- It is designed to create and reuse automated test scripts for UI without coding.
- It allows running automated tests of UI elements including pop-ups, iFrames, and wait-time.
- It eases deployment and allows wider set of integrations compared to Selenium.

Selecting the right software testing tools is crucial for ensuring high-quality software delivery. This blog post has explored various tools, from automation and performance testing to security and usability testing. By understanding the features and benefits of each tool, you can make informed decisions that align with your project requirements and goals. Implementing effective software testing tools enhances the efficiency of your testing processes, reduces bugs, and improves overall software quality.

4.13 Debugging

In the context of software engineering, debugging is the process of fixing a bug in the software. When there's a problem with software, programmers analyze the code to figure out why things aren't working correctly. They use different debugging tools to carefully go through the code, step by step, find the issue, and make the necessary corrections.

Figure 4.20

Process of Debugging

Debugging is a crucial skill in programming. Here's a simple, step-by-step explanation to help you understand and execute the debugging process effectively:

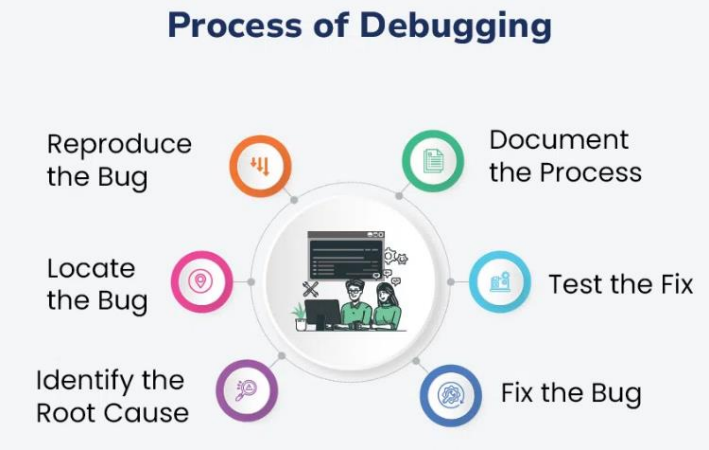


Figure 4.21

Step 1: Reproduce the Bug

- To start, you need to recreate the conditions that caused the bug. This means making the error happen again so you can see it firsthand.
- Seeing the bug in action helps you understand the problem better and gather important details for fixing it.

Step 2: Locate the Bug

- Next, find where the bug is in your code. This involves looking closely at your code and checking any error messages or logs.
- Developers often use debugging tools to help with this step.

Step 3: Identify the Root Cause

- Now, figure out why the bug happened. Examine the logic and flow of your code and see how different parts interact under the conditions that caused the bug.
- This helps you understand what went wrong.

Step 4: Fix the Bug

- Once you know the cause, fix the code. This involves making changes and then testing the program to ensure the bug is gone.
- Sometimes, you might need to try several times, as initial fixes might not work or could create new issues.
- Using a version control system helps track changes and undo any that don't solve the problem.

Step 5: Test the Fix

After fixing the bug, run tests to ensure everything works correctly. These tests include:

- Unit Tests: Check the specific part of the code that was changed.
- Integration Tests: Verify the entire module where the bug was found.
- System Tests: Test the whole system to ensure overall functionality.
- Regression Tests: Make sure the fix didn’t cause any new problems elsewhere in the application.

Step 6: Document the Process

- Finally, record what you did. Write down what caused the bug, how you fixed it, and any other important details.
- This documentation is helpful if similar issues occur in the future.

Why is debugging important?

Fixing mistakes in computer programming, known as bugs or errors, is necessary because programming deals with abstract ideas and concepts. Computers understand machine language, but we use programming languages to make it easier for people to talk to computers. Software has many layers of abstraction, meaning different parts must work together for an application to function properly. When errors happen, finding and fixing them can be tricky. That's where debugging tools and strategies come in handy. They help solve problems faster, making developers more efficient. This not only improves the quality of the software but also makes the experience better for the people

using it. In simple terms, debugging is important because it makes sure the software works well and people have a good time using it.

Debugging Approaches/Strategies

Brute Force: Study the system for a longer duration to understand the system. It helps the debugger to construct different representations of systems to be debugged depending on the need. A study of the system is also done actively to find recent changes made to the software.

Backtracking: Backward analysis of the problem which involves tracing the program backward from the location of the failure message to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.

Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused on to find the defect.

Using A debugging experience with the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

Cause elimination: it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

Static analysis: Analyzing the code without executing it to identify potential bugs or errors. This approach involves analyzing code syntax, data flow, and control flow.

Dynamic analysis: Executing the code and analyzing its behavior at runtime to identify errors or bugs. This approach involves techniques like runtime debugging and profiling.

Collaborative debugging: Involves multiple developers working together to debug a system. This approach is helpful in situations where multiple modules or components are involved, and the root cause of the error is not clear.

Logging and Tracing: Using logging and tracing tools to identify the sequence of events leading up to the error. This approach involves collecting and analyzing logs and traces generated by the system during its execution.

Automated Debugging: The use of automated tools and techniques to assist in the debugging process. These tools can include static and dynamic analysis tools, as well as tools that use machine learning and artificial intelligence to identify errors and suggest fixes.

Debugging Tools

Debugging tools are essential for software development, helping developers locate and fix coding errors efficiently. With the rapid growth of software applications, the demand for advanced debugging tools has increased significantly. Companies are investing heavily in these tools, and researchers are developing innovative solutions to enhance debugging capabilities, including AI-driven debuggers and autonomous debugging for specialized applications.

Debugging tools vary in their functionalities, but they generally provide command-line interfaces to help developers identify and resolve issues. Many also offer remote debugging features and tutorials, making them accessible to beginners. Here are some of the most commonly used debugging tools:

1. Integrated Development Environments (IDEs)

IDEs like Visual Studio, Eclipse, and PyCharm offer features for software development, including built-in debugging tools. These tools allow developers to:

- Execute code line-by-line (step debugging)
- Stop program execution at specific points (breakpoints)
- Examine the state of variables and memory
- IDEs support many programming languages and scripting languages, often through open-source plugins.

2. Standalone Debuggers

Standalone debuggers like GDB (GNU Debugger) provide advanced debugging features:

- Conditional breakpoints and watchpoints
- Reverse debugging (running a program backwards)
- These tools are powerful but have a steeper learning curve compared to IDE debuggers.

3. Logging Utilities

Logging utilities log a program's state at various points in the code, which can then be analyzed to find problems. Logging is particularly useful for debugging issues that only occur in production environments.

4. Static Code Analyzers

Static code analysis tools examine code without executing it to find potential errors and deviations from coding standards. They focus on the semantics of the source code, helping developers catch common mistakes and maintain consistent coding styles.

5. Dynamic Analysis Tools

Dynamic analysis tools monitor software as it runs to detect issues like resource leaks or concurrency problems. These tools help catch bugs that static analysis might miss, such as memory leaks or buffer overflows.

6. Performance Profilers

Performance profilers help developers identify performance bottlenecks in their code. They measure:

- CPU usage
- Memory usage
- I/O operations

4.14 Software Implementation

The software implementation phase is part of the continuous improvement cycle where deliverables designed are adopted and made operational by an

organization or end-users. It is an executing phase of the system development lifecycle that includes the steps of concept to function by coding, testing, and deployment. Implementation encompasses a range of steps that include installation, configuration, customization, and integration to make sure it runs as expected and fulfills the expected objectives and requirements.

Why is software implementation important?

Access to the Latest Technology: Software implementation is a method that allows a company to meet the latest tech requirements, i.e. replacing old-fashioned apps with new-generation ones, that have more functionalities and better capabilities.

Effective Process for Implementation: It is a fact that implementing a clearly defined process for software implementation increases our chances of delivering the anticipated outcomes. This selection is made, provided that the applications are within budget and also have a compatible system baseline.

Minimized Downtime: By sanely and correctly picking up and executing programs, you drastically reduce software failure and downtime. Therefore, the operations of the organization are not affected.

Efficiency and Productivity: Besides accurate installation of software, it can also eliminate processes, implement crush tasks, and raise productivity. Automation can be the best way to reduce human errors, reduce the general redundant tasks, and improve the entire productivity of a company.

Competitive Advantage: Successfully executed software adaptations can help businesses get better of the competition by speeding up the reaction, improving customer service, and increasing the innovation of the company.

Adaptation to Change: The implementation of software organizations usually faces change management, which implies a process that provides companies with different kinds of techniques, technologies, procedures, and business principles. It facilitates a climate that is based on constant process improvement and adaptability.

Maximizing ROI: A well-executed software implementation can yield significant financial returns. This could be through cost savings, increased revenue, or both. An effective implementation ensures the software's features are fully utilized, thereby maximizing the return on investment.

Elements of a Successful Software Implementation

There are elements of a successful software implementation which is follow:

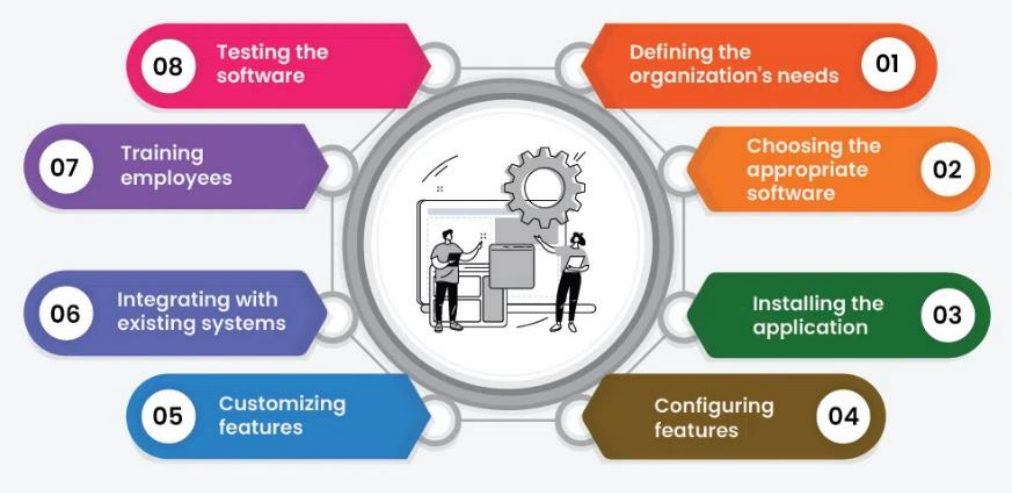


Figure 4.22

1. Defining the organization's needs

- Here you determine the requirements necessary to be satisfied so that you can choose from among many available applications.
- First of all, enumerate the purposes of that software.
- This can be represented, for example, in the case of a company that is engaged in writing and editing web copy, which would significantly benefit from the use of a content management system.
- An integrated HRMS solution for an organization could be more likely to bring order to its payrolls and benefits administration.
- Fortify the exact features of the software after identifying the software functioning for the business. Also, consider the additional needs of the organization.

2. Choosing the appropriate software

- When you have established the organization's need for software, you can move on to begin searching for the right software.
- Searching for the bestselling items belonging to your industry and what your rivals are employing might also assist you.
- Maybe, it is best to choose the updated products that offer new attributes that your company needs.
- When opting for the right application, you can search for the reviews written by users with whom you can discuss with the vendors to make sure the product can handle your team size.
- If they are someone else, you could ask the vendors if they have a pilot program that will allow your team to test the software before buying it.

3. Installing the application

- Vendors provide whether or not well enough support to make the installation process.
- Installing landscaping services is included for others in the software package, but some others provide the service for extra money.
- If you ask the possible vendor to install that new application, try to get your IT department people work alongside the vendor.
- Working together on this will guarantee all the devices get the application installed and a smooth integration with the current system and infrastructure of the community centre.
- If you let your IT department individually install the new application for you, offer the vendor's user's manual or contacts about this issue as a workaround.

4. Configuring features

- During the installation process, have your IT department simply set up the minimum features at this step at the beginning.
- Setting up simple features by default is what configurations of programs are usually limited to.

- For example, the template customer form can be taken from the program and you will then add more fields to the form.
- This is a sequential development process that allows future users and customers to have the application ready to be used virtually immediately.
- Through less time spent repairing things, you certainly gain a productivity boost and enhance customer satisfaction.
- Additionally, this ease can be misleading; troubleshooting might be involved when complex features are added.

5. Customizing features

- The application's hard-wired options can be useful in achieving your goals, but there could be times when you may be interested in customizing them to gain more flexibility for employees and customers.
- Say for example, assuming the form template has an address option or field, you can add to the form more purposeful information regarding your customers like address or home information among others.
- This other possibility is the integration of the dashboard which should be the main signpost in regards to the company's performance.
- Productivity metrics can enable the employees to measure their achievements and hence, keep striving to attain their purposes.

6. Integrating with existing systems

- The choice of software is elicited during the selection process, and it is imperative to choose an app that can integrate with the IT environment of the organization.
- Compatibility allows multiple features to work together as one, and no errors are raised to a higher authority.
- As your team is transferring the new app into the system, they should take care of how to duplicate the data to a function that you don’t use any more.

- Automatic migration of data can assist employees in setting aside an adequate amount of time to reduce the occurrences of information leaks, including customer payment details.

7. Training employees

- A good training program must be organized for employees to be aware of how to use the application effectively.
- The presentations could focus on what the new software can do which the old system could not, how the features can be maximized, and so much more.
- Hence, the training program should have login information for the account provided for the employees not forgetting to set the required permissions.

8. Testing the software

- Employees usually send IT department bugs they found while using new software, but the first sign of such a problem is testing.
- App testing functions as a pro to measure the results of every feature and detect the mistakes that concern the big group or the unlikely bugs.
- Your surveys can be processed with the utilization or not the program with the experiences of the employees and customers.
- When you get every critical matter right, you can start making the required amendments and carry out more testing.
- Another important advantage of drawing up a list of expected outcomes is being able to locate areas that are the vendor's responsibility to resolve.

In conclusion, software implementation efficiency is a paramount criterion for maximizing the potential of having the latest technologies that enhance productivity whatsoever, as well as enhance customer satisfaction levels. It implies a structured process of assessing the company's needs, choosing software that is likely to perform those tasks, and handling the installation, configuration, customization and integration of software

packages. The precise training and rigorous testing that brings down downtime and ensures a smooth implementation is hence required. Implementing with proper steps allows companies to become more efficient, provide an edge over the competition, and execute change concurrently with business operations and technology.

4.15 Coding Practices and Principles

Good software development organizations want their programmers to maintain to some well-defined and standard style of coding called coding standards. They usually make their own coding standards and guidelines depending on what suits their organization best and based on the types of software they develop. It is very important for the programmers to maintain the coding standards otherwise the code will be rejected during code review.

Purpose of Having Coding Standards

The following are the purpose of having Coding Standards:

- A coding standard gives a uniform appearance to the codes written by different engineers.
- It improves readability, and maintainability of the code and it reduces complexity also.
- It helps in code reuse and helps to detect errors easily.
- It promotes sound programming practices and increases the efficiency of the programmers.

Coding Standards in Software Engineering

Some of the coding standards are given below:

Limited use of globals: These rules tell about which types of data that can be declared global and the data that can't be.

Standard headers for different modules: For better understanding and maintenance of the code, the header of different modules should follow some standard format and information. The header format must contain below things that is being used in various companies:

- Name of the module
- Date of module creation
- Author of the module
- Modification history
- Synopsis of the module about what the module does
- Different functions supported in the module along with their input output parameters
- Global variables accessed or modified by the module

Naming conventions for local variables, global variables, constants and functions: Some of the naming conventions are given below:

- Meaningful and understandable variables name helps anyone to understand the reason of using it.
- Local variables should be named using camel case lettering starting with small letter (e.g. localData) whereas Global variables names should start with a capital letter (e.g. GlobalData). Constant names should be formed using capital letters only (e.g. CONSDATA).
- It is better to avoid the use of digits in variable names.
- The names of the function should be written in camel case starting with small letters.
- The name of the function must describe the reason of using the function clearly and briefly.

Indentation: Proper indentation is very important to increase the readability of the code. For making the code readable, programmers should use White spaces properly. Some of the spacing conventions are given below:

- There must be a space after giving a comma between two function arguments.
- Each nested block should be properly indented and spaced.
- Proper Indentation should be there at the beginning and at the end of each block in the program.

- All braces should start from a new line and the code following the end of braces also start from a new line.

Error return values and exception handling conventions: All functions that encountering an error condition should either return a 0 or 1 for simplifying the debugging.

Coding Guidelines in Software Engineering

Coding guidelines give some general suggestions regarding the coding style that to be followed for the betterment of understandability and readability of the code.

Some of the coding guidelines are given below :

Avoid using a coding style that is too difficult to understand: Code should be easily understandable. The complex code makes maintenance and debugging difficult and expensive.

Avoid using an identifier for multiple purposes: Each variable should be given a descriptive and meaningful name indicating the reason behind using it. This is not possible if an identifier is used for multiple purposes and thus it can lead to confusion to the reader. Moreover, it leads to more difficulty during future enhancements.

Code should be well documented: The code should be properly commented for understanding easily. Comments regarding the statements increase the understandability of the code.

Length of functions should not be very large: Lengthy functions are very difficult to understand. That's why functions should be small enough to carry out small work and lengthy functions should be broken into small ones for completing small tasks.

Try not to use GOTO statement: GOTO statement makes the program unstructured, thus it reduces the understandability of the program and also debugging becomes difficult.

Advantages of Coding Guidelines

- Coding guidelines increase the efficiency of the software and reduces the development time.
- Coding guidelines help in detecting errors in the early phases, so it helps to reduce the extra cost incurred by the software project.
- If coding guidelines are maintained properly, then the software code increases readability and understandability thus it reduces the complexity of the code.
- It reduces the hidden cost for developing the software.

Coding standards and guidelines ensure consistent, readable, and maintainable code, promoting efficient development and error detection. They standardize naming, indentation, and documentation practices, reducing complexity and facilitating code reuse. Adhering to these practices enhances overall software quality and development efficiency.

4.16 Maintenance and its Types

Software maintenance is a continuous process that occurs throughout the entire life cycle of the software system.

- The goal of software maintenance is to keep the software system working correctly, efficiently, and securely, and to ensure that it continues to meet the needs of the users.
- This can include fixing bugs, adding new features, improving performance, or updating the software to work with new hardware or software systems.
- It is also important to consider the cost and effort required for software maintenance when planning and developing a software system.
- It is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders.

- It's important to note that software maintenance can be costly and complex, especially for large and complex systems. Therefore, the cost and effort of maintenance should be taken into account during the planning and development phases of a software project.
- It's also important to have a clear and well-defined maintenance plan that includes regular maintenance activities, such as testing, backup, and bug fixing.

Several Key Aspects of Software Maintenance

Bug Fixing: The process of finding and fixing errors and problems in the software.

Enhancements: The process of adding new features or improving existing features to meet the evolving needs of the users.

Performance Optimization: The process of improving the speed, efficiency, and reliability of the software.

Porting and Migration: The process of adapting the software to run on new hardware or software platforms.

Re-Engineering: The process of improving the design and architecture of the software to make it more maintainable and scalable.

Documentation: The process of creating, updating, and maintaining the documentation for the software, including user manuals, technical specifications, and design documents.

Several Types of Software Maintenance

Corrective Maintenance: This involves fixing errors and bugs in the software system.

Patching: It is an emergency fix implemented mainly due to pressure from management. Patching is done for corrective maintenance but it gives rise to unforeseen future errors due to lack of proper impact analysis.

Adaptive Maintenance: This involves modifying the software system to adapt it to changes in the environment, such as changes in hardware or software, government policies, and business rules.

Perfective Maintenance: This involves improving functionality, performance, and reliability, and restructuring the software system to improve changeability. Preventive Maintenance: This involves taking measures to prevent future problems, such as optimization, updating documentation, reviewing and testing the system, and implementing preventive measures such as backups.

Maintenance can be categorized into proactive and reactive types. Proactive maintenance involves taking preventive measures to avoid problems from occurring, while reactive maintenance involves addressing problems that have already occurred.

Maintenance can be performed by different stakeholders, including the original development team, an in-house maintenance team, or a third-party maintenance provider. Maintenance activities can be planned or unplanned. Planned activities include regular maintenance tasks that are scheduled in advance, such as updates and backups. Unplanned activities are reactive and are triggered by unexpected events, such as system crashes or security breaches. Software maintenance can involve modifying the software code, as well as its documentation, user manuals, and training materials. This ensures that the software is up-to-date and continues to meet the needs of its users.

Software maintenance can also involve upgrading the software to a new version or platform. This can be necessary to keep up with changes in technology and to ensure that the software remains compatible with other systems. The success of software maintenance depends on effective communication with stakeholders, including users, developers, and management. Regular updates and reports can help to keep stakeholders informed and involved in the maintenance process.

Software maintenance is also an important part of the Software Development Life Cycle (SDLC). To update the software application and do all modifications in software application so as to improve performance is the main focus of software maintenance. Software is a model that runs on the

basis of the real world. so, whenever any change requires in the software that means the need for real-world changes wherever possible.

Need for Maintenance

Software Maintenance must be performed in order to:

- Correct faults.
- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.
- Migrate legacy software.
- Retire software.
- Requirement of user changes.
- Run the code fast

Challenges in Software Maintenance

The various challenges in software maintenance are given below:

- The popular age of any software program is taken into consideration up to ten to fifteen years. As software program renovation is open-ended and might maintain for decades making it very expensive.
- Older software programs, which had been intended to paint on sluggish machines with much less reminiscence and garage ability cannot maintain themselves tough in opposition to newly coming more advantageous software programs on contemporary-day hardware.
- Changes are frequently left undocumented which can also additionally reason greater conflicts in the future.
- As the era advances, it turns into high prices to preserve vintage software programs.

- Often adjustments made can without problems harm the authentic shape of the software program, making it difficult for any next adjustments.
- There is a lack of Code Comments.

Lack of documentation: Poorly documented systems can make it difficult to understand how the system works, making it difficult to identify and fix problems.

Legacy code: Maintaining older systems with outdated technologies can be difficult, as it may require specialized knowledge and skills.

Complexity: Large and complex systems can be difficult to understand and modify, making it difficult to identify and fix problems.

Changing requirements: As user requirements change over time, the software system may need to be modified to meet these new requirements, which can be difficult and time-consuming.

Interoperability issues: Systems that need to work with other systems or software can be difficult to maintain, as changes to one system can affect the other systems.

Lack of test coverage: Systems that have not been thoroughly tested can be difficult to maintain as it can be hard to identify and fix problems without knowing how the system behaves in different scenarios.

Lack of personnel: A lack of personnel with the necessary skills and knowledge to maintain the system can make it difficult to keep the system up-to-date and running smoothly.

High-Cost: The cost of maintenance can be high, especially for large and complex systems, which can be difficult to budget for and manage.

To overcome these challenges, it is important to have a well-defined maintenance process in place, which includes testing and validation, version control, and communication with stakeholders. It is also important to have a clear and well-defined maintenance plan that includes regular maintenance activities, such as testing, backup, and bug fixing. Additionally, it is important

to have personnel with the necessary skills and knowledge to maintain the system.

Categories of Software Maintenance

Maintenance can be divided into the following categories.

Corrective maintenance: Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.

Adaptive maintenance: This includes modifications and updations when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.

Perfective maintenance: A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer's demands.

Preventive maintenance: This type of maintenance includes modifications and updations to prevent future problems with the software. It goals to attend to problems, which are not significant at this moment but may cause serious issues in the future.

UNIT V

AGILE DEVELOPMENT AND DEV/OPS

5.1 Agile Development

Agile Software Development is a Software Development Methodology that values flexibility, collaboration, and customer satisfaction. It is based on the Agile Manifesto, a set of principles for software development that prioritize individuals and interactions, working software, customer collaboration, and responding to change.

Agile Software Development is an iterative and incremental approach to Software Development that emphasizes the importance of delivering a working product quickly and frequently. It involves close collaboration between the development team and the customer to ensure that the product meets their needs and expectations.

Agile is used because it helps teams deliver value quickly and continuously. By prioritizing the delivery of difficult results early in the project, customers benefit from seeing and using the product sooner, allowing for quick feedback and adjustments. Agile also encourages teams to focus on what truly matters, concentrating on tasks that add value and avoiding unnecessary work.

Agile as a Mindset: Agile represents a shift in culture that values adaptability, collaboration, and client happiness. It gives team members more authority and promotes a cooperative and upbeat work atmosphere.

Quick Response to Change: Agile fosters a culture that allows teams to respond swiftly to constantly shifting priorities and requirements. This adaptability is particularly useful in sectors of the economy or technology that experience fast changes.

Regular Demonstrations: Agile techniques place a strong emphasis on regular demonstrations of project progress. Stakeholders may clearly see the project's status, upcoming problems, and upcoming new features due to this transparency.

Cross-Functional Teams: Agile fosters self-organizing, cross-functional teams that share information effectively, communicate more effectively and feel more like a unit.

Agile Software Development Process

Agile software development, often just called Agile, focuses on being flexible and practical when delivering software. Instead of launching everything at once, Agile delivers small, valuable updates to users over time. This approach allows teams to adjust and improve the product along the way, verifying that each update brings real value to the users. It's all about making progress in manageable steps and responding quickly to changes.

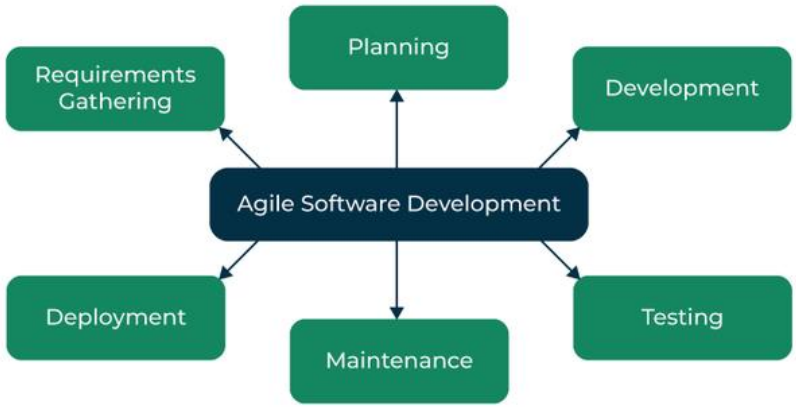


Figure 5.1

1. Requirements Gathering

This is the first step where the development team works closely with the customer to understand what they really need from the software. The team listens carefully to the customer's needs, then sorts and prioritizes these requirements to make sure the most important features are developed first.

2. Planning

In this stage, the team creates a clear plan for how they'll build the software. They decide which features to focus on in each development cycle (called an iteration). Think of it like mapping out the journey of the project, so everyone knows what to expect and when things will be delivered.

3. Development

This is where the team starts turning their plan into reality. They work in short, focused cycles, building small, usable pieces of the product. Each cycle builds on the last, which helps the team stay on track and get quick feedback to keep improving.

4. Testing

As the software gets built, it's also tested to make sure it works properly and meets the customer's needs. Testing ensures the product is of high quality and free from errors, so problems are caught early on before they become bigger issues.

5. Deployment

Once everything is tested and working as expected, the software is deployed, which means it's ready for customers or end-users to start using. It's the moment when all the development work comes to life.

6. Maintenance

Even after the software is released, the job isn't done. The team keeps maintaining the software, verifying it continues to work well and stays up-to-date with any new needs or changes from the customer. This keeps the software relevant and helpful over time.

Agile Software Development is widely used by software development teams and is considered to be a flexible and adaptable approach to software development that is well-suited to changing requirements and the fast pace of software development. Agile is a time-bound, iterative approach to software delivery that builds software incrementally from the start of the project, instead of trying to deliver all at once.

Agile Software Development Cycle

Let's see a brief overview of how development occurs in Agile philosophy.

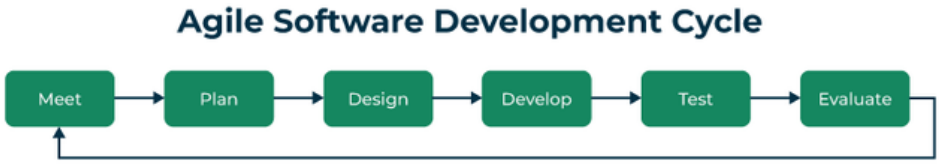


Figure 5.2

Step 1: In the first step, concept, and business opportunities in each possible project are identified and the amount of time and work needed to complete the project is estimated. Based on their technical and financial viability, projects can then be prioritized and determined which ones are worthwhile pursuing.

Step 2: In the second phase, known as inception, the customer is consulted regarding the initial requirements, team members are selected, and funding is secured. Additionally, a schedule outlining each team's responsibilities and the precise time at which each sprint's work is expected to be finished should be developed.

Step 3: Teams begin building functional software in the third step, iteration/construction, based on requirements and ongoing feedback. Iterations, also known as single development cycles, are the foundation of the Agile software development cycle.

5.2 Agile Teams

A collection of people arranged to collaborate effectively and deliver useful goods or services in a flexible and versatile way is called an agile team. Although, software development gave rise to the idea of agile, it has subsequently been used in a variety of fields and project kinds.

- In order to react quickly to evolving needs, the Agile methodology places a strong emphasis on iterative development, flexibility and customer input.
- Extreme Programming (XP), Kanban and Scrum are popular frameworks for putting agile techniques into practice. To achieve the

concepts and ideals of agility, agile teams may stick to particular guidelines and practices provided by these frameworks.

Characteristics of an Agile Team:

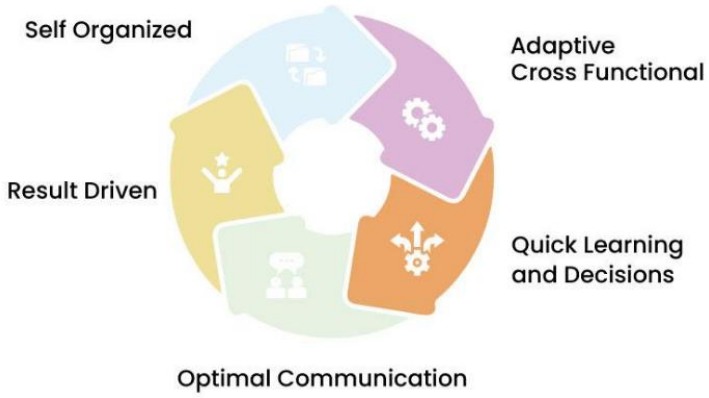


Figure 5.3

Clear Purpose: An agile team works on a clearly defined common objective that provides direction and alignment to the project.

Cross Functional: The agile teams are cross-functional with members bringing diverse skillsets - developers, testers, designers, product experts etc. This diversity enables the team to holistically drive a product to completion without dependencies.

Result-Driven/Metrics Driven Approach: Agile teams are relentlessly focused on delivering tangible, working results in the form of shippable product increments. The progress of the development is measured through metrics like velocity, defect rates and benchmarked through value delivered to and feedback from end-customers.

Communication: The Agile team encourages regular, open dialogue as well as information exchange among team members. It involves having the ability to recognize the motives and useful opinions of others, as well as having a constructive discussion about the expected benefit, the intended outcomes, and the tasks that each sprint (a set period during which specific work has to be completed and made ready for review) needs to complete. The group can also discuss with individuals outside the organization as well.

Quick Cycles of Learning and Decision-Making: Agile teams are formed to adapt to dynamic, uncertain contexts. As a result, their learning, product development, and decision-making cycles are inherently brief, which leads to constant tiny, targeted adjustments that gradually increase value.

Adaptive: The agile teams constantly inspect and adapt to changing needs. The iterative approach allows for feedback and course corrections. The roles are flexible based on work assigned. Moreover, plans are revised routinely.

Self-Organization: The Agile teams are empowered to self-organize in the way they best see fit to accomplish goals. They define the optimal workflows, tasks, and norms best suited for the product development. It also leads to leadership in the team emerging organically.

Roles and Responsibilities of an Agile Team

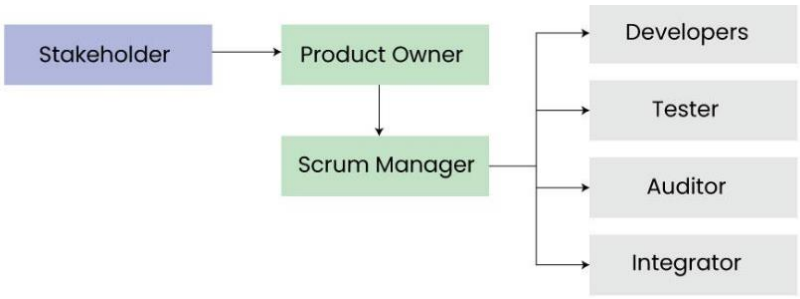


Figure 5.4

Product Owner: The product owner is responsible for representing the stakeholders and the end customers. They define the product vision and roadmap, prioritize the product backlog, and accept or reject the resulting work or outcome. They also manage stakeholders expectation, communicate goals, and ensures the team is delivering maximum business value.

Scrum Master: The scrum master is responsible for managing the agile processes and removing obstacles that prevent the development team from being productive. They facilitate meetings, promote collaboration, and encourage the team to improve its practices. In simple terms, they act as a binding force towards reaching conclusion of the project.

Development Members/Developers: The developers are a cross-functional group that carries out the actual work to build the product incrementally. They collaborate daily in short, high-paced meetings to analyse, design, develop, test, and implement the user stories from the product backlog. The development team are usually structured yet flexible and self organized while carrying their task execution.

Stakeholders: Stakeholders are individuals or groups who are impacted by the project. They may be within or external to the organization. They provide feedback on the requirements while reviewing progress of the project and also evaluate the final product. The stakeholders are usually engaged and kept in loop to allow the project team to meet their required business needs.

Integrator: The integrator is responsible for technical integration of work by the development team. They ensure that the software/hardware components fit together properly. The integrator also makes sure the system meets quality standards and integrates smoothly with external systems.

Independent Auditor and Tester: The independent tester objectively evaluates the system to ensure the quality of the product. They audit the product for bugs and flaws from an unbiased perspective. The tester identifies defects and works with the team to get them fixed.

Advantages of an Agile Team:

Increased Communication & Collaboration: The agile team promotes various practices like daily meetings, retrospectives, and reviews that enable real-time and transparent communication among team members. The constant interactions lead to greater coordination which helps identify impediments in the early stages of product development.

Greater Flexibility: The agile teams can demonstrate enhanced flexibility and can swiftly change direction or requirements either to fulfil the features for an optimal product or according to the client if needed. The individual team members are also empowered to take on varied roles outside their specialty to

meet emerging demands. This is possible by a less rigid structure and hierarchy, allowing teams to self-organize organically.

Speedy Delivery of Product: Agile enables speedy delivery of product through short sprints and iterations that focus effort on consumable solutions released frequently. Just-in-time planning removes wasted time in designing everything upfront. With a DevOps approach, automation accelerates build, testing and release cycles dramatically. Absorbing rapid feedback through daily stand-ups and demos also speeds up delivery.

Transparency: Information exchange can provide real-time visibility into the status, issues, open challenges, etc, while working in a team. The collective ownership and accountability leaves no room for finger-pointing. This also leads to open dialogues which builds trust and exposes uncertainties in early development of the product.

Faster Feedback: The iterative approach with regular deliveries allows for early and frequent response from customers and end-users. It can also be gained by engaging customers during sprints which enables correction in the course of development.

Continuous Improvements: Once constructive feedbacks are provided, the dedicated the agile team dedicates team for regular introspection on what went well and what needs tuning, i.e., teams constantly look for ways to optimize and improve the workflow of their project.

Reduction in Risks: Agile teams can reduce possible risks by dividing work into smaller chunks, allowing uncertainties to be exposed progressively. This is done through continuous testing and integration which uncovers defects early when they are cheaper to fix.

5.3 Team and Scrum

A Scrum Team is like a special group that works on projects using a method called Scrum. In this group, different people can do various jobs, and they work together to create a part of a product. They aim to finish and deliver

something valuable in a short period, which they call a "Sprint." Scrum Team is just a group of people working together to create something awesome, and they have the freedom to decide the best way to do it.

Structure of Scrum Team

Here are the following structure of Scrum Team:

Team Size: Scrum teams are small, usually between five to nine members. It's better to have multiple small teams working on a feature rather than larger teams.

Cross-Functional Collaboration: Scrum teams are designed to deliver complete features, emphasizing the importance of collaboration among team members. Cross-functional teams support each other's strengths and weaknesses.

Self-Organization: Teams are set up to be self-organizing, which means they have a clear purpose and are encouraged to work independently.

Input into Team Design: Ideally, team members should be involved in shaping how their teams are organized.

Scrum Team Roles and Responsibilities

The structure of a scrum team suggests working together, open communication, and adaptability. Usually, it includes the following essential components:

Product Owner

The team member who understands the needs of the customer and their relative business value is known as the product owner. After that, They can communicate the customer's needs and preferences back to the Scrum team. The product owner needs to understand the product's business case as well as the features that customers desire. To ensure that the team is correctly carrying out the product vision, he must be accessible for consultation. The Product Owner is in charge of overseeing the Product Backlog, which comprises the following items, and most crucially, he must have the power to make any decisions required to finish the project:

- Clearly expressing items from the product backlog.

- Arranging the items in the product backlog in order of greatest achievement of objectives and missions.
- Maximizing the worth of the work that the team completes.
- Ensuring that everyone can see, understand, and access the product backlog, which outlines the tasks the team will continue to work on.

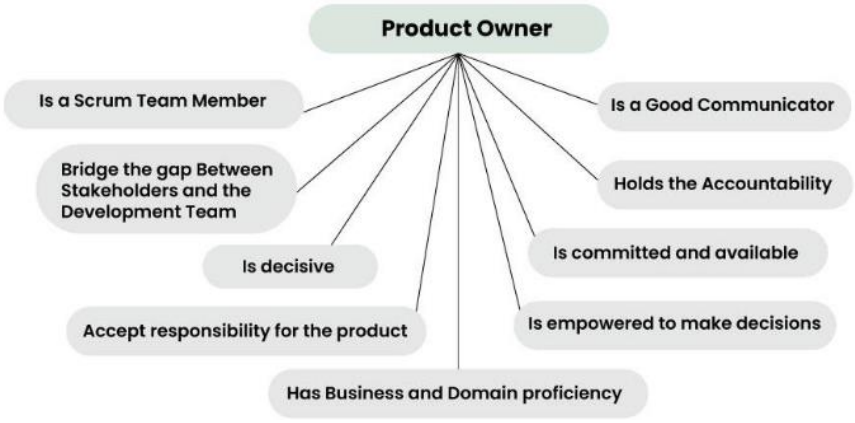


Figure 5.5

Scrum Master

The scrum master assists in removing any obstacles that might be affecting the team's productivity and in holding members of the team accountable for their commitments to the company. They reviewed work and deliverables with the team regularly, usually once a week. A scrum master's job is to coach and inspire team members, not impose rules on them.

A scrum master's responsibilities include:

- Make sure everything goes as planned.
- Remove any barriers that affect output
- Plan the important meetings and events.

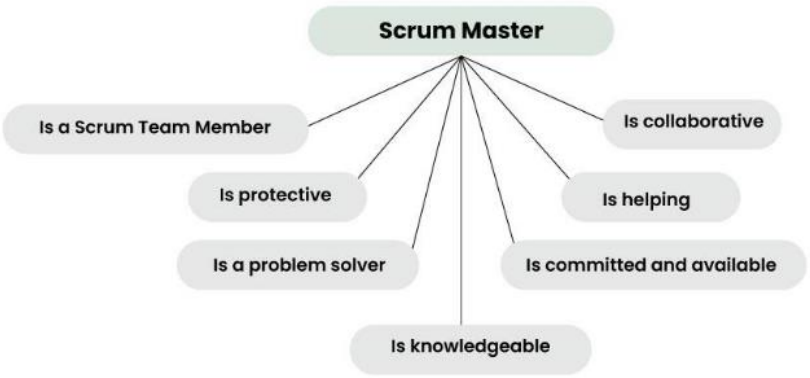


Figure 5.6

Development Team

The organization provides structure and authority to Development Teams so they can plan and oversee their own work. The Development Team's overall effectiveness and efficiency are maximized by the resulting eynergy. The following traits are present in development teams:

- The Development Team is not given instructions on how to convert the Product Backlog into potentially releaseable functionality increments, not even by the Scrum Master;
- Cross-functional development teams possess all the teamwork abilities required to produce a product increment.
- Regardless of the domains that need to be addressed, such as testing, architecture, operations, or business analysis, Scrum does not recognize sub-teams within the Development Team.
- Although each member of the Development Team may have specific expertise and areas of interest, the Development Team as a whole is ultimately accountable.

Figure 5.6

What is the Ideal Size of a Scrum Team?

Scrum teams are typically composed of five to nine members, although seven is the optimal number. The development team consists of five to seven people, plus one scrum master and a product owner. Sub-teams do not exist. The scrum team members ought to be full-time employees, preferably based in the same office. If the work needs to be done in multiple locations, a scrum team should be assigned to each location.

How do Scrum Teams Work?

The Scrum Guide states that a Scrum team should consist of ten or fewer members. The size of the team is primarily determined by the project being worked on. By adhering to a framework for rapid delivery and iterative planning, the Scrum framework seeks to provide value to the end user.

Transparency: When describing the product backlog items and customer/stakeholder priorities, product owners in particular must be precise and unambiguous. In order to swiftly address obstacles and roadblocks, the development team should also be open and honest about them.

Accountability: The completion of the final sprint goal and self-accountability are the two main goals of the Scrum team.

Self-organization: Each team member needs to be aware of their own duties and responsibilities and take the initiative to solve problems.

Benefits of a Scrum Team

Here are the following Benefits of Scrum Team:

Work happens Simultaneously: Rather than working on various project components one after the other, Scrum teams work on them simultaneously. This helps partners save time by enabling them to make ongoing, crucial changes as the project progresses rather than at the very end. Furthermore, collaborating as a team while working concurrently encourages teams to incorporate different viewpoints into their work. The final products' quality will only increase as a result.

Workflow processes are made clear: Within the scrum framework, there are particular workflow benchmarks that assist teams in staying focused and on task. Project planning, release planning, sprint planning, sprints, daily scrum, sprint review, and retrospective are all phases that scrum teams go through. Different collaborative processes are needed for each of these phases. For example, sprints are brief development cycles that last anywhere from one day to four weeks, during which the team concentrates on producing products that can be shipped.

Return on Investment (ROI) increases and risk decreases: Organizations that use scrum teams typically see an increase in return on investment (ROI), which indicates that the benefits of the investment outweigh the expenses. Teams using Scrum work faster and more effectively than teams using other frameworks. This implies that over time, they may even need less labor since they make fewer expensive errors. A company's return on investment is frequently higher when it invests less in the completion of a high-value project. Furthermore, if an organization consistently collaborates with a scrum team that boosts ROI, it might encounter less investment risk in project management.

Team Morale Improves: Both the Scrum framework and the guiding Agile principles are fundamentally human-centered, focusing on the capabilities and workflows of the employees. Scrum teams really value face-to-face

communication, teamwork, feedback, and sustainable development. The Scrum framework also requires teams to take time to reflect on what is working and what is not, so they can adjust their workflow as needed.

5.4 Branches

Software engineering branches into specialized fields focused on developing, deploying, and maintaining software systems. Key areas include Web (front-end, back-end, full-stack), Mobile (iOS/Android), Cloud computing (AWS, Azure), AI/Machine Learning, DevOps/Infrastructure, Embedded Systems, Game Development, and Security engineering. These specializations, often categorized as either systems or application software development, are tailored to specific technologies and platforms.

Web Software Engineering: Focuses on creating, building, and maintaining web applications, including user-facing interfaces (front-end) and server-side logic (back-end).

Mobile Software Engineering: Specializes in developing applications for mobile devices (iOS and Android), focusing on performance, portability, and power efficiency.

Cloud Software Engineering: Involves designing, building, and managing software on cloud platforms like AWS, Google Cloud, or Azure.

AI/Machine Learning Engineering: Integrates AI and machine learning models into software applications to enable predictive analysis and automation.

DevOps/Infrastructure Engineering: Bridges the gap between development and operations, focusing on CI/CD pipelines, system administration, and infrastructure management.

Embedded Systems Engineering: Develops software for specialized hardware, such as sensors, IoT devices, aviation, and automotive systems.

Software Quality Assurance (QA) & Testing: Ensures software quality by testing for bugs, performance issues, and usability to maintain high standards.

Game Development: Focuses on creating interactive software for entertainment on consoles, PCs, and mobile platforms.

Security Engineering: Focuses on identifying vulnerabilities and building secure software systems to protect against cyber threats.

Data Engineering: Focuses on building systems for data collection, storage, and processing, often supporting data scientists.

These branches frequently overlap, requiring a blend of skills in coding, system design, and specialized, technology-focused expertise.

5.5 Pull Requests

Pull requests are an essential component of collaborative software development workflows. They enable developers to propose and discuss changes to a codebase in a structured and transparent manner.

In software engineering, a pull request (also known as a merge request) is a mechanism for submitting proposed modifications to a shared codebase. This allows other team members to review, discuss, and ultimately merge the changes into the main project.

At the core of pull requests is version control, typically facilitated by distributed version control systems (DVCS) like Git. Git is a powerful open-source version control system that enables developers to manage and track project changes effectively. While Git is a command-line tool, popular web-based platforms (i.e., SaaS solutions) like GitHub, GitLab, and Bitbucket have emerged as collaborative hubs where developers can host their Git repositories, manage their projects, and coordinate with team members through pull requests. Pull requests facilitate collaborative code development, enabling teams to work on a shared codebase while maintaining a clear and organized history of changes.

Pull Request Workflow

Effective software development relies on a structured approach to managing code changes and collaborating within a team. A key component of this

process is the pull request workflow, which enables parallel development, code review, and seamless integration of new features and bug fixes into the main codebase.

Versioning and Branching for Feature Development

Effective software development relies on a well-structured branching model to enable parallel development and maintain a clear, organized codebase. The branching model typically involves a main branch, usually named "main" or "master," which serves as the primary, stable version of the codebase. Developers create separate source branches from this main branch to work on new functionality or bug fixes, allowing multiple features to be developed simultaneously without disrupting the main codebase.

When starting a new feature or addressing a bug, developers create a new branch, often named based on the specific task, such as "feature/new-login-page" or "bugfix/fix-checkout-flow." This branching strategy allows developers to experiment, make changes, and test their work in isolation without directly impacting the main codebase. As the feature or bug fix progresses, the developer can regularly merge the latest changes from the main branch into their feature branch to keep it up to date and resolve any potential conflicts early on.

Submitting a Pull Request

Once a developer has completed their work on a feature branch, they can merge their changes back into the main codebase by submitting a pull request. A pull request is a request to merge the changes from a feature branch into the main branch, which triggers a review process by other team members.

The process of creating a pull request typically involves the following steps:

- Committing the changes to the feature branch.
- Pushing the feature branch to the remote repository (e.g., GitHub, GitLab, Bitbucket).
- Initiating the pull request on the hosting platform, providing a clear title and description of the changes.

- Referencing any relevant issues, tasks, or other related information in the pull request description.
- Optionally, requesting specific team members to review the changes.

The pull request typically includes a title that briefly describes the changes, a detailed description that explains the purpose and scope of the changes, and a list of the individual commits that make up the feature or bug fix. This information helps the reviewers understand the context and rationale behind the proposed changes, allowing them to provide meaningful feedback and suggestions.

Review and Merging

Once a pull request is submitted, the review process begins. The pull request validation process in a GitHub repository helps ensure the quality and integrity of the codebase before changes are merged into the main branch.

Team members, often called "reviewers," examine the proposed changes, provide feedback, and suggest improvements or modifications. During the review, team members can examine the changes, provide comments, and suggest improvements. This collaborative code review allows for the identification of potential issues, the exchange of knowledge, and the enhancement of code quality.

Once the pull request has been reviewed and approved by the necessary team members, it can be merged into the main branch. This process typically involves a final check to ensure the changes do not introduce regressions or conflicts. Then, the pull request is officially merged, permanently adding the new feature or bug fix to the main codebase.

Benefits of Pull Requests

The pull request workflow offers several key benefits to software development teams:

1. Collaborative Code Review

Pull requests enable efficient code review, facilitate knowledge sharing, and improve code quality. By requiring team members to review and approve

changes, potential issues, bugs, or suboptimal design decisions can be identified and addressed before the changes are integrated into the main codebase.

Pull requests can be integrated with Continuous Integration (CI) pipelines, which automatically run tests, linting, and other quality checks before allowing the changes to be merged, ensuring code stability and reliability.

2. Traceability and Documentation

The pull request history provides a clear and centralized record of changes, contributions, and discussions, improving project documentation and traceability. This traceability allows team members to understand the rationale behind specific changes, track the evolution of the codebase, and quickly identify the source of any issues or regressions.

3. Automated Checks and Continuous Integration

Pull requests can be integrated with continuous integration (CI) pipelines, which automatically run tests, linting, and other quality checks before merging the changes, ensuring code stability and reliability.

Best Practices for Effective Pull Requests

To ensure the success of the pull request workflow in data engineering, it's essential to follow the best practices:

Clear and concise descriptions: Provide a clear and concise title and description for each pull request, outlining the purpose of the changes and any relevant context.

Granular and focused changes: Break down larger changes into smaller, more manageable pull requests to facilitate better code review and easier rollback in case of issues.

Thorough code reviews: Encourage team members to actively participate in the code review process, providing constructive feedback and suggestions for improvement.

Addressing review comments: Respond to comments promptly and make the necessary changes to address any concerns raised by the reviewers.

Maintaining a clean commit history: Ensure a clean and organized commit history by squashing or amending commits as necessary to maintain the overall clarity and traceability of the change history.

5.6 Agile Iterations

Agile iterations are short, time-boxed cycles—typically 1–4 weeks—where software teams plan, develop, test, and deliver functional, incremental improvements. Known as sprints in Scrum, these cycles allow for continuous feedback and adaptation, ensuring high-value, working software is produced at regular intervals instead of waiting until the final project release.

Key Components of Agile Iterations

Timeboxed: Iterations are fixed-length, providing a consistent cadence for delivery.

Workflow: Typically consists of five steps: plan requirements, develop, test, deliver, and gather feedback.

Incremental & Iterative: Work is broken down into small, manageable pieces (incremental) and developed in repeating cycles (iterative).

Continuous Improvement: Each iteration ends with a review and retrospective to improve the product and process for the next cycle.

Structure of an Iteration

Planning: The team defines the iteration goal and selects backlog items.

Execution (Development): Developers build the software based on requirements.

Testing/Quality Assurance: Continuous testing is performed to ensure high quality.

Review/Demo: The working software is shown to stakeholders for feedback.

Retrospective: The team reflects on how to improve their process.

Benefits

Flexibility: Requirements can be adjusted between iterations.

Faster Delivery: Functional software is delivered continuously.

Reduced Risk: Early detection of issues through testing and feedback.
Enhanced Collaboration: Active stakeholder involvement throughout the development process.

Common frameworks like Scrum and Extreme Programming (XP) rely heavily on these iterations to maintain a fast, adaptable pace in software engineering.

5.7 Reporting and fixing bugs

Software testing is the process of verifying that a software product or application is doing what it is supposed to do. The benefits of testing include preventing distractions, reducing development costs, and improving performance.

A malfunction in the software/system is an error that may cause components or the system to fail to perform its required functions. In other words, an error encountered during the test can cause malfunction. For example, incorrect data description, statements, input data, design, etc. There are many different types of software testing, each with specific goals and strategies. Some of them are below:

- Acceptance Testing: Ensuring that the whole system works as intended.
- Integration Testing: Ensuring that software components or functions work together.
- Unit Testing: To ensure that each software unit operates as expected. The unit is a testable component of the application.
- Functional Testing: Evaluating activities by imitating business conditions, based on operational requirements. Checking the black box is a common way to confirm tasks.
- Performance Testing: A test of how the software works under various operating loads. Load testing, for example, is used to assess performance under real-life load conditions.

- Re-testing: To test whether new features are broken or degraded. Hygiene checks can be used to verify menus, functions, and commands at the highest level when there is no time for a full reversal test.

Reasons Why Bugs Occur?

The Reasons Why Bugs Occur are as follows:

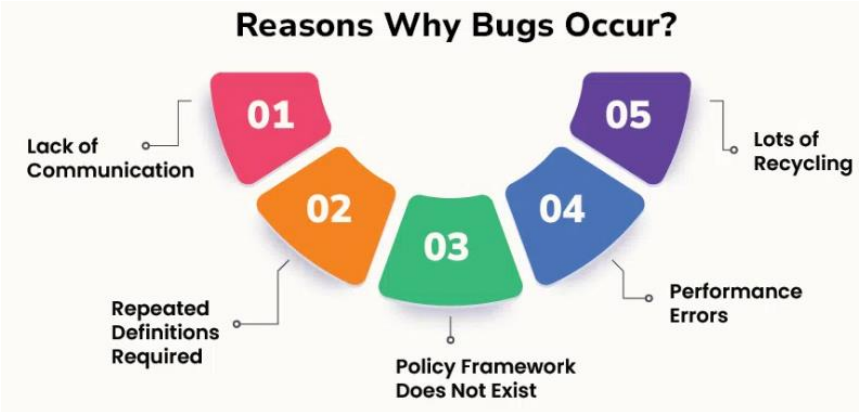


Figure 5.7

1. Lack of Communication

- This is a key factor contributing to the development of software bug fixes.
- Thus, a lack of clarity in communication can lead to misunderstandings of what the software should or should not do. In many cases, the customer may not fully understand how the product should ultimately work.
- This is especially true if the software is designed for a completely new product. Such situations often lead to many misinterpretations from both sides.

2. Repeated Definitions Required

- Constantly changing software requirements creates confusion and pressure in both software development and testing teams.
- Usually, adding a new feature or deleting an existing feature can be linked to other modules or software components. Observing such problems causes software interruptions.

3. Policy Framework Does Not Exist

- Also, debugging a software component/software component may appear in a different or similar component.
- Lack of foresight can cause serious problems and increase the number of distractions.
- This is one of the biggest problems because of what interruptions occur as engineers are often under pressure related to timelines;
- constantly changing needs, increasing the number of distractions, etc.
- Addition, Design and redesign, UI integration, module integration, database management all add to the complexity of the software and the system as a whole.

4. Performance Errors

- Significant problems with software design and architecture can cause problems for systems. Improved software tends to make mistakes as programmers can also make mistakes.
- As a test tester, data/announcement reference errors, control flow errors, parameter errors, input/output errors, etc.

5. Lots of Recycling

- Resetting resources, redoing or discarding a finished work, changes in hardware/software requirements may also affect the software.
- Assigning a new developer to a project in the middle of nowhere can cause software interruptions. This can happen if proper coding standards are not followed, incorrect coding, inaccurate data transfer, etc.
- Discarding part of existing code may leave traces on other parts of the software; Ignoring or deleting that code may cause software interruptions.
- In addition, critical bugs can occur especially with large projects, as it becomes difficult to pinpoint the location of the problem.

Life Cycle of a Bug in Software Testing

Below are the steps in the lifecycle of the bug in software testing:

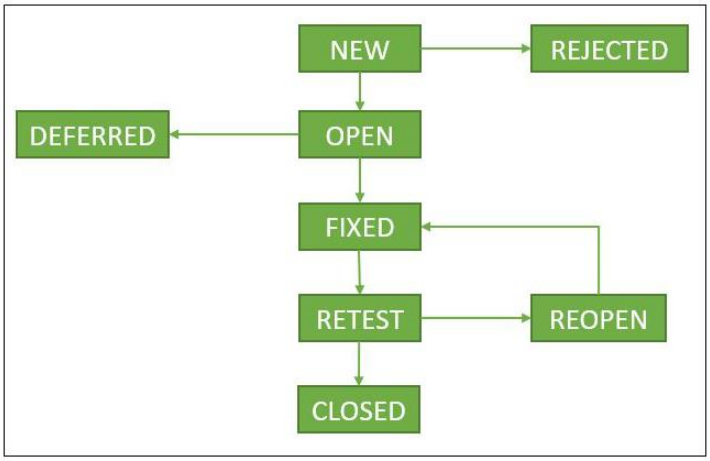


Figure 5.8

Open: The editor begins the process of analyzing bugs here, where possible, and works to fix them. If the editor thinks the error is not enough, the error for some reason can be transferred to the next four regions, Reject or No, i.e. Repeat.

New: This is the first stage of the distortion of distractions in the life cycle of the disorder. In the later stages of the bug's life cycle, confirmation and testing are performed on these bugs when a new feature is discovered.

Shared: The engineering team has been provided with a new bug fixer recently built at this level. This will be sent to the designer by the project leader or team manager.

Pending Review: When fixing an error, the designer will give the inspector an error check and the feature status will remain pending 'review' until the tester is working on the error check.

Fixed: If the Developer completes the debugging task by making the necessary changes, the feature status can be called "Fixed."

Confirmed: If the tester had no problem with the feature after the designer was given the feature on the test device and thought that if it was properly adjusted, the feature status was given "verified".

Open again / Reopen: If there is still an error, the editor will then be instructed to check and the feature status will be re-opened.

Closed: If the error is not present, the tester changes the status of the feature to 'Off'.

Check Again: The inspector then begins the process of reviewing the error to check that the error has been corrected by the engineer as required.

Repeat: If the engineer is considering a factor similar to another factor. If the developer considers a feature similar to another feature, or if the definition of malfunction coincides with any other malfunction, the status of the feature is changed by the developer to 'duplicate'.

Few more stages to added here are

- Rejected: If a feature can be considered a real factor the developer will mean “Rejected” developer.
- Duplicate: If the engineer finds a feature similar to any other feature or if the concept of the malfunction is similar to any other feature the status of the feature is changed to ‘Duplicate’ by the developer.
- Postponed: If the developer feels that the feature is not very important and can be corrected in the next release, however, in that case, he can change the status of the feature such as 'Postponed'.
- Not a Bug: If the feature does not affect the performance of the application, the corrupt state is changed to “Not a Bug”.

Bug Report

Here are the template for a detailed bug report.

Defect/ Bug Name: A short headline describing the defect. It should be specific and accurate.

Defect/Bug ID: Unique identification number for the defect.

Defect Description: Detailed description of the bug including the information of the module in which it was detected. It contains a detailed summary including the severity, priority, expected results vs actual output, etc.

Severity: This describes the impact of the defect on the application under test.

Priority: This is related to how urgent it is to fix the defect. Priority can be High/ Medium/ Low based on the impact urgency at which the defect should be fixed.

Reported By: Name/ ID of the tester who reported the bug.

Reported On: Date when the defect is raised.

Steps: These include detailed steps along with the screenshots with which the developer can reproduce the same defect.

Status: New/ Open/ Active

Fixed By: Name/ ID of the developer who fixed the defect.

Data Closed: Date when the defect is closed.

Factors to be Considered while Reporting a Bug

- The whole team should clearly understand the different conditions of the trauma before starting research on the life cycle of the disability.
- To prevent future confusion, a flawed life cycle should be well documented.
- Make sure everyone who has any work related to the Default Life Cycle understands his or her best results work very clearly.
- Everyone who changes the status quo should be aware of the situation which should provide sufficient information about the nature of the feature and the reason for it so that everyone working on that feature can easily see the reason for that feature.
- A feature tracking tool should be carefully handled in the course of a defective life cycle work to ensure consistency between errors.

Bug Tracking Tools

Below are some of the Bug Tracking Tools

1. KATALON TESTOPS

Katalon TestOps is a free, powerful orchestration platform that helps with your process of tracking bugs. TestOps provides testing teams and DevOps teams with a clear, linked picture of their testing, resources, and locations to launch the right test, in the right place, at the right time.

Features

- Applies to Cloud, Desktop: Window and Linux program.
- Compatible with almost all test frames available: Jasmine, JUnit, Pytest, Mocha, etc .; CI / CD tools: Jenkins, CircleCI, and management platforms: Jira, Slack.
- Track real-time data for error correction, and for accuracy.
- Live and complete performance test reports to determine the cause of any problems.
- Plan well with Smart Scheduling to prepare for the test cycle while maintaining high quality.
- Rate release readiness to improve release confidence.
- Improve collaboration and enhance transparency with comments, dashboards, KPI tracking, possible details - all in one place.

2. KUALITEE

Collection of specific results and analysis with solid failure analysis in any framework. The Kualitee is for development and QA teams look beyond the allocation and tracking of bugs. It allows you to build high-quality software using tiny bugs, fast QA cycles, and better control of your build. The comprehensive suite combines all the functions of a good error management tool and has a test case and flow of test work built into it seamlessly. You would not need to combine and match different tools; instead, you can manage all your tests in one place.

Features

- Create, assign, and track errors.
- Tracing between disability, needs, and testing.
- Easy-to-use errors, test cases, and test cycles.
- Custom permissions, fields, and reporting.
- Interactive and informative dashboard.
- Integration of external companies and REST API.

- An intuitive and easy-to-use interface.

3. QA Coverage

QACoverage is the place to go for successfully managing all your testing processes so that you can produce high-quality and trouble-free products. It has a disability control module that will allow you to manage errors from the first diagnostic phase until closed. The error tracking process can be customized and tailored to the needs of each client. In addition to negative tracking, QACoverage has the ability to track risks, issues, enhancements, suggestions, and recommendations. It also has full capabilities for complex test management solutions that include needs management, test case design, test case issuance, and reporting.

Features

- Control the overall workflow of a variety of Tickets including risk, issues, tasks, and development management.
- Produce complete metrics to identify the causes and levels of difficulty.
- Support a variety of information that supports the feature with email attachments.
- Create and set up a workflow for enhanced test visibility with automatic notifications.
- Photo reports based on difficulty, importance, type of malfunction, disability category, expected correction date, and much more.

4. BUG HERD

BugHerd is an easy way to track bugs, collect and manage webpage responses. Your team and customers search for feedback on web pages, so they can find the exact problem. BugHerd also scans the information you need to replicate and resolve bugs quickly, such as browser, CSS selector data, operating system, and screenshot. Distractions and feedback, as well as technical information, are submitted to the Kanban Style Task Board, where distractions can be assigned and managed until they are eliminated. BugHerd can also

integrate with your existing project management tools, helping to keep your team on the same page with bug fixes.

Features

- Allows users to report bugs directly on the web page where they occur.
- Users can annotate their bug reports with comments, highlights, and drawings to clearly indicate issues.
- Automatically captures screenshots with each bug report.
- Enables clients and non-technical team members to report bugs without logging into the bug-tracking system.

Software testing is essential for ensuring that a product works as expected, reducing development costs, and enhancing performance. Types of testing include Acceptance, Integration, Unit, Functional, and Performance testing, each focusing on different aspects of the software.

A software bug is a malfunction causing the system to fail in performing required functions. Bugs commonly arise from lack of communication, changing requirements, poor policy frameworks, performance errors, and excessive resource recycling.

5.8 Dev/Ops

DevOps is a modern approach to software development that brings development and operations teams together to deliver applications faster and more reliably. It focuses on collaboration, automation, and continuous improvement across the software lifecycle.

- Encourages close collaboration between development and operations teams.
- Automates build, test, and deployment processes to reduce errors.
- Enables faster and more frequent software releases.
- Improves system reliability, monitoring, and feedback loops.

Stages of DevOps are:

- Plan Stage

- Teams define project requirements, goals, timelines, and success metrics.
- Work is broken down into tasks and user stories to ensure clarity and alignment.
- Collaboration between development and operations begins at this stage to avoid future bottlenecks.
- Common Tools: Jira, Confluence, Azure Boards, Trello.

Code Stage

- Developers write application code and configuration files following best practices.
- Version control systems are used to manage changes and collaborate efficiently.
- Code reviews and branching strategies help maintain code quality and stability.
- Common Tools: Git, GitHub, GitLab, Bitbucket.

Build Stage

- The application code is automatically compiled and packaged into deployable artifacts.
- Dependencies and libraries are resolved to ensure consistent builds across environments.
- Build automation ensures faster feedback and reduces manual errors.
- Common Tools: Jenkins, GitLab CI/CD, Maven, Gradle, Docker.

Test Stage

- The software undergoes thorough testing to catch bugs and security risks before release.
- Diffrent Testing methods includes unit, integration, performance, and security testing.
- Issues are identified early, that will reduce the cost and impact of failures.

- Common Tools: Selenium, JUnit, TestNG, SonarQube, JMeter

Release Stage

- Tested builds are prepared and approved for deployment.
- Release versions are tagged and documented for traceability.
- Deployment strategies are planned to minimize risk during production rollout.
- Common Tools: Git tags, Jenkins, GitLab CI/CD, ArgoCD.

Deploy Stage

- The application is deployed to production or target environments.
- Deployment strategies such as blue-green, canary, or rolling updates are used to ensure minimal downtime.
- Infrastructure automation ensures consistency across environments.
- Common Tools: Kubernetes, Helm, Ansible, Terraform.

Operate and Monitor Stage

- A key aspect of DevOps is learning from real-world performance and using that feedback to improve future releases.
- The application is continuously monitored to ensure availability and performance.
- Logs, metrics, and alerts help detect and resolve issues quickly.
- Feedback from monitoring and users is fed back into the planning stage for continuous improvement.
- Common Tools: Prometheus, Grafana, ELK Stack, Datadog, New Relic

How to Adopt a DevOps Model?

To adopt a DevOps model, ensure the following points:

Adopt a DevOps Mindset: Adopt a devops mindset by fostering collaboration, shared ownership, and accountability across development, operations, QA, and security teams.

Recognize Infrastructure Requirements: Recognize infrastructure requirements by assessing current workflows, identifying bottlenecks, and evaluating scalability, availability, and security needs

Create a DevOps Strategy: Create a devops strategy by defining clear, measurable goals such as faster deployments, improved reliability, and better cross-team collaboration

Choose the Right DevOps Tools:Choose the right devops tools that align with your workflows for version control, ci/cd, infrastructure automation, and monitoring

Increase Test Automation: Increase test automation and align qa with development to catch defects early and ensure consistent software quality

Adopt Application Containerization: Adopt application containerization to standardize environments, simplify deployments, and improve scalability

Focus on iterative adoption by continuously monitoring performance, collecting feedback, and optimizing processes over time.

DevOps for AI and ML

Even though Artificial Intelligence (AI) and Machine Learning (ML) are still growing in DevOps, they are already making a big difference.

Handling Big Data: DevOps tools generate a huge amount of data from testing, deployment, and monitoring. AI and ML are great at reading all this data quickly, finding useful insights, and helping teams make faster and smarter decisions.

Saving Time with Smart Suggestions: AI can learn how developers and operations teams work, then suggest better ways to do tasks or automatically set up the needed tools and servers, reducing manual work.

Spotting Bugs Early: AI and ML can look at code and test results to find problems (like bugs) early. They can detect unusual patterns that may cause issues later and warn the DevOps team before users are affected.

Improving Security: These technologies can scan security logs and alerts to find threats, such as hacking attempts or breaches. Once something risky is

found, they can even respond automatically. For example, by blocking access or sending alerts.

5.9 Three-Tier

The chief benefit of three-tier architecture is that because each tier runs on its own infrastructure, each tier can be developed simultaneously by a separate development team. And can be updated or scaled as needed without impacting the other tiers.

For decades three-tier architecture was the prevailing architecture for client-server applications. Today, most three-tier applications are targets for modernization that uses cloud-native technologies such as containers and microservices and for migration to the cloud.

The three tiers in detail

Presentation tier

The presentation tier is the user interface and communication layer of the application, where the end user interacts with the application. Its main purpose is to display information to and collect information from the user. This top-level tier can run on a web browser, as desktop application, or a graphical user interface (GUI), for example. Web presentation tiers are developed by using HTML, CSS, and JavaScript. Desktop applications can be written in various languages depending on the platform.

Application tier

The application tier, also known as the logic tier or middle tier, is the heart of the application. In this tier, information that is collected in the presentation tier is processed - sometimes against other information in the data tier - using business logic, a specific set of business rules. The application tier can also add, delete, or modify data in the data tier.

The application tier is typically developed by using Python, Java, Perl, PHP or Ruby, and communicates with the data tier by using API calls.

Data tier

The data tier, sometimes called database tier, data access tier or back-end, is where the information that is processed by the application is stored and managed. This can be a relational database management system such as PostgreSQL, MySQL, MariaDB, Oracle, Db2, Informix or Microsoft SQL Server, or in a NoSQL Database server such as Cassandra, CouchDB, or MongoDB.

In a three-tier application, all communication goes through the application tier. The presentation tier and the data tier cannot communicate directly with one another.

Benefits of three-tier architecture

Again, the chief benefit of three-tier architecture is its logical and physical separation of functionality. Each tier can run on a separate operating system and server platform - for example, web server, application server, database server - that best fits its functional requirements. And each tier runs on at least one dedicated server hardware or virtual server, so the services of each tier can be customized and optimized without impacting the other tiers.

Other benefits (compared to single- or two-tier architecture) include:

Faster development: Because each tier can be developed simultaneously by different teams, an organization can bring the application to market faster. And programmers can use the latest and best languages and tools for each tier.

Improved scalability: Any tier can be scaled independently of the others as needed.

Improved reliability: An outage in one tier is less likely to impact the availability or performance of the other tiers.

Improved security: Because the presentation tier and data tier can't communicate directly, a well-designed application tier can function as an internal firewall, preventing SQL injections and other malicious exploits.

Three-tier application in web development

In web development, the tiers have different names but perform similar functions:

The web server is the presentation tier and provides the user interface. This is usually a web page or website, such as an ecommerce site where the user adds products to the shopping cart, adds payment details or creates an account. The content can be static or dynamic, and is developed using HTML, CSS, and JavaScript.

The application server corresponds to the middle tier, housing the business logic that is used to process user inputs. To continue the ecommerce example, this is the tier that queries the inventory database to return product availability, or adds details to a customer's profile. This layer often developed using Python, Ruby, or PHP and runs a framework such as Django, Rails, Symphony, or ASP.NET.

The database server is the data or backend tier of a web application. It runs on database management software, such as MySQL, Oracle, DB2, or PostgreSQL.

5.10 Service level objectives

Service Level Objectives (SLOs) are specific, measurable targets for a software service's performance or reliability over a designated time period. They define the acceptable level of service, often set as a percentage (e.g., 99.9% uptime), allowing teams to balance feature velocity with system stability.

Key Aspects of SLOs

Measurement: Based on Service Level Indicators (SLIs), which are quantitative metrics such as latency, uptime, or error rates.

Purpose: They bridge the gap between business goals and technical performance, setting clear expectations for users.

Error Budgets: SLOs define an "error budget" (allowable unreliability), providing an objective framework to decide when to prioritize reliability over new feature development.

Distinction from SLA: While a Service Level Agreement (SLA) is a legal contract with penalties, an SLO is an internal goal that is usually stricter than the SLA.

Common Examples

Availability: 99.95% of user login requests must succeed within a 30-day window.

Latency: 99% of search queries must return results in under 500ms.

Correctness: 99.99% of financial transactions must be processed without errors.

Implementing SLOs ensures teams can focus on user experience and, when necessary, slow down deployments to address technical debt when error budgets are depleted.

5.11 Releases and feature flags

Feature flags (or toggles) are a software engineering technique that allows teams to enable or disable functionality at runtime without deploying new code, enabling safer, decoupled, and faster releases. By acting as a "kill switch," they reduce deployment risk, facilitate Progressive Delivery (e.g., canary launches), and separate code deployment from feature release.

Key Aspects of Feature Flags in Releases

Decoupling Deployment and Release: Developers can merge code into production (deployed) but keep it hidden from users (not released), facilitating continuous integration and avoiding long-lived, complex branches.

Risk Mitigation: If a new feature causes issues, it can be instantly turned off, reducing the Mean Time to Remediate (MTTR) without requiring a full code rollback.

Progressive Rollouts/Canary Testing: Features can be released to a small subset of users (e.g., 1%–5%) to test performance and gather feedback before a full rollout, minimizing the "blast radius" of bugs.

Types of Flags:

Release Flags: Short-lived flags used to manage the rollout of new features.

Experimentation/Business Flags: Used for A/B testing or controlling access for specific user segments.

Ops Flags: Used to disable, limit, or throttle functionality during system strain (kill switches).

Best Practices: It is critical to remove flags once a feature is fully released to prevent technical debt and code complexity.

Feature flags are essential for modern DevOps, enabling safe, controlled, and rapid experimentation and deployment.

5.12 Monitoring and finding bottlenecks

In the world of system design and performance optimization, understanding and addressing bottleneck conditions are pivotal for ensuring smooth operations. A bottleneck refers to a point in a system where the flow of data or processes is limited, leading to a slowdown in overall performance. Identifying and resolving such bottlenecks are critical for enhancing efficiency and maintaining a seamless user experience.

A bottleneck condition is a limitation in a system that makes it difficult for data, resources, or activities to flow through it, which lowers overall performance. It acts as a limiting factor, constraining the system’s ability to perform tasks at its optimal speed and efficiency. Understanding the various bottleneck conditions is essential for diagnosing and addressing them effectively.

Types of Bottleneck Conditions

1. CPU Bottlenecks

CPU bottlenecks occur when the central processing unit is unable to handle the volume of processing tasks, leading to significant delays in task execution and overall system responsiveness. Such bottlenecks often arise due to intensive computational tasks, poorly optimized code, or inefficient multithreading, where the CPU becomes the limiting factor in processing data.

Mitigation Strategies for CPU Bottlenecks:

- Employ parallel processing techniques to distribute computational tasks across multiple cores, thereby maximizing CPU utilization and minimizing processing delays.
- Optimise algorithms and code to reduce unnecessary processing overhead, improving the overall efficiency of the CPU's processing capabilities.
- Upgrade to a more powerful CPU or implement specialized hardware accelerators to handle complex computational tasks efficiently.

2. Memory Bottlenecks

Memory bottlenecks occur when the system's memory resources are insufficient to meet the demands of data processing and storage, leading to increased access times and decreased overall system performance. This bottleneck type often arises when the volume of data exceeds the available memory capacity, causing frequent data swapping between the RAM and the disk, resulting in significant latency and decreased throughput.

Mitigation Strategies for Memory Bottlenecks:

- Optimise data storage and retrieval processes to minimize unnecessary data access and reduce memory consumption.
- Implement memory caching mechanisms to store frequently accessed data in the faster memory caches, reducing the frequency of data retrieval from slower memory sources.

- Upgrade the system's memory capacity or adopt high-speed memory technologies to accommodate the growing data processing demands effectively.

3. Network Bottlenecks

Network bottlenecks occur when the network bandwidth is insufficient to handle the data transmission requirements, resulting in communication delays, packet losses, and degraded network performance. Such bottlenecks often emerge in scenarios where large volumes of data are being transmitted over the network, leading to congestion and reduced data transfer speeds.

Mitigation Strategies for Network Bottlenecks:

- Implement network traffic shaping and quality of service (QoS) techniques to prioritize critical data traffic and ensure the efficient transmission of essential data packets.
- Upgrade network infrastructure components, such as routers, switches, and network cables, to support higher data transfer speeds and reduce network congestion.
- Implement data compression techniques and optimized data protocols to minimize the data payload size, reducing the overall network traffic and mitigating the risk of network bottlenecks.

4. Storage Bottlenecks

Storage bottlenecks occur when the storage infrastructure is unable to handle the data storage and retrieval demands efficiently, leading to increased latency, slow data access times, and potential data loss. Such bottlenecks often arise due to storage device limitations, inadequate storage configurations, or improper data access patterns that result in excessive disk I/O operations.

Mitigation Strategies for Storage Bottlenecks:

- Implement storage tiering mechanisms to allocate data across different storage tiers based on access frequency, ensuring that frequently accessed data resides on faster storage mediums, reducing access latency.

- Utilise data deduplication and compression techniques to optimize storage space utilization and reduce the overall data storage footprint, thereby minimizing storage I/O operations.
- Upgrade to high-speed storage devices, such as solid-state drives (SSDs) or NVMe drives, to enhance the system's storage performance and minimize storage access delays.

Causes of Bottlenecks

Insufficient Resources Allocation: Improper distribution of resources, such as CPU, memory, network bandwidth, or storage, can lead to bottleneck conditions within the system.

Inefficient Code or Algorithms: Poorly optimized code or inefficient algorithms can significantly impact system performance, leading to bottlenecks during data processing and execution.

Hardware Limitations: Outdated hardware or insufficient hardware capabilities may result in bottleneck conditions, especially when handling complex and resource-intensive tasks.

Consequences of Bottleneck Conditions

Reduced Throughput: Bottleneck conditions can lead to a decrease in the overall throughput of the system, hampering the ability to handle concurrent operations efficiently.

Increased Latency: Users may experience increased response times and delays in data retrieval due to bottleneck conditions, leading to a degraded user experience.

System Instability: Prolonged bottleneck conditions can cause system instability, leading to crashes, errors, and potential data loss, thereby impacting the system's reliability and integrity.

Ways for Bottleneck Condition Identification

Profiling Tools: Utilize specialized profiling tools to analyze the system's performance metrics, identify resource-intensive components, and pinpoint potential bottleneck areas within the system.

Performance Monitoring: Implement robust performance monitoring systems to continuously track key performance indicators, such as CPU utilization, memory usage, network traffic, and storage latency, to detect anomalies and potential bottleneck conditions in real-time.

Benchmarking: Conduct comprehensive benchmarking tests to compare the system's performance against industry standards and best practices, thereby identifying any performance gaps and potential bottleneck areas that require attention.

Common Bottleneck Scenarios

High-Traffic Load Situations: When a system experiences sudden spikes in user traffic, it can lead to CPU, memory, or network bottlenecks, affecting the system's responsiveness and stability.

Resource-Intensive Operations: Complex data processing tasks or resource-intensive operations, such as large-scale data analytics or real-time video rendering, can strain the system's resources and potentially create bottleneck conditions.

Underprovisioned Infrastructure: Inadequate allocation of resources, such as limited memory or storage capacities, can lead to frequent storage or memory bottlenecks, impacting the system's overall performance and scalability.

Mitigation Strategies

Optimized Resource Allocation: Ensure proper allocation of resources based on the system's requirements, optimizing CPU, memory, network bandwidth, and storage capacities to prevent bottleneck conditions.

Code Optimization and Algorithm Refinement: Conduct regular code reviews and optimizations to enhance the efficiency of the system's codebase and algorithms, minimizing the risk of bottleneck conditions during data processing and execution.

Infrastructure Scaling and Upgradation: Consider scaling the infrastructure or upgrading hardware components to accommodate the growing demands and prevent potential bottleneck conditions caused by resource limitations.

Load Balancing Techniques: Implement effective load balancing techniques to distribute the workload evenly across multiple servers or resources, preventing any single point of failure and mitigating the risk of bottleneck conditions.

Best Practices to Follow

Regular Performance Testing: Conduct routine performance testing and analysis to proactively identify and address potential bottleneck conditions, ensuring the system's stability and optimal performance.

Continuous Monitoring and Alerting: Implement robust monitoring and alerting mechanisms to detect any deviations from the performance benchmarks, enabling timely intervention and resolution of potential bottleneck conditions.

Scalable Architecture Design: Adopt a scalable and resilient architecture design that allows seamless horizontal and vertical scaling, ensuring the system can handle varying workloads without succumbing to bottleneck conditions.

Proactive Capacity Planning: Engage in proactive capacity planning to anticipate future resource requirements and prevent potential bottleneck conditions by ensuring adequate resource provisioning and allocation.

In conclusion, understanding, identifying, and mitigating bottleneck conditions are crucial for maintaining a high-performing and resilient system. By implementing effective identification techniques, proactive mitigation strategies, and adhering to best practices, organizations can ensure smooth operations, enhanced user experiences, and optimized system performance in the dynamic digital landscape.

5.13 Improving rendering and database performance with caching

Caching is a concept that involves storing frequently accessed data in a location that is easily and quickly accessible. The purpose of caching is to improve the performance and efficiency of a system by reducing the amount of time it takes to access frequently accessed data.

Caching acts as the local store for the data and retrieving the data from this local or temporary storage is easier and faster than retrieving it from the database. In a typical web application, we can add an application server cache and an in-memory store like Redis alongside our application server.

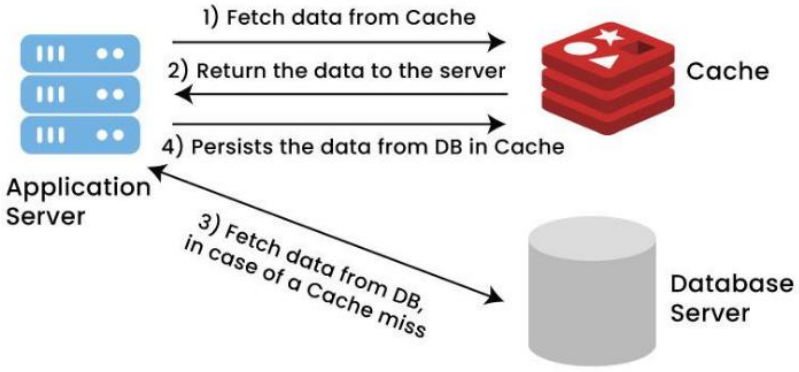


Figure 5.9

How Does Cache Work?

Web application stores data in a database. Reading data from the database needs network calls and I/O operations which is a time-consuming process. Cache reduces the network calls to the database and speeds up the performance of the system.

- When the first time a request is made a call will have to be made to the database to process the query. This is known as a cache miss.
- Before giving back the result to the user, the result will be saved in the cache.
- When the second time a user makes the same request, the application will check your cache first to see if the result for that request is cached or not.
- If it is then the result will be returned from the cache. This is known as a cache hit.
- The response time for the second time request will be a lot less than the first time.

Why not store all data in cache?

As you know there are many benefits of the cache but that doesn't mean we will store all the information in the cache memory for faster access, we can't do this for multiple reasons, such as:

- Hardware of the cache which is much more expensive than a normal database.
- Also, the search time will increase if you store tons of data in your cache.
- Cache is typically a volatile storage, meaning data is lost if the system crashes or restarts. For critical and long-term data, storing it only in cache would risk data loss.
- In short, a cache needs to have the most relevant information according to the request which is going to come in the future.

Types of Cache

In common there are four types of Cache:

1. Application Server Cache

An Application Server Cache is a storage layer within an application server that temporarily holds frequently accessed data, so it can be quickly retrieved without needing to go back to the main database each time. This helps applications run faster by reducing the load on the database and speeding up response times for users.

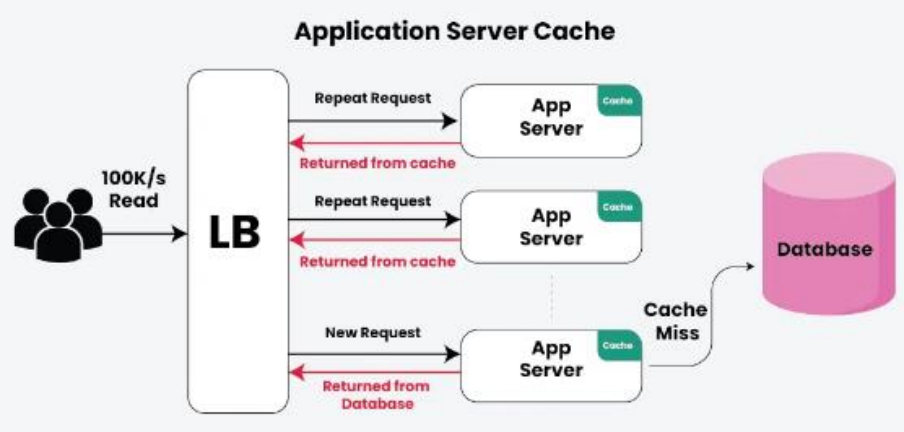


Figure 5.10

Drawbacks of Application Server Cache:

- When you add multiple servers to handle a high volume of requests.
- With several servers, a load balancer sends requests to different nodes, but each node only has its own cache and doesn't know about the cached data on other nodes.
- This results in many cache misses, meaning the data has to be re-fetched frequently, slowing things down.

2. Distributed Cache

In the distributed cache, each node will have a part of the whole cache space and then using the consistent hashing function each request can be routed to where the cache request could be found.

- Each of its nodes will have a small part of the cached data.
- To identify which node has which request the cache is divided up using a consistent hashing function, so that each request can be routed to where the cached request could be found.
- If a requesting node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to check if the data is available.

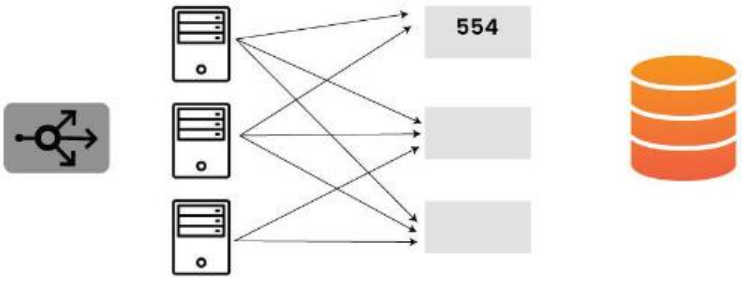


Figure 5.11

3. Global Cache

As the name suggests, you will have a single cache space and all the nodes use this single space. Every request will go to this single cache space. There are two kinds of the global cache

First, when a cache request is not found in the global cache, it's the responsibility of the cache to find out the missing piece of data from anywhere underlying the store (database, disk, etc).

Second, if the request comes and the cache doesn't find the data then the requesting node will directly communicate with the DB or the server to fetch the requested data.

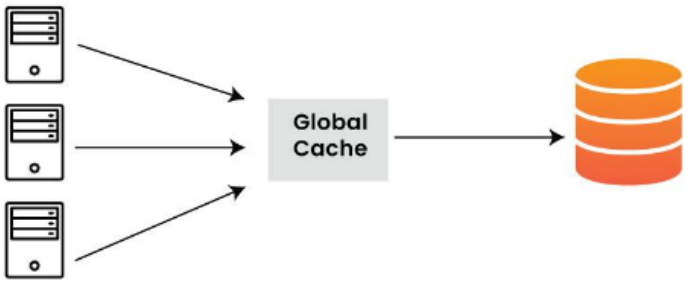


Figure 5.12

4. CDN (Content Delivery Network)

A CDN is essentially a group of servers that are strategically placed across the globe with the purpose of accelerating the delivery of web content. A CDN:

- Manages servers that are geographically distributed over different locations.
- Stores the web content in its servers.
- Attempts to direct each user to a server that is part of the CDN and close to the user so as to deliver content quickly.
- Used where a large amount of static content is served by the website.

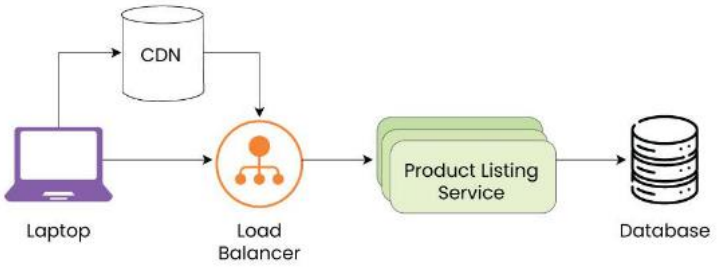


Figure 5.13

Applications of Caching

Caching is used in many areas to speed up processes, reduce load and make systems more efficient. Below are some common applications of caching:

Web Page Caching: In order to speed up loading times in the future, browsers save copies of frequently visited websites. This saves bandwidth and shortens the time it takes for a web page to load.

Database Caching: Frequent database queries can strain servers and cause lag. Caching allows apps to quickly retrieve frequently used data without repeatedly asking the database by storing it in memory.

Content Delivery Networks (CDNs): CDNs use caching to keep copies of data (such as pictures and videos) in several places throughout the globe. This enhances website performance by enabling visitors to obtain content more quickly from a nearby server.

Session Caching: Applications store session data in a cache to remember user information (like login status) between visits, making the experience seamless and personalized without needing to re-login.

API Response Caching: Frequently requested API data, like stock prices or weather data, can be cached so responses are faster, reducing the load on the server and delivering data in real-time.

Cache Invalidation Strategies

- For systems that use caching to improve performance, cache invalidation is essential. Data is temporarily kept for faster access when it is cached. However, the cached version goes out of date if the original data changes.
- In order to guarantee that users obtain the most recent information, cache invalidation techniques make sure that out-of-date records are either updated or deleted.
- Common strategies include time-based expiration, where cached data is discarded after a certain time and event-driven invalidation, triggered by changes to the underlying data.

- Proper cache invalidation optimizes performance and avoids serving users with obsolete or inaccurate content from the cache.

Eviction Policies of Caching

- For caching systems to effectively manage their limited cache capacity, eviction policies are essential. An eviction policy decides which existing item to remove when the cache is full and a new item needs to be stored.
- The Least Recently Used (LRU) policy is a popular strategy that eliminates the item that has been accessed the least recently. According to this assumption, items which have been used recently are more likely to be utilized again shortly.
- Another method is the Least Frequently Used (LFU) policy, removing the least frequently accessed items.
- Alternatively, there's the First-In-First-Out (FIFO) policy, evicting the oldest cached item.

Pros of Caching

As it maximizes resource utilization, reduces server loads and enhances overall scalability, caching is a helpful technique in software development.

Improved performance: By significantly reducing down on the time it takes to get frequently used data, caching can enhance system responsiveness and performance.

Reduced load on the original source: By significantly reducing down on the time it takes to get frequently used data, caching can enhance system responsiveness and performance.

Cost savings: Caching can reduce the need for expensive hardware or infrastructure upgrades by improving the efficiency of existing resources.

Cons of Caching

Despite its advantages, caching comes with drawbacks also and some of them are:

Data inconsistency: If cache consistency is not maintained properly, caching can introduce issues with data consistency.

Cache eviction issues: If cache eviction policies are not designed properly, caching can result in performance issues or data loss.

Additional complexity: Caching can add additional complexity to a system, which can make it more difficult to design, implement and maintain.

5.14 Defending customer data in application

Defending customer data in application software engineering requires a multi-layered, "security-by-design" approach that covers the entire data lifecycle, from collection to deletion. Key strategies include data minimization, robust encryption (at rest and in transit), strict access controls, and adherence to security frameworks like OWASP to prevent common vulnerabilities.

Core Strategies for Customer Data Defense

Implement Zero-Trust Architecture: Assume any request, even internal, is a potential threat. Require strict authentication (Multi-Factor Authentication - MFA) for all users and services.

Data Minimization: Only collect the data necessary for business purposes. The less data stored, the lower the risk of a breach, and the lower the impact if a breach occurs.

Encrypt Data Everywhere:

In Transit: Use TLS 1.2 or above (HTTPS) to protect data moving between the client and server.

At Rest: Encrypt data stored in databases, backups, and logs using strong standards like AES-256.

Secure Authentication & Authorization:

Password Hashing: Use modern, slow hashing algorithms (e.g., Argon2, Bcrypt) with unique salts for each user.

Role-Based Access Control (RBAC): Restrict employee access to sensitive data based on job function, following the principle of least privilege.

Input Sanitization & Validation: Prevent SQL injection and Cross-Site Scripting (XSS) by validating all incoming user data and using parameterized queries.

Secure API Management: Use token-based authentication (OAuth 2.0, JWT) and rate limiting to protect API endpoints.

Operational & Regulatory Practices

Regular Audits and Pen Testing: Conduct regular vulnerability assessments and penetration testing to identify weaknesses before they are exploited.

Stay Compliant: Comply with regulations such as GDPR (Europe), CCPA (California), and HIPAA (healthcare) to avoid hefty fines.

Software Updates & Dependency Management: Regularly update application components, frameworks, and plugins to patch known security holes.

Incident Response Plan: Develop a documented plan to quickly detect, contain, and notify stakeholders of a data breach.

Employee Training: Educate staff to recognize phishing, social engineering, and weak,, unhygienic password practices.

Defending Against Common Vulnerabilities (OWASP Top 10)
Following the OWASP Top 10 list is standard industry practice. Current key threats include:

Broken Access Control: Ensuring users cannot access data or functionality outside their permissions.

Cryptographic Failures: Fixing weak encryption or improper key management.

Injection: Blocking SQL, NoSQL, and OS command injection.

Vulnerable and Outdated Components: Patching libraries and frameworks.

Data Privacy by Design

Proactively incorporate security at the beginning of the development lifecycle (SDLC), rather than treating it as an afterthought. This includes using techniques like data masking to hide PII (Personally Identifiable Information) during testing and development.