SDLC TRAINING

Introduction to SDLC:

- ➢ Software Development Life Cycle (SDLC) is a framework that defines the steps involved in the development of software at each phase. It covers the detailed plan for building, deploying and maintaining the software.
- ➢ SDLC defines the complete cycle of development i.e. all the tasks involved in planning, creating, testing, and deploying a Software Product.

Software:

Software refers to a collection of programs, data, and instructions that enable a computer system to perform specific tasks or functions. It's intangible and consists of lines of code written in programming languages. Software can take various forms, including:

1. **Applications Software:** This includes programs designed for end-users to perform specific tasks, such as word processors, web browsers, games, and productivity software.
2. **System Software:** This software manages and controls computer hardware resources, such as operating systems (e.g., Windows, Linux, macOS), device drivers, and utility programs.
3. **Middleware:** Middleware acts as an intermediary between different software applications, facilitating communication and data exchange between them.

   Ex:Web Servers and Application Servers,

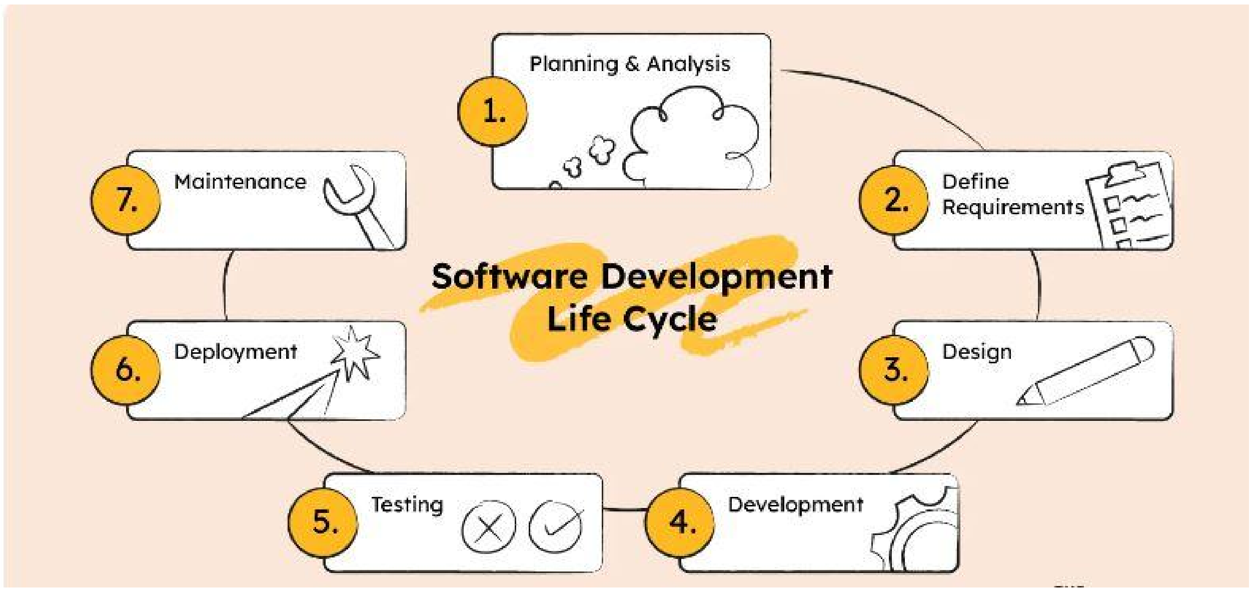
   Database Middleware: ODBC (Open Database Connectivity).
4. **Firmware:** Firmware is a type of software that is embedded in electronic devices and is responsible for controlling their operation. It is typically stored in non-volatile memory and is closely tied to the hardware of the device.

   Ex: BIOS of a computer or the software on a microcontroller, Printers,

   Digital Cameras,Smartphone Firmware

What is the Software Development Lifecycle?

- ➢ The software development lifecycle is a methodology that aims at producing the highest quality software with the lowest cost and time.
- ➢ The lifecycle provides a series of well-organized steps that guide you to build robust and high-quality software that is deployable for your end-users right away.
- ➢ The process consists of successive phases, the sync between which ensures that the software does not lack in any aspect. It includes defining the idea behind the software, discussing the purpose it will serve, visualizing what it would eventually look like,building, testing, and finally deploying it.



Planning:

1. The first phase of the SDLC is the project planning stage where you are gathering business requirements from your client or stakeholders. This phase is when you evaluate the feasibility of creating the product, revenue potential, the cost of production, the needs of the end-users, etc.
2. To properly decide what to make, what not to make, and what to make first, you can use a feature prioritization framework that takes into account the value of the software/update, the cost, the time it takes to build, and other factors.

3. Once it is decided that the software project is in line with business and stakeholder goals, feasible to create, and addresses user needs, then you can move on to the next phase.

Requirement Gathering and Analysis:

1. This phase is critical for converting the information gathered during the planning and analysis phase into clear requirements for the development team.
2. This process guides the development of several important documents: a software requirement specification (SRS), a Use Case document, and a Requirement Traceability Matrix document(RTM).
3. Business analyst and Project Manager set up a meeting with the customer to gather all the information like what the customer wants to build, who will be the end-user, what is the purpose of the product. Before building a product a core understanding or knowledge of the product is very important.

   Example:

   ➔ Suppose a company wants to develop a new e-commerce website. During this phase, the team interviews stakeholders, including marketing, sales, and customers, to understand their needs. They might identify requirements such as user registration, product catalog, shopping cart, and payment processing.

   ➔ Once the requirement gathering is done, an analysis is done to check the feasibility of the development of a product. In case of any ambiguity, a call is set up for further discussion.

   ➔ Once the requirement is clearly understood, the SRS (Software Requirement Specification) document is created. This document should be thoroughly understood by the developers and also should be reviewed by the customer for future reference.

Design:

1. The design phase is where you put pen to paper—so to speak. The original plan and vision are elaborated into a software design document (SDD) that includes the system design, programming language, templates, platform to use, and application security

measures. This is also where you can flowchart how the software responds to user actions.

2. In most cases, the design phase will include the development of a prototype model. Creating a pre-production version of the product can give the team the opportunity to visualize what the product will look like and make changes without having to go through the hassle of rewriting code.

Example: In the e-commerce project, the design phase would result in wireframes and mockups of the website's pages, defining the layout, color schemes, and user interaction patterns. It would also outline the database structure for storing product information and user data.

Development:

1. In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS(Document Data Service) during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.
2. Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code.
3. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Example: During implementation, developers create the website's frontend using HTML, CSS, and JavaScript for the user interface. They also develop the backend using a programming language like Python or PHP to handle user registration, product listing, and payment processing.

Testing:

1. Before getting the software product out the door to the production environment, it’s important to have your quality assurance team perform validation testing to make sure it is functioning properly and does what it’s meant to do. The testing process can also help hash out any major user experience issues and security issues.
2. In some cases, software testing can be done in a simulated environment. Other simpler tests can also be automated.
   The types of testing to do in this phase:

- **Performance testing:** Assesses the software's speed and scalability under different conditions
- **Functional testing:** Verifies that the software meets the requirements
- **Security testing:** Identifies potential vulnerabilities and weaknesses
- **Unit-testing:** Tests individual units or components of the software
- **Usability testing:** Evaluates the software's user interface and overall user experience
- **Acceptance testing:** Also termed end-user testing, beta testing, application testing, or field testing, this is the final testing stage to test if the software product delivers on what it promises

Example: Testers for the e-commerce website would perform various tests, including:

- Unit Testing: Checking individual components like buttons and forms.
- Integration Testing: Ensuring different parts of the website work together, like user registration and login.
- System Testing: Testing the entire system, including ordering products and processing payments.
- User Acceptance Testing (UAT): Inviting actual users to test the website and provide feedback.

Deployment:

1. Once the product is tested and ready to be deployed it is released formally in the appropriate market.
2. Sometimes product deployment happens in stages as per the business strategy of that organization.
3. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Example: After thorough testing, the e-commerce website is deployed to a web server, configured, and connected to a domain name (e.g., www.example.com). Users can now access and use the website for online shopping.

Maintenance:

1. After deployment, the software enters the maintenance phase, where issues are addressed, updates are released, and user support is provided.

Example: In the maintenance phase for the e-commerce website, the team continually monitors the site for performance issues, resolves any bugs reported by users, and periodically adds new features or product updates based on customer feedback.

Advantages Of SDLC :

1. **Structured Approach:-** SDLC provides a structured approach to software development, In addition, it enables developers to plan and organize their work more efficiently. Besides, this approach helps to minimize errors, increase productivity, and ensure the timely delivery of software.
2. **Risk Management:-** SDLC helps to identify and manage risks associated with software development. Further, by identifying potential risks early in the development process, developers can take steps to mitigate them, which reduces the overall risk of software development.
3. **Consistency:-** SDLC ensures consistency in software development by providing a standard framework and methodology. Besides, this consistency helps to improve the quality of the software and ensures that the end product meets the client's expectations.
4. **Collaboration:-** SDLC encourages collaboration among team members by providing a common application framework and language for communication. This collaboration helps to improve the overall quality of the software and ensures that the end product meets the client's requirements.
5. **Cost-Effective:-** SDLC helps to reduce development costs by identifying potential issues early in the development process using prototype software like Figma and others. Furthermore, by identifying issues early, developers can take steps to mitigate them.Which reduces the overall cost of development.

Disadvantages Of SDLC :

1. **Time-Consuming:-** SDLC can be time-consuming, especially if the development process is complex. This can result in delays in the delivery of software, which can be frustrating for clients.
2. **Inflexibility:-** SDLC can be inflexible, especially if the requirements change during the development process. This inflexibility can result in a suboptimal end product that does not meet the client's expectations.
3. **High Upfront Cost:-** SDLC requires a significant upfront investment in terms of time, money, and resources. Moreover, this can be a barrier to entry for small businesses or startups that may not have the resources to invest in SDLC.

Improving Efficiency And Quality In Your SDLC Process :

1. Start with a clear understanding of the project scope and objectives.
2. Develop a detailed project plan and schedule, including all necessary tasks and resources.
3. Involve all stakeholders in the planning process to ensure their buy-in and support.
4. Establish clear communication channels and protocols to keep all team members informed.
5. Adopt an iterative development approach to allow for continuous feedback and improvement.
6. Ensure that testing and quality assurance are integrated into every stage of the SDLC.
7. Use automated tools to streamline repetitive tasks and improve efficiency.
8. Document all processes and decisions to ensure knowledge retention and transfer.
9. Conduct regular reviews and retrospectives to identify areas for improvement.
10. Emphasize collaboration, teamwork, and continuous learning throughout the SDLC process.

Objectives of SDLC:

1. **High-Quality Software:** The primary objective of SDLC is to develop software that meets or exceeds customer expectations in terms of functionality, performance, and reliability. This includes identifying and eliminating defects and bugs through rigorous testing and quality assurance processes.

2. **Timely Delivery:** SDLC aims to ensure that the software is developed and delivered within the specified time frame. This involves setting realistic project schedules, monitoring progress, and making necessary adjustments to meet deadlines.

3. **Cost-Effective Development:** SDLC seeks to manage and control project costs effectively. This includes budgeting, resource allocation, and cost estimation to prevent overruns and minimize wastage of resources.

4. **Customer Satisfaction:**Customer satisfaction is a crucial objective of SDLC. It involves gathering and understanding customer requirements, communicating progress, and involving stakeholders in the development process to ensure the final product aligns with their needs and expectations.

5. **Risk Management:** SDLC aims to identify and mitigate potential risks throughout the software development process. This involves risk assessment, planning for contingencies, and implementing strategies to minimize disruptions.

6. **Maintainability and Scalability:** Developing software that is maintainable and scalable is an important SDLC objective. This ensures that the software can be easily updated, extended, or adapted to changing business needs without significant rework.

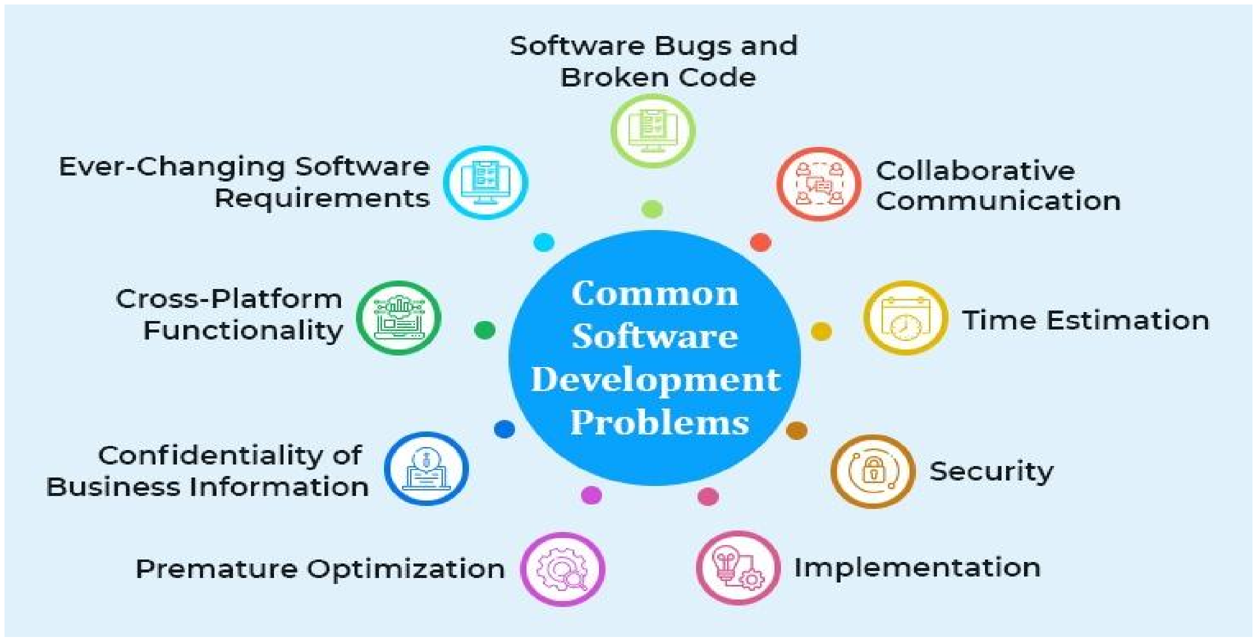
7. **Documentation and Knowledge Transfer:**SDLC emphasizes the creation of comprehensive documentation, including design specifications, user manuals, and code documentation. This helps in knowledge transfer within the development team and facilitates future maintenance and enhancements.

NOTE: These objectives collectively contribute to the successful planning, execution, and delivery of software projects, meeting both business and customer needs. However, it's important to note that the specific objectives may vary based on the project's scope, industry, and organization's goals.

What is software development?

- ➢ Software development refers to a set of computer science activities dedicated to the process of creating, designing, deploying and supporting software.
- ➢ Software itself is the set of instructions or programs that tell a computer what to do. It is independent of hardware and makes computers programmable.
- ➢ There are three basic types:
    - **System software** to provide core functions such as operating systems, disk management, utilities, hardware management and other operational necessities.
    - **Programming software** to give programmers tools such as text editors, compilers, linkers, debuggers and other tools to create code.
    - **Application software** (applications or apps) to help users perform tasks. Office productivity suites, data management software, media players and security programs are examples. Applications also refers to web and mobile applications like those used to shop on Amazon.com, socialize with Facebook or post pictures to Instagram.
- ➢ A possible fourth type is **embedded software.** Embedded systems software is used to control machines and devices not typically considered computers — telecommunications networks, cars, industrial robots and more. These devices, and their software, can be connected as part of the Internet of Things (IoT).
- ➢ Software development is primarily conducted by programmers, software engineers and software developers. These roles interact and overlap, and the dynamics between them vary greatly across development departments and communities.

Common Software Development Problems Faced by Developers:


Software Bugs and Broken Code
Collaborative Communication
Time Estimation
Security
Implementation
Premature Optimization
Confidentiality of Business Information
Cross-Platform Functionality
Ever-Changing Software Requirements
Common Software Development Problems

Ever-Changing Software Requirements:

- ➢ Every house has a foundation, right? Requirement gathering is the foundation of a software project. Unclear requirements mean a weak foundation and subsequent weaker product.
- ➢ Unclear requirements also mean wrong time and cost estimates. Time and cost estimation of a software development project involves a lot of variables that highly differ from project to project based on complexity, the quantum of features and the sophistication of underlying technologies. So, inadequate software requirement gathering equals unrealistic time and cost estimates.
- ➢ Another issue is ever-changing software requirements. So, on day one, you asked for a car with some specific engine capacity, fuel type, power, torque, etc. The next day, you ask to increase the torque. The third day, you say, I need an auto sliding door. See, what happens here - If you keep on adding to the requirements, your scope changes and so does the development. This will lead to a substantial increase in cost as well as development time.

Software Bugs and Broken Code:

- Let’s point out here only: Perfect software is a myth. Bugs and glitches are a part of every software and are inevitable. Our goal here is to reduce their occurrence.
- The quality of the software is what you are working so hard for at the end of the day. Flawed and buggy software can significantly affect your business.
- First, you might have a delay in launch because developers will take some extra time to fix the issues.
- Second, if the issues are discovered after launch, it would mean a huge dissatisfaction of users and might subsequently result in a huge blow to your reputation and might not last long in the marketplace.

Collaborative Communication:

- Irrespective of whether you are hiring some local software vendor or outsourcing software development to some other continent, communication is what allows the software firm and the business stakeholders to collaborate together.
- Poor communication can slow a project down to the point where it’s no longer relevant or valuable. Issues with communication might be lingual or might be not getting regular updates.
- Miscommunication can lead to missed deadlines and feature requests and may incur additional expenses and time or poor products that don’t meet stakeholder expectations.

Time Estimation:

- Everyone says time is of utmost importance and today estimating time is one of the crucial yet common problems especially faced among software developers.
- You see new technologies keep on popping every now and then. And as soon as developers get to know about one aspect of the next thing in line they get in the dilemma of how to use it at the right time. Choose developers on how to make a good estimate.

Security:

- With so many security breaches happening day in and day out, web applications must be fully secured and that’s the duty of your software development team to take care of. So how do you cope with evolving security threats? How do you keep each layer of your software or application secure?

- Simple if everyone sticks to their specific roles and responsibilities. You see, security is not just the responsibility of the software engineer but also the responsibility of all the stakeholders involved including the project management, project managers, business analysts, quality assurance managers, technical architects, and application and developers.
- Also, there are certain best practices that must be considered account:
  - Look beyond new technologies to improve the security of your software.
  - Develop software using high-level programming languages with built-in security features.
  - Require security assurance activities such as penetration testing and code review.
  - Perform essential core activities to produce secure applications and systems.

Implementation:

- Of course, planning and plotting is something I am sure you must be very much focused on but all this works when it's being implemented in the right way. You see software development is not a one time process or has a push-button it can be successfully done by implementing a step-by-step plan.
- Choose a software development team that does know how to ideate as well as implement things in the right way. And even after you have a software solution at hand, just ensure that your chosen development team is right there for maintenance and support.

Premature Optimization:

- Premature optimization mostly occurs when there is a lack of a robust code (it can be described as code that can handle unexpected inputs or conditions gracefully without crashing or producing incorrect results.)reviewal in the whole procedure of software development.
- Whenever it comes to coding, speed is frequently prioritized over cleanness and legitness. Developers frequently fall into the trap of focusing on producing optimized code that runs quickly rather than code that can be read, understood, and updated over time.
- Code that is difficult to read can slow down your team, especially when programmers come and go. You've got a serious problem if a software product developer comes in and can't decipher code.
- Premature optimization is most common when there isn't a robust code review procedure in place. Things get ugly and cause trouble down the line if this crucial step is skipped.

Confidentiality of Business Information:

- ➢ Understanding the confidentiality of business deals with the part where you know which information to share and which shouldn't be shared. It might possibly make you feel overwhelmed.Thus, you need to be extra cautious while dealing with third party companies and sharing your financial information to them
- ➢ But there is a solution to this oversharing of information. In such cases, one should be concerned that their ideas or trade secrets are safe and not stolen or disclosed.
- ➢ For this, signing an NDA is a one-stop solution. A Non-Disclosure Agreement (NDA) is a viable option. A non-disclosure agreement (NDA) is a legal contract between two parties, such as the software provider and you, that outlines the information to be given and requires that it be kept confidential.

Cross-Platform Functionality:

- ➢ Cross-platform applications have become the need of the hour and every business is making its way towards it. Software development companies are expected to provide a unified—or rather, "seamless" experience to users across all platforms, channels, and devices.

Understanding relation between Project, Process and Product:

Product:

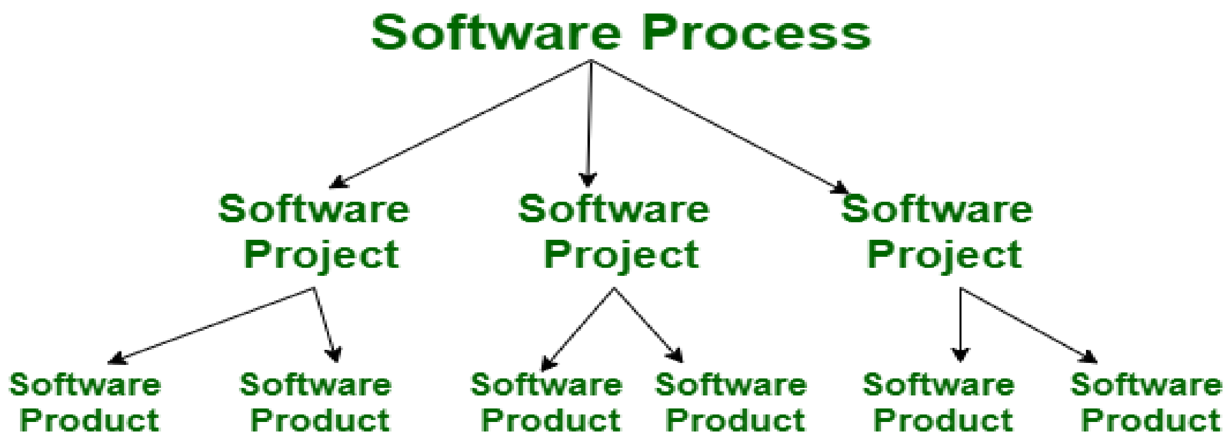
In the context of software engineering, Product includes any software manufactured based on the customer's request. This can be a problem solving software or computer based system. It can also be said that this is the result of a project.

Process:

- ➢ Process is a set of sequence steps that have to be followed to create a project. The main purpose of a process is to improve the quality of the project.

- ➢ The process serves as a template that can be used through the creation of its examples and is used to direct the project.

The main difference between a process and a product is that the process is a set of steps that guide the project to achieve a convenient product. while on the other hand, the product is the result of a project that is manufactured by a wide variety of people.



Difference between Product and Process:-

S. No.	Product	Process
1.	Product is the final production of the project.	While the process is a set of sequence steps that have to be followed to create a project.
2.	A product focuses on the final result.	Whereas the process is focused on completing each step being developed.

3.	In the case of products, the firm guidelines are followed.	In contrast, the process consistently follows guidelines.
4.	A product tends to be short-term.	Whereas the process tends to be long-term.
5.	The main goal of the product is to complete the work successfully.	While the purpose of the process is to make the quality of the project better.
6.	Product is created based on the needs and expectations of the customers.	A process serves as a model for producing various goods in a similar way.
7.	A product layout is a style of layout design in which the materials required to make the product are placed in a single line depending on the order of operations.	When resources with similar processes or functions are grouped together, it is referred to as a process layout.
8.	Product patents are thought to offer a greater level of protection than process patents.	A process patent provides the inventor only limited protection.

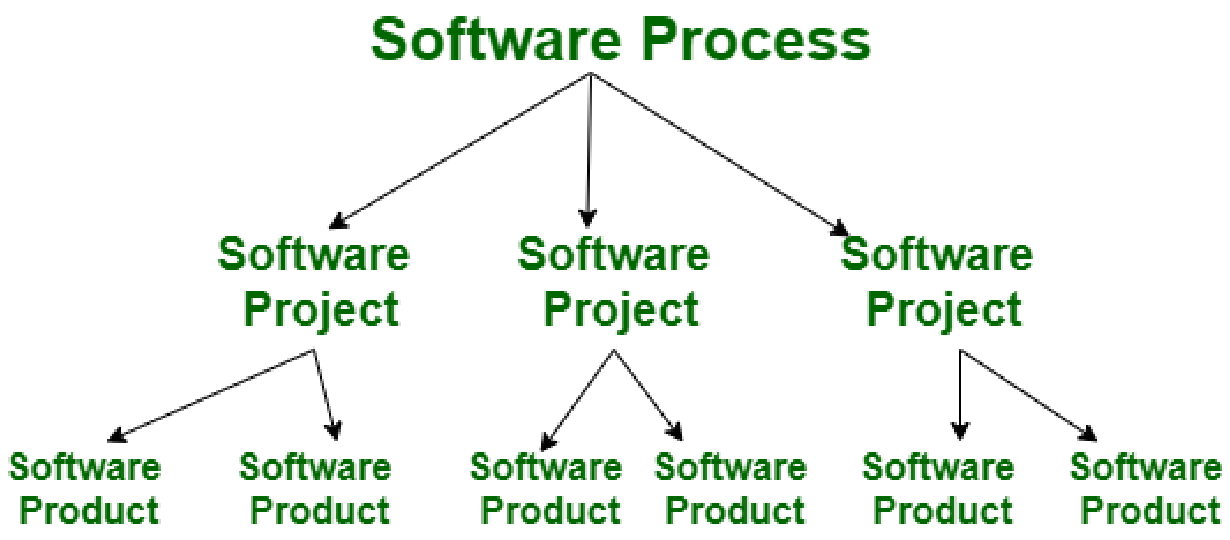
Difference between Project and Product:

Project :

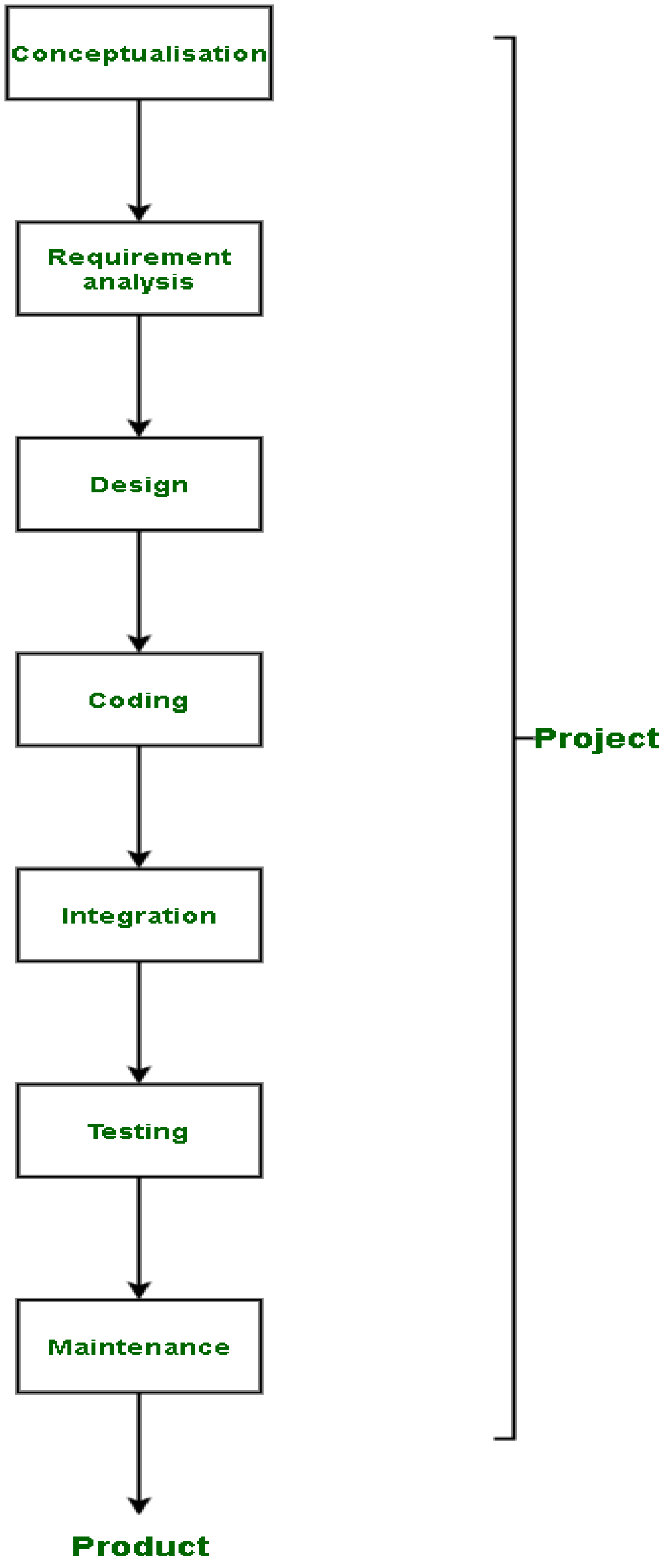
➢ Also known as a software project comprises the steps involved in making a product before it is actually available to the market. The project can be handled by people who are as few as one person to the involvement of a lot of people (over 100). These are usually assigned by an enterprise and are undertaken to form a new product that has not already been made.

Product :

➢ The study of products is a part of Software engineering. The software is built by developers on requests from the customer. After the customer is satisfied with the development process, he launches the software by manufacturing it. This can be a problem-solving software or computer based system. This is the result of a project. The software project, when completed, is called a product after it is available to the market for usage.


Software Process
Software Project
Software Project
Software Project
Software Product
Software Product
Software Product
Software Product
Software Product
Software Product

The product comes into existence after the project is complete.



Difference between Project and Product :

S.NO.	Project	Product
1.	It comprises the steps involved in making a software before it is actually available to the market.	It is the manufacture of the project for users.
2.	The main goal of a project is to form a new product that has not already been made.	The main goal of the product is to complete the work successfully (solve a specific problem).
3.	Project is undertaken to form a new software.	Product is the final production of the project.
4.	It focuses on increasing the performance of the software that is being built.	A product focuses on the final result and the efficiency with which it can solve the given problem.
5.	A project is done only once to get new software.	A product can be made again and again for the purpose of distribution among users.
6.	It is more risky as here, a software is being made for the first time.	It is relatively less risky as the software has already been made and tested. The only risk in most cases would be of wear and tear.
7.	It is handled by the project managers.	It is handled by the product managers.
8.	It exists before the software is made.	It exists after the completion of the software development phases.
9.	It has some cost barriers as the enterprise sets a fixed budget for the project.	It has no cost barrier as it is continued to be developed only as long as the distributing enterprise is profitable.
10.	Project is based on Predictive planning (up-front).	The product is based on adaptive planning (iterative).

Software and Hardware behavior:

Software and hardware are two fundamental components of a computer system, each with its own distinct behavior and functions. Understanding their behavior is essential for grasping how computer systems operate. Let's explore the behavior of both software and hardware:

Software Behavior:

1. **Program Execution:** Software, also known as programs or applications, consists of a set of instructions written in programming languages. When a program is executed, the computer's central processing unit (CPU) interprets and follows these instructions sequentially.
2. **Abstraction:** Software abstracts the underlying hardware, allowing users and developers to interact with the computer system without needing to understand the intricate details of hardware components. Abstraction layers like the operating system provide a user-friendly interface.
3. **Flexibility and Dynamism:** Software behavior is highly flexible and dynamic. Developers can modify, update, or add new software to enhance or change a computer's functionality without physical alterations to the hardware.
4. **Multitasking and Parallelism:** Modern software can leverage the capabilities of multi-core processors and parallel computing to execute multiple tasks concurrently, enhancing efficiency and performance.
5. **Errors and Bugs:** Software can exhibit errors, bugs, and unexpected behaviors due to coding mistakes, compatibility issues, or runtime errors. Testing, debugging, and software updates are necessary to address these issues.
6. **User Interaction:** Software interacts with users through graphical user interfaces (GUIs), command-line interfaces (CLIs), or other input/output mechanisms, allowing users to communicate their intentions to the computer system.
7. **Resource Utilization:** Software manages hardware resources such as memory, storage, and peripherals to ensure efficient and fair resource allocation among running processes or applications.
8. **Security and Vulnerabilities:** Software can be vulnerable to security threats, including viruses, malware, and hacking attempts. Implementing security measures and regular updates are essential to protect software from these threats.

Hardware Behavior:

1. **Physical Operations:** Hardware comprises physical components such as the CPU, memory (RAM), storage devices (e.g., hard drives, SSDs), input/output devices (e.g., keyboard, monitor, mouse), and other peripherals. These components perform physical operations and data manipulation.
2. **Deterministic and Fast:** Hardware behavior is deterministic and operates at very high speeds. The behavior of hardware components is governed by the laws of physics, ensuring consistent and predictable outcomes.
3. **Resource Constraints:** Hardware has finite resources, such as memory capacity and processing power. Software must work within these constraints to perform efficiently.
4. **Concurrency and Parallelism:** Modern hardware is designed to support concurrency and parallelism, enabling multiple tasks to be processed simultaneously. This is crucial for multitasking and improving overall system performance.
5. **Reliability:** Hardware components are designed to be reliable and durable. However, they can still fail due to wear and tear or manufacturing defects. Redundancy and backup systems are often employed to mitigate hardware failures.
6. **Low-Level Interactions:** Hardware interacts directly with electrical signals, binary data, and low-level control mechanisms. This low-level behavior is typically managed by device drivers and firmware.
7. **Energy Consumption:** Hardware consumes energy, and its behavior can be optimized to reduce power consumption in devices like laptops and smartphones.
8. **Physical Constraints:** Hardware behavior is constrained by physical limitations, such as the speed of light for data transmission and the finite size of transistors in integrated circuits.

Overall:

- ➢ Software and hardware have distinct behaviors and characteristics, but they work together in a symbiotic relationship to enable computer systems to function.
- ➢ Software provides the instructions and logic for the hardware to follow, while hardware performs the physical operations required to execute those instructions.
- ➢ Understanding the behavior of both components is essential for effective computer system design, development, and operation.

Bug’s in SDLC:

In the context of Software Development Life Cycle (SDLC), a bug, also known as a software defect or issue, refers to an unexpected and unintended flaw or problem in a software application or system. Bugs can have a wide range of impacts, from minor annoyances to critical errors that can cause the software to malfunction or crash. Here's an overview of bugs in SDLC, their types, and methods for prevention:

Types of Bugs:

1. **Functional Bugs:** These bugs affect the core functionality of the software. They can lead to incorrect calculations, unexpected behavior, or features not working as intended. Examples include buttons not responding, incorrect data processing, or broken links.
2. **Performance Bugs:** Performance bugs impact the software's speed, responsiveness, or resource utilization. Common examples include slow response times, high CPU or memory usage, and inefficient algorithms.
3. **Compatibility Bugs:** Compatibility issues arise when the software does not work as expected on certain hardware, operating systems, web browsers, or other environments. These bugs can result in rendering problems, layout issues, or non-functional features on specific platforms.
4. **Security Vulnerabilities:** Security bugs can lead to security breaches and data vulnerabilities. Examples include buffer overflows, SQL injection, and authentication bypass vulnerabilities.
5. **Usability Bugs:** Usability bugs affect the user experience, making the software difficult to understand or use. Examples include poor user interface design, confusing navigation, and unclear error messages.
6. **Regression Bugs:** Regression bugs occur when a previously working feature or functionality breaks after a software update or code change. These bugs often happen unintentionally when new code changes interact with existing code in unexpected ways.
7. **Concurrency Bugs:** These bugs occur in multi-threaded or parallel processing environments when multiple threads or processes access shared resources simultaneously, leading to race conditions, deadlocks, or data corruption.

Methods for Bug Prevention:

Preventing bugs in software development is crucial for ensuring the quality and reliability of the final product. Here are some methods for bug prevention:

1. **Code Reviews:** Conduct code reviews where developers review each other's code to identify and fix issues early in the development process.

2. **Automated Testing:** Implement automated testing frameworks to perform unit testing, integration testing, and regression testing. Automated tests can help catch bugs before they reach production.
3. **Static Code Analysis:** Use static code analysis tools to scan code for potential issues, such as coding standards violations, security vulnerabilities, and common programming errors.
4. **Requirements Gathering:** Ensure that requirements are well-defined, clear, and unambiguous. Misunderstood or incomplete requirements can lead to bugs during implementation.
5. **Design and Architecture Reviews:** Conduct design and architecture reviews to identify potential issues in the early stages of development.
6. **Coding Standards:** Enforce coding standards and best practices to maintain code consistency and reduce the likelihood of coding errors.
7. **Testing Environment:** Create a dedicated testing environment that closely resembles the production environment to identify and fix environment-specific bugs.
8. **Documentation:** Maintain comprehensive documentation, including design documents, user manuals, and release notes to help users and developers understand the software's functionality.
9. **Change Management:** Implement a robust change management process to track and manage code changes, ensuring that new code does not introduce regressions.
10. **User Feedback:** Encourage user feedback and actively gather information on issues and bugs encountered by end-users.

Overall:
Bug prevention is a proactive approach to improving software quality. By identifying and addressing issues early in the development process, teams can reduce the cost and effort required to fix bugs later in the SDLC, leading to more reliable and user-friendly software.

Impact of a bug:

The impact of a bug in software can vary widely depending on several factors, including the nature of the bug, the context in which the software is used, and the severity of the issue. Bugs can have significant consequences, ranging from minor inconveniences to major disasters. Here are some common impacts of bugs in software:

1. **User Frustration:** Minor bugs or glitches can lead to user frustration. These issues may not affect the core functionality of the software but can create a poor user experience, leading to annoyance and dissatisfaction.
2. **Reduced Productivity:** Bugs can hinder productivity when software is used in business and professional settings. Users may spend extra time troubleshooting problems, finding workarounds, or seeking technical support.
3. **Data Loss or Corruption:** Certain bugs can result in the loss or corruption of user data. This can be particularly detrimental in applications that handle critical information, such as financial software or databases.
4. **Security Vulnerabilities:** Security-related bugs, such as software vulnerabilities or weaknesses, can be exploited by malicious actors. These bugs can lead to unauthorized access, data breaches, or other security incidents.
5. **Financial Loss:** Bugs in software can have financial implications. For businesses, software bugs can lead to lost revenue due to downtime, service interruptions, or the cost of addressing the issue.
6. **Reputation Damage:** Publicly known or widely publicized software bugs can damage an organization's reputation. Users may lose trust in the software or the company that develops it, resulting in a loss of customers or stakeholders.
7. **Legal and Regulatory Consequences:** In some cases, software bugs can result in legal liabilities if they lead to harm or non-compliance with industry regulations. This can lead to legal action, fines, or other penalties.
8. **Operational Disruption:** In critical systems like those used in healthcare, aviation, or industrial control, software bugs can lead to operational disruptions, potentially putting lives and safety at risk.
9. **Delayed Deliverables:** In software development projects, the discovery of significant bugs can lead to project delays as developers must allocate time and resources to identify, fix, and test the issues.

10. **Maintenance Costs:** Bugs often require resources to identify, address, and test fixes. These maintenance costs can be substantial, particularly if the bug is complex or impacts multiple parts of the software.
11. **Customer Churn:** In the case of commercial software, customers dissatisfied with frequent or severe bugs may choose to switch to competing products, resulting in customer churn and revenue loss.
12. **User Abandonment:** In the case of consumer-facing applications or mobile apps, users may abandon or uninstall the software if they encounter persistent bugs or poor performance.

It's crucial for software developers and organizations to prioritize bug identification, testing, and resolution throughout the software development life cycle to mitigate these potential impacts. Quality assurance practices, thorough testing, code reviews, and the use of bug tracking systems are some of the strategies employed to reduce the likelihood and severity of bugs in software. Additionally, rapid response and effective communication with users and stakeholders when bugs are discovered can help minimize the negative consequences.

System support and entropy in sdlc:

In the context of the Software Development Life Cycle (SDLC), "system support" and "entropy" represent two different aspects related to the maintenance and evolution of software systems.

System Support:

System support, often referred to as software system maintenance, is one of the later stages in the SDLC. It involves activities and processes aimed at ensuring the continued functionality, performance, and adaptability of a software system after it has been deployed in a production environment. System support encompasses several key activities:

1. **Bug Fixes:** Addressing and resolving software defects (bugs) that are discovered by users or during routine monitoring.
2. **Enhancements:** Making improvements to the software system to add new features, optimize performance, or enhance user experience based on changing requirements or user feedback.

3. **Updates and Patches:** Providing updates, patches, and security fixes to protect the system from vulnerabilities and security threats.
4. **Compatibility:** Ensuring that the software remains compatible with evolving hardware and software environments, such as new operating systems or browsers.
5. **Data Maintenance:** Managing and maintaining the integrity and availability of data stored by the software system.
6. **Documentation Updates:** Keeping documentation up-to-date to reflect changes in the software's functionality and operation.
7. **User Support:** Providing technical support to users who encounter issues or have questions related to the software.

System support is crucial for the long-term success and sustainability of software systems. It helps extend the system's life cycle, adapt it to changing needs, and ensure its continued reliability and security.

Entropy in SDLC:

What Is Software Entropy?

➢ Software entropy is a term derived from the second law of thermodynamics.

➢ It's used to describe the natural degradation of software systems over time.

➢ Like thermodynamics, software entropy refers to the tendency for systems to become more disordered, complex, and challenging to maintain over time.

➢ The result is that the software becomes more challenging to understand, modify, and support.

➢ Eventually, maintaining it may become so complex that it's no longer usable.

Some factors that lead to software entropy include:

- **Accumulation of technical debt:** As new features are added to a software system, the complexity of the codebase increases. Over time, this can result in technical debt, a measure of the work needed to bring the system up to current standards. Technical debt can make it harder to maintain the system as changes become more complex and time-consuming.

- **Lack of documentation:** Proper documentation is integral to software development and maintenance, but developers often neglect it. As a result, the knowledge of how the system works and how to maintain it becomes fragmented and even lost over time. This neglect can lead to software entropy, as developers must reverse engineer the system to make changes.
- **Inadequate testing:** When software developers don't thoroughly test software systems, they're more prone to bugs and unexpected behavior. Over time, these bugs can accumulate, leading to a system that's difficult to maintain and prone to other errors.
- **Unmaintainable code:** When your code is poorly written or does not follow best practices, it's more likely to become unmaintainable over time, as developers will have a more challenging time modifying and updating sloppy code.
- **Changing requirements:** As software systems evolve, system requirements may change. Complicated systems become more complex and opaque, leading to bad design, clunky chunks of code, and bugs.

Software It Impacts:

➢ Entropy affects all software to some degree. However, the degree of impact can vary depending on factors such as the system's design and architecture, the code's quality, the development processes used, and the rate at which changes are made to the system.

➢ Examples of software systems that are particularly susceptible to software entropy include large, complex systems with many interdependent components, systems maintained over a long period of time by multiple teams, and systems that have undergone significant changes or updates without proper refactoring or cleanup. Software that one person can easily understand is less prone to software entropy.

The Broken Window Theory:

➢ The "broken window theory" in software entropy refers to the idea that minor, ignored software bugs or design issues can accumulate over time and lead to more significant problems and decreased software quality, much like a broken window in a building can lead to vandalism and further neglect.

- ➢ This theory suggests that it's essential to promptly address and fix minor issues to prevent them from becoming bigger problems that can be harder and more expensive to solve.
- ➢ You can apply this theory in software development by implementing regular code reviews, automated testing, and continuous integration and deployment to identify and fix issues early in development.
- ➢ The idea is to maintain a high level of software quality and prevent software entropy, which can make the software difficult to maintain and enhance in the future.

Ways to Mitigate Software Entropy:

1. **Don't Leave Dirty Code Around for Too Long:**
    - ➢ Developers use the term "dirty code" to describe code with poor writing, lacks understandability, poses maintenance challenges, and lacks organization.
    - ➢ It can refer to inefficient code, poorly commented, or written in a way that makes it hard for others — or even its original author — to understand.
    - ➢ Dirty code can also refer to code that violates best practices or common coding standards and may contain bugs, security vulnerabilities, or other issues.
    - ➢ Dirty code is a problem because it can make it harder to add new features, fix bugs, or maintain the code over time.
    - ➢ It can also slow development, increase the risk of bugs and security issues, and make it impossible for others to understand how the code works.
    - ➢ Therefore, writing clean, well-organized, and maintainable code is considered a crucial aspect of software development and is often emphasized in coding best practices and training programs.
    - ➢ The following measures will help reduce dirty code:

Regular Code Reviews:;

- Code reviews are critical to maintaining code quality and preventing software entropy. During these reviews, developers can examine the code for messy, redundant, or outdated sections and clean them up. Regular checks help keep the codebase organized and manageable and identify potential issues before they become unmanageable problems.

Automated Testing:

- Automated testing is an essential tool for mitigating software entropy. By automating tests for different parts of the codebase, developers can catch issues before they spiral out of control. Testing also helps ensure that code changes don't break existing functionality. Automated tests include unit tests, integration tests, and end-to-end tests covering different aspects of code.

Documentation

- Keeping detailed documentation of code changes and updates can help developers track other developers' changes and why. This information can be helpful when making future changes or fixing bugs. Documentation includes comments in the code, inline documentation, design documents, architecture diagrams, and user manuals. Poor documentation can lead to a lack of knowledge about the codebase's design and current state.

Culture of Maintainability

- Encouraging a culture of code maintainability and cleanliness within the development team can help prevent software entropy from taking hold. You can promote this through regular training and education and by setting code quality and maintainability expectations. The development team should be encouraged to prioritize code quality and maintainability, making it a high priority in their workflow.

Adhering to Coding Standards

- Coding standards help developers write code consistently and make code changes that align with the rest of the codebase. Standards can include guidelines for naming conventions, indentation, comments, and rules for how developers should structure code. Adhering to coding standards helps maintain code quality, readability, and consistency.

Refactoring

- Refactoring involves changing the code to improve its quality and maintainability without changing its functionality. These tasks include cleaning up code, removing duplicate code, and enhancing code structure. Regular refactoring can prevent entropy from building over time by ensuring the code remains maintainable and scalable.

Keeping Dependencies Up-to-Date

- Staying up-to-date with the latest versions of libraries and dependencies can help ensure that the code continues to work as expected. Current versions also address security vulnerabilities promptly. In addition, keeping dependencies up-to-date ensures that the codebase uses the most recent and stable version of the necessary libraries and dependencies.

Implementing Continuous Integration and Delivery:

- Continuous integration and delivery (CI/CD) involve automating the build, test, and deployment process. These automated processes include checks to ensure code changes are controlled and consistent, reducing the likelihood of introducing entropy. With CI/CD in place, you can automatically build, test, and deploy code changes with rapid feedback—reducing the risk of introducing bugs into the codebase.

Work With Integrity & Don't Rush Projects:

- Technology is an incredibly fast-paced field. Innovations are constantly hitting the market, and development teams are pressured to deliver results. However, taking your time to do the job right will lead to better outcomes, build trust and credibility, and promote a sustainable pace of work.

Proper Planning & Design

- Taking the time to properly plan and design software before beginning development lets you focus on the big picture. From the beginning, you can implement procedures to promote scalability, maintainability, and performance. Software teams frequently rush to market, but allowing time to plan will pay off in the long run.

Avoid Shortcuts

- Along with poor planning, cutting corners or taking shortcuts during development can lead to messy code, technical debt, and other issues contributing to software entropy. On the other hand, by taking a systematic and deliberate approach, developers can build well-designed, well-architected, and scalable software.

Collaboration

- Diverse teams that collaborate frequently are more likely to include all perspectives of the development process. Frequent collaboration and knowledge-sharing prevent software entropy by avoiding the creation of knowledge silos, which can result in poor code functionality.

Regular Reassessment

- Regularly reassessing and adjusting project priorities helps ensure that the focus remains on delivering high-quality software. In addition, avoiding feature creep, prioritizing bug fixes and performance improvements, and staying focused on the end goal lets developers keep software entropy in check.

Know the Warning Signs

- Despite following best practices, software entropy may creep into your system, particularly if you've added additional features to previous software. The following warning signs may indicate your software is starting to rot.

User Feedback

- Paying attention to feedback from users and stakeholders about the performance of the software can provide early warning signs of software entropy. For example, if users report slow performance, crashes, or other issues, it may be a sign that the software is becoming bloated and unwieldy.

Difficulty Making Changes

- If making changes to the codebase becomes an extended chore, consider examining the codebase for flaws. Changes that require more work than expected can signal that software entropy is taking hold.

Increasing Fail Rate

- Another sign of software entropy is an increased failure rate. You may notice increased downtime, decreased performance, and other issues contributing to software rot.

Bugs

- Increased bugs and issues with the software should raise a red flag for entropy. Bloated systems make it harder to identify and fix bugs, leading to more defects in the code.

Scalability

- If the software can't scale to meet increasing demands, your system may be too complex already. More dependencies and elements bring more coupling, which interferes with scalability.

Increased Technical Debt

- Addressing technical debt as soon as possible is crucial to preventing the effects of software entropy from spiraling out of control. Technical debt is sometimes inevitable, but proactively addressing it lets you keep software entropy in check.

Managing entropy in SDLC involves proactive efforts to control and reduce disorder and complexity as a software system evolves. Strategies for managing entropy include:

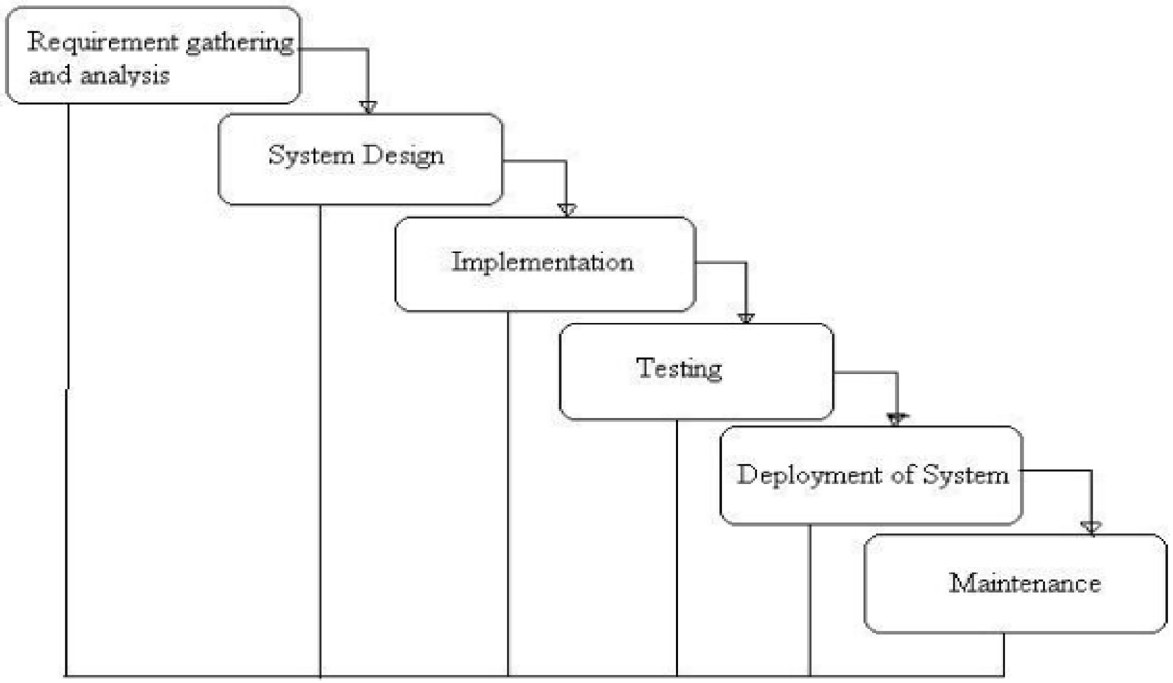
- ➔ Regular code reviews and refactoring to improve code quality.
- ➔ Documentation maintenance and knowledge sharing within the development team.
- ➔ Managing technical debt by periodically addressing and paying it down.
- ➔ Modernizing legacy systems through upgrades or rewrites.
- ➔ Employing good software engineering practices and design principles.

By actively managing entropy, organizations can ensure that their software systems remain maintainable, adaptable, and efficient throughout their life cycles.

Waterfall Model:

- ➢ The Waterfall Model was the first Process Model to be introduced.
- ➢ It is also referred to as a linear-sequential life cycle model.
- ➢ It is very simple to understand and use.
- ➢ In a waterfall model, each phase must be completed fully before the next phase can begin.
- ➢ This type of software development model is basically used for the project which is small and there are no uncertain requirements.

At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project.



Phases of Waterfall Model in Software Engineering:

- ➢ Let us assume that Citibank is planning to have a new banking application developed and they have approached your organization in the 1990's.

Requirements Gathering and Analysis:

- ➢ In this phase the requirements are gathered by the business analyst and they are analyzed by the team. Requirements are documented during this phase and clarifications can be sought.
- ➢ The Business Analysts document the requirement based on their discussion with the customer.
- ➢ Going through the requirements and analyzing them has revealed that the project team needs answers to the following questions which were not covered in the requirements document –
  - Will the new banking application be used in more than one country?
  - Do we have to support multiple languages?
  - How many users are expected to use the application? etc

System Design:

- ➢ The architect and senior members of the team work on the software architecture, high level and low level design for the project.
- ➢ It is decided that the banking application needs to have redundant backup and failover capabilities such that system is accessible at all times.
- ➢ The architect creates the Architecture diagrams and high level / low level design documents.

Implementation:

- ➢ The development team works on coding the project.
- ➢ They take the design documents / artifacts and ensure that their solution follows the design finalized by the architect.
- ➢ Since the application is a banking application and security was a high priority in the application requirements, they implement several security checks, audit logging features in the application.
- ➢ They also perform several other activities like a senior developer reviewing the other developers code for any issues. Some developers perform static analysis of the code.

Testing:

- ➢ The testing team tests the complete application and identifies any defects in the application.
- ➢ These defects are fixed by the developers and the testing team tests the fixes to ensure that the defect is fixed.
- ➢ They also perform regression testing of the application to see if any new defects were introduced.
- ➢ Testers with banking domain knowledge were also hired for the project so that they could test the application based on the domain perspective.
- ➢ Security testing teams were assigned to test the security of the banking application.

Deployment:

- ➢ The team builds and installs the application on the servers which were procured for the banking application.
- ➢ Some of the high level activities include installing the OS on the servers, installing security patches, hardening the servers, installing web servers and application servers, installing the database etc.
- ➢ They also coordinate with network and IT administrative teams etc to finally get the application up and running on the production servers.

Maintenance:

- ➢ During the maintenance phase, the team ensures that the application is running smoothly on the servers without any downtime.
- ➢ Issues that are reported after going live are fixed by the team and tested by the testing team.

Examples of Waterfall Model:

- ➢ In the olden days, Waterfall model was used to develop enterprise applications like Customer Relationship Management (CRM) systems, Human Resource Management Systems (HRMS), Supply Chain Management Systems, Inventory Management Systems, Point of Sales (POS) systems for Retail chains etc.
- ➢ Waterfall model was used significantly in the development of software till the year 2000.
- ➢ Even after the Agile manifesto was published in 2001, the Waterfall model continued to be used by many organizations till the last decade.

When to use SDLC Waterfall Model?

Waterfall Methodology can be used when:

- Requirements are not changing frequently
- Application is not complicated and big
- Project is short
- Requirement is clear
- Environment is stable
- Technology and tools used are not dynamic and is stable
- Resources are available and trained

Advantages and Disadvantages of Waterfall Model:

Here are the popular advantages of Waterfall model in Software Engineering with some disadvantages:

Advantages	**Dis-Advantages**
● Before the next phase of development, each phase must be completed	● Error can be fixed only during the phase
● Suited for smaller projects where requirements are well defined	● It is not desirable for complex project where requirement changes frequently

● They should perform quality assurance test (Verification and Validation) before completing each stage	● Testing period comes quite late in the developmental process
● Elaborate documentation is done at every phase of the software's development cycle	● Documentation occupies a lot of time of developers and testers
● Project is completely dependent on project team with minimum client intervention	● Clients valuable feedback cannot be included with ongoing development phase
● Any changes in software is made during the process of the development	● Small changes or errors that arise in the completed software may cause a lot of problems

Incremental Model:

➢ The incremental model is a software development methodology that can be used to build large and complex systems. It's based on the idea of adding new features, or "increments," to an existing system instead of building the entire thing from scratch at once.

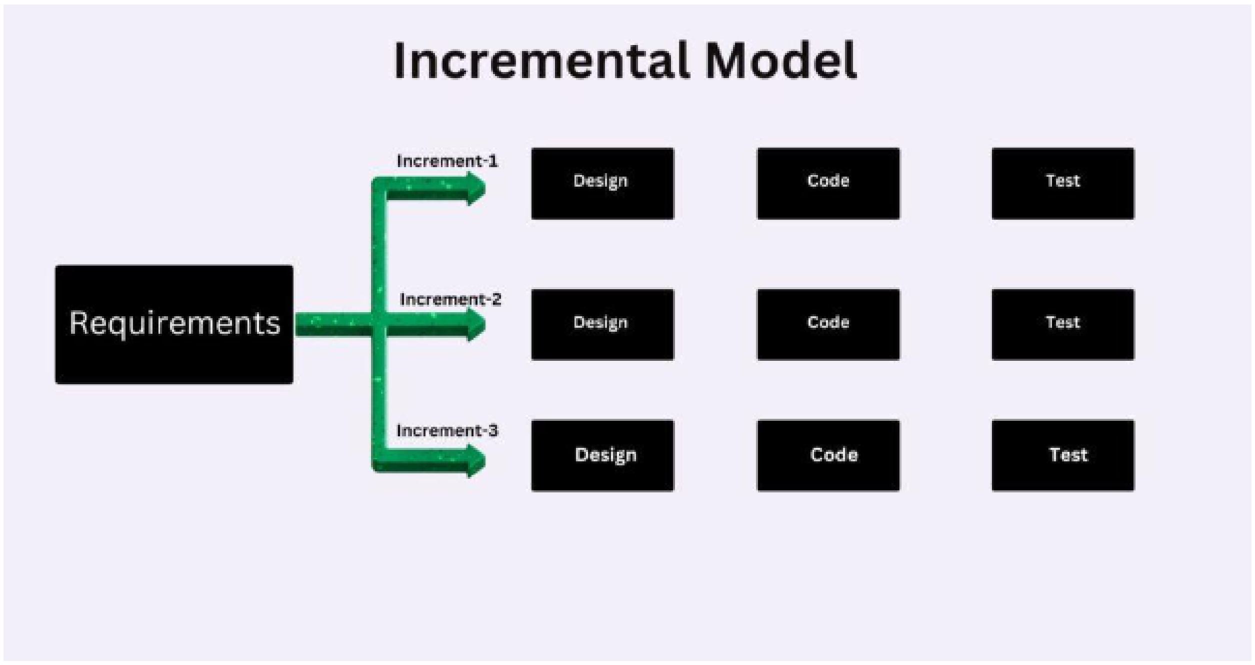
➢ In the incremental model, software requirements are divided or broken down into several stand-alone modules or increments in the SDLC (software development life cycle).

➢ The incremental development is then carried out in phases once the modules have been divided, encompassing all analysis, design, implementation, necessary testing or verification, and maintenance.

➢ Each stage's functionality is built upon the previously produced functionality, and this process is repeated until the software is finished.

➢ The incremental model is one in which the development team tries to finish each incremental build as quickly as possible.

➢ The goal is to deliver a working product bit by bit. The process includes regular releases, with each release representing an increment in functionality and quality.



Types of Incremental Models:

There are two types of incremental models in the software development life cycle:

- **Staged delivery model**—In this kind of incremental model, just one section of the project is built at a time. This allows for the product or service to be developed and delivered in stages, with each stage building on the previous one.
- **Parallel delivery model:** In this kind of incremental model, different subsystems are built concurrently. It can reduce the time required for the development process as long as there are enough resources available.

Characteristics of Incremental Models:

- The project is divided into smaller, incremental pieces that are developed and tested individually. As a result, the software will have multiple small subprojects.
- The final product is not fully developed until the final increment is completed.
- Each requirement/stage is prioritized, and the one with the highest priority is addressed first.
- Once the incremented portion is developed, the requirement of a portion is frozen.
- Each incremental version goes through the phases of analysis, design, coding, and testing. Often, the iterative waterfall model or other methods are used to make each incremental version.

Phases of Incremental Models:

The incremental model is an iterative process that helps identify and correct defects early on in the development process. It has four main phases:

1. **Requirements and analysis—**This phase involves gathering and analyzing the requirements of the project, including the scope, objectives, and constraints. This phase also involves defining the overall project plan and identifying the resources needed to complete the project. Once this document has been created, it goes through a series of reviews and revisions before proceeding to the next phase.
2. **Design—**In this phase, the project team develops a detailed plan for implementing the project, including the design of the system, the selection of appropriate technologies, and the development of any necessary prototypes or proofs of concept.
3. **Coding and implementation—**This phase involves the actual implementation of the project by writing the code according to product requirements. Without using unnecessary hard codes or defaults, the coding standards must be adhered to appropriately to improve and update product quality. This stage also makes it possible to physically execute the designs.
4. **Test—**In this phase, the various components of the system, as well as additional functionality, are checked, integrated, and tested as a whole. After every code implementation, the product should be tested to ensure that it works properly with other pieces of code already written for that project. The system is then deployed to the production environment, where it can be used by end users.

How Incremental Models Work in the SDLC:

- ➢ In the incremental model, the software is divided into different modules that can be developed and delivered incrementally.
- ➢ Since the plan just considers the next increment, there aren't really any long-term plans throughout this period. This makes it possible to alter a version in order to meet users' demands.
- ➢ During the development life cycle, the development team creates the system's fundamental components first before adding new functionalities in successive versions.
- ➢ Once these features are successfully implemented, they're then improved by including new functionality in later editions to increase the functionalities of the product.
- ➢ This process is iterative and follows a waterfall model, with each version building upon the previous one.
- ➢ As the software is developed, each new version is delivered to the customer site for testing and feedback, which is taken into account to improve the version.
- ➢ Once the collection and specification of the requirements are completed, the development process is then separated into different versions. Starting with version 1, each version is developed and then deployed to the client site after each successful increment.

A Comparison: Incremental vs. Waterfall Models:

Some key differences between these models include:

Incremental Model	**Waterfall Model**
● Incurs low cost	● Incurs low cost
● Works on small and large teams	● Works on large teams
● Is simple to make changes to	● Not easy to make changes to
● Early-stage planning is required	● Early-stage planning is required
● Has a low level of risk	● Has a high-level risk
● Has some possibility for reuse	● Chances of reusability are slim
● The client has more influence over the administrator	● The administrator has little influence over the client
● Testing is carried out following each phase iteration	● Testing is done when the development phase is finished
● It's possible to go back to an earlier step or phase	● Not possible to go back to an earlier step or phase
● Operating software requires a short wait time	● Operating software requires a lengthy wait time
● Uses an iterative framework	● Uses a linear framework

Using an Incremental Model:

When the incremental model is used in the SDLC, there are greater possibilities of experiencing lower risks and improved project quality. However, here are some additional benefits of this model:

- **Reduced project risk**—When projects are broken down into smaller chunks, it's easier for teams to understand and manage their risks. Because they're testing each aspect of their project in stages, they're more likely to identify risks earlier in the process, so they can take action to reduce or mitigate them.

- **Improved project quality**—When teams are able to test their project in stages, they can find and correct issues before they have a larger impact. Therefore, they're more likely to produce a high-quality product.
- **Improved project visibility**—The incremental model can help teams to improve their project visibility by providing regular milestones to work with. Using these milestones, they can test their progress, identify issues, and adjust their plans accordingly.
- **Improved team collaboration**—When teams are able to break down their projects into smaller chunks, they can better manage their resources and dependencies, which can help to improve collaboration across teams.

Pros of the Incremental Model:

The incremental model offers many advantages, including the ability to risk-manage progress and increase efficiency. Here are a few other advantages of this approach:

- **Improved efficiency**—The incremental model can help to increase efficiency by allowing teams to plan and organize work more effectively. This approach can also help to reduce the risk of bottlenecks or dependency issues.
- **Risk-managed progress**—Teams can risk-manage progress by testing their project in stages. This means that they can identify and address issues as they emerge and make adjustments as necessary.
- **Clear reporting and tracking**—The incremental model allows teams to clearly report and track their progress, which can help to improve project visibility and collaboration across teams.
- **Feedback is possible**— In the incremental model, the user or the customer can submit feedback at each level to avoid sudden changes.
- **Meeting goals**—Once the requirements are mapped out, all software goals and objectives can be satisfied completely through incremental development.

Cons of Incremental Model:

The incremental model is not suitable for every project and comes with some inherent risks, including:

- This model has to be carefully planned and designed at the beginning, or the entire goal of incrementing will be defeated.
- There's a risk that teams will lose focus and their project will become uncoordinated.
- The iteration stages are rigorous, and they don't overlap.
- If teams don't take time to plan their work and organize their project, they may end up delivering low-quality work and falling behind schedule.
- It can lead to wasted effort, as teams may continually have to renegotiate and reprioritize work, which can lead to inefficiencies.

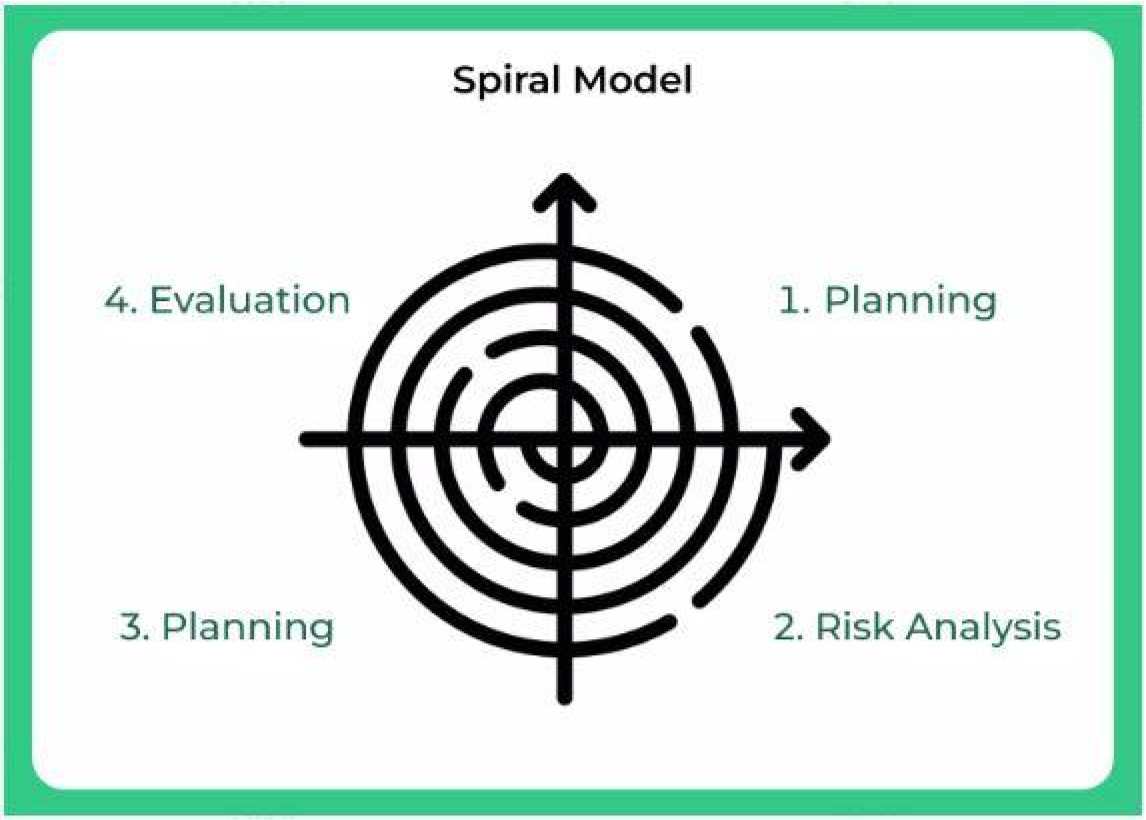
- When there’s a problem in one unit, it has to be fixed in all units, which is time-consuming.

When to Use an Incremental Model:
The incremental model can be used when:

- The objectives of the entire system are clearly stated and recognized, though some details can evolve at each increment over time.
- There’s a pressing need to get a product to market as soon as possible.
- The consumer expects the product to be readily available as soon as possible.
- new technology is being deployed.
- The software team is inexperienced or unskilled.
- A project’s development timeline is prolonged.
- There are some high-risk features and goals.
- There are no resources with the necessary skills.

Spiral Model:

1. Spiral Model is a risk-driven software development process model. It is a combination of waterfall model and iterative model. Spiral Model helps to adopt software development elements of multiple process models for the software project based on unique risk patterns ensuring efficient development process.
2. Each phase of the spiral model in software engineering begins with a design goal and ends with the client reviewing the progress. The spiral model in software engineering was first mentioned by Barry Boehm in his 1986 paper.
3. The development process in the Spiral model in SDLC, starts with a small set of requirements and goes through each development phase for those sets of requirements.
4. The software engineering team adds functionality for the additional requirement in every-increasing spirals until the application is ready for the production phase. The below figure very well explain Spiral Model:



Planning:

- In this phase, the project team defines the goals and objectives of the project, and develops a plan for achieving them. This may involve gathering requirements from stakeholders, identifying risks and developing risk-mitigation strategies, and estimating the resources and budget required for the project.

Risk assessment:

- In this phase, the project team assesses the risks associated with the project and develops strategies to mitigate those risks. This may involve conducting a thorough analysis of the project, identifying potential problem areas, and developing contingency plans to deal with those problems if they arise.

Engineering:

- In this phase, the project team begins the actual development of the software. This may involve writing code, building prototypes, and conducting testing and debugging.

Review and evaluation:

- In this phase, the project team reviews the progress of the project and evaluates the results of the engineering phase. This may involve conducting user acceptance testing, gathering feedback from stakeholders, and making any necessary modifications to the software.

Planning for the next iteration:

- After completing the review and evaluation phase, the project team begins planning for the next iteration of the Spiral model. This may involve revising the project plan, reassessing risks, and making any necessary adjustments to the project.

When to use a Spiral Model?

- A Spiral model in software engineering is used when project is large
- When releases are required to be frequent, spiral methodology is used
- When creation of a prototype is applicable
- When risk and costs evaluation is important
- Spiral methodology is useful for medium to high-risk projects
- When requirements are unclear and complex, Spiral model in SDLC is useful
- When changes may require at any time
- When long term project commitment is not feasible due to changes in economic priorities

Spiral Model Advantages:

1. The spiral model is perfect for projects that are large and complex in nature as continuous prototyping and evaluation help in mitigating any risk.

2. Because of its risk handling ability, the model is best suited for projects which are very critical like software related to the health domain, space exploration, etc.

3. This model supports the client feedback and implementation of change requests (CRs) which is not possible in conventional models like a waterfall.

4. Since customers gets to see a prototype in each phase, there are higher chances of customer satisfaction.

Spiral Model Disadvantages:

1. Because of the prototype development and risk analysis in each phase, it is very expensive and time taking.

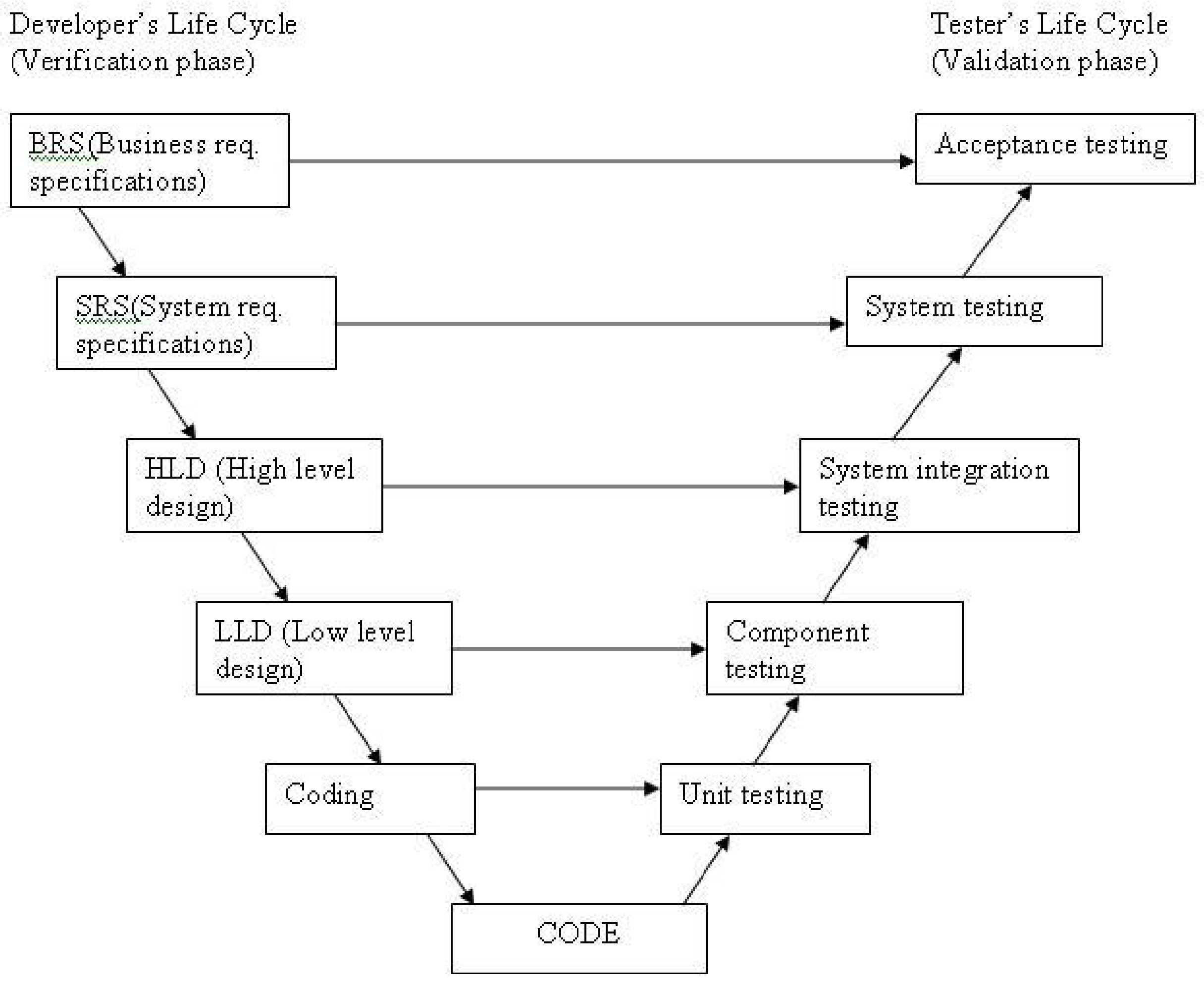
2. It is not suitable for a simpler and smaller project because of multiple phases.

3. It requires more documentation as compared to other models.

4. Project deadlines can be missed since the number of phases is unknown in the beginning and frequent prototyping and risk analysis can make things worse.

'V' Model:

- V- model means Verification and Validation model.
- Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution of processes.
- Each phase must be completed before the next phase begins.
- V-Model is one of the many software development models.
- Testing of the product is planned in parallel with a corresponding phase of development in V-model.


Developer's Life Cycle (Verification phase)
Tester's Life Cycle (Validation phase)
BRS(Business req. specifications)
Acceptance testing
SRS(System req. specifications)
System testing
HLD (High level design)
System integration testing
LLD (Low level design)
Component testing
Coding
Unit testing
CODE

The various phases of the V-model are as follows:

- ➢ Requirements like BRS(Business Requirement Specification) and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.
- ➢ The high-level design (HLD) phase focuses on system architecture and design. It provides an overview of the solution, platform, system, product and service/process. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.
- ➢ The low-level design (LLD) phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. Component tests are created in this phase as well.
- ➢ The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.
- ➢ Coding: This is at the bottom of the V-Shape model. Module design is converted into code by developers. Unit Testing is performed by the developers on the code written by them.

What are V-model- advantages, disadvantages and when to use it?

Advantages of V-model:

- Simple and easy to use.
- Testing activities like planning, test designing happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
- Proactive defect tracking – that is defects are found at an early stage.
- Avoids the downward flow of the defects.
- Works well for small projects where requirements are easily understood.

Disadvantages of V-model:

- Very rigid and least flexible.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- If any changes happen midway, then the test documents along with requirement documents have to be updated.

When to use the V-model:

- The V-shaped model should be used for small to medium sized projects where requirements are clearly defined and fixed.

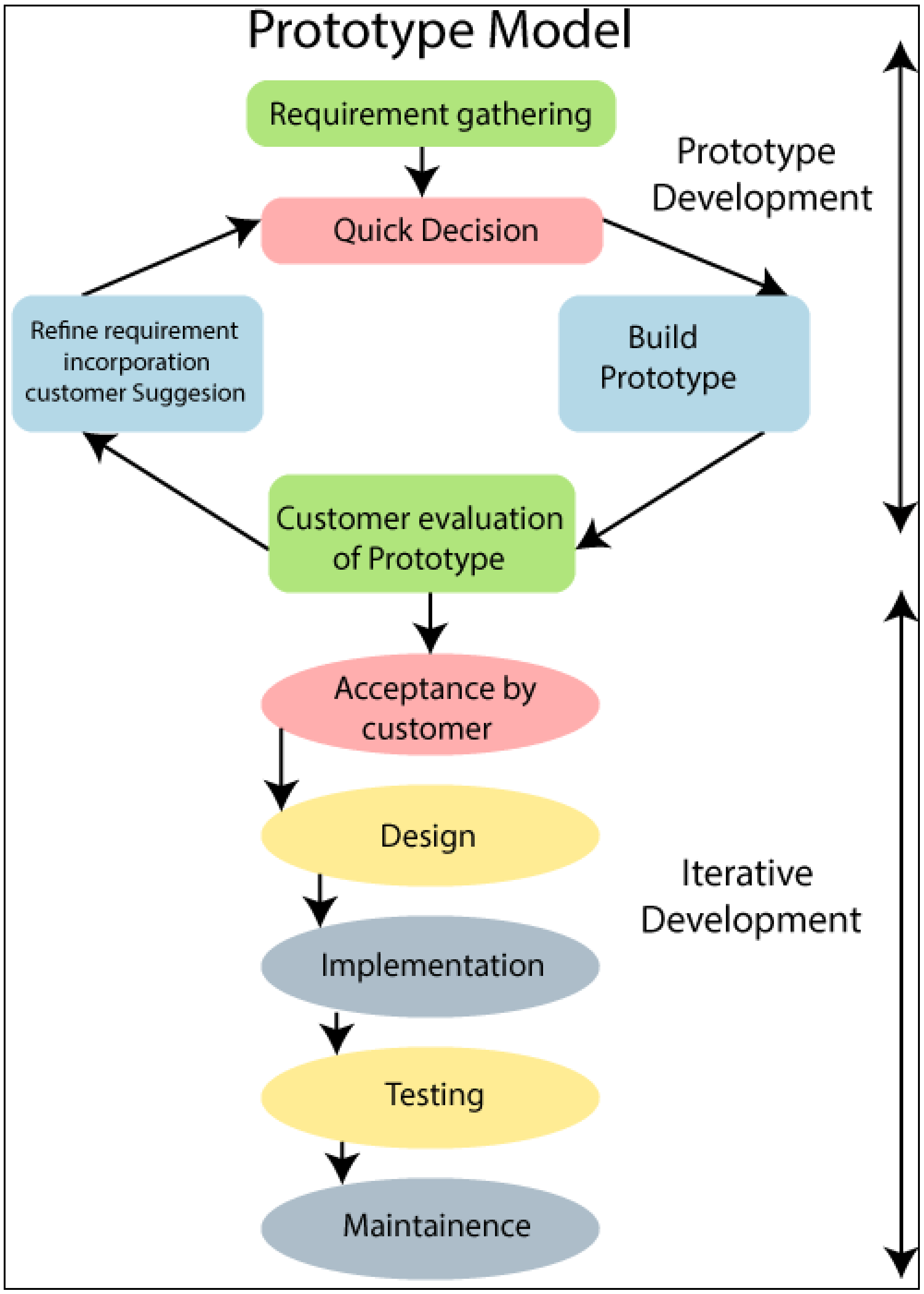
- The V-Shaped model should be chosen when ample technical resources are available with needed technical expertise.
- High confidence of customers is required for choosing the V-Shaped model approach. Since, no prototypes are produced, there is a very high risk involved in meeting customer expectations.

Prototyping Model:

➢ Prototyping Model is a software development model in which a prototype is built, tested, and reworked until an acceptable prototype is achieved.

➢ It also creates a base to produce the final system or software.

➢ It works best in scenarios where the project's requirements are not known in detail. It is an iterative, trial and error method which takes place between developer and client.



- ➢ A step-by-step approach is followed to design a software prototype

Remark:- In this model client satisfaction is guaranteed.

Basic Requirement Identification:

- ➢ This step involves understanding the basic requirements of the product, especially in terms of the user interface. At this stage, it is possible to ignore the more complex details of the internal design and external aspects such as performance and safety.

Developing the initial Prototype:

- ➢ The initial prototype is developed at this stage, where the basic requirements are shown, and user interfaces are provided.
- ➢ These features may not work in exactly the same way internally with the actual software developed. Meanwhile, alternative solutions are used to give the same appearance to the customer in the developed prototype.

Review of the Prototype:

- ➢ The developed prototype is demonstrated to the client and other stakeholders of the project. Comments are collected in an organized manner and used for further improvements in the product under development.

Revise and Enhance the Prototype:

- ➢ Review comments and feedback are discussed during this phase, and some negotiations take place with the client based on factors such as time and budget constraints and technical feasibility of the actual implementation.
- ➢ Accepted changes are incorporated back into the newly developed prototype, and the cycle is repeated until customer expectations are met.
- ➢ Prototypes can have horizontal or vertical dimensions.
- ➢ A horizontal prototype shows the user interface for the product and offers a broader view of the entire system, without focusing on internal functions. On the other hand, a vertical prototype is a comprehensive explanation of a specific function or subsystem in the product.
- ➢ The purpose of the horizontal and vertical prototype is different. Horizontal prototypes are used to obtain more information about the user interface and business requirements. It can also be presented at sales demonstrations to do business in the market.
- ➢ Vertical prototypes are technical and are used to obtain details on the exact operation of the subsystems.
- ➢ For example, the requirements of the database, the interaction, and the data processing are loaded into a given subsystem.
- ➢ The prototyping of the software is used in typical cases, and the decision must be taken very carefully so that the efforts dedicated to the construction of the prototype add considerable value to the final software developed.

Advantage of Prototype Model:

1. Reduce the risk of incorrect user requirement
2. Good where requirement are changing/uncommitted
3. Regular visible process aids management
4. Support early product marketing
5. Reduce Maintenance cost.
6. Errors can be detected much earlier as the system is made side by side.

Disadvantage of Prototype Model:

1. An unstable/badly implemented prototype often becomes the final product.
2. Require extensive customer collaboration
    - Costs customer money
    - Needs committed customer
    - Difficult to finish if customer withdraw
    - May be too customer specific, no broad market
3. Difficult to know how long the project will last.
4. Easy to fall back into the code and fix without proper requirement analysis, design, customer evaluation, and feedback.
5. Prototyping tools are expensive.
6. Special tools & techniques are required to build a prototype.
7. It is a time-consuming process.

Introduction to Software Requirement and Specification

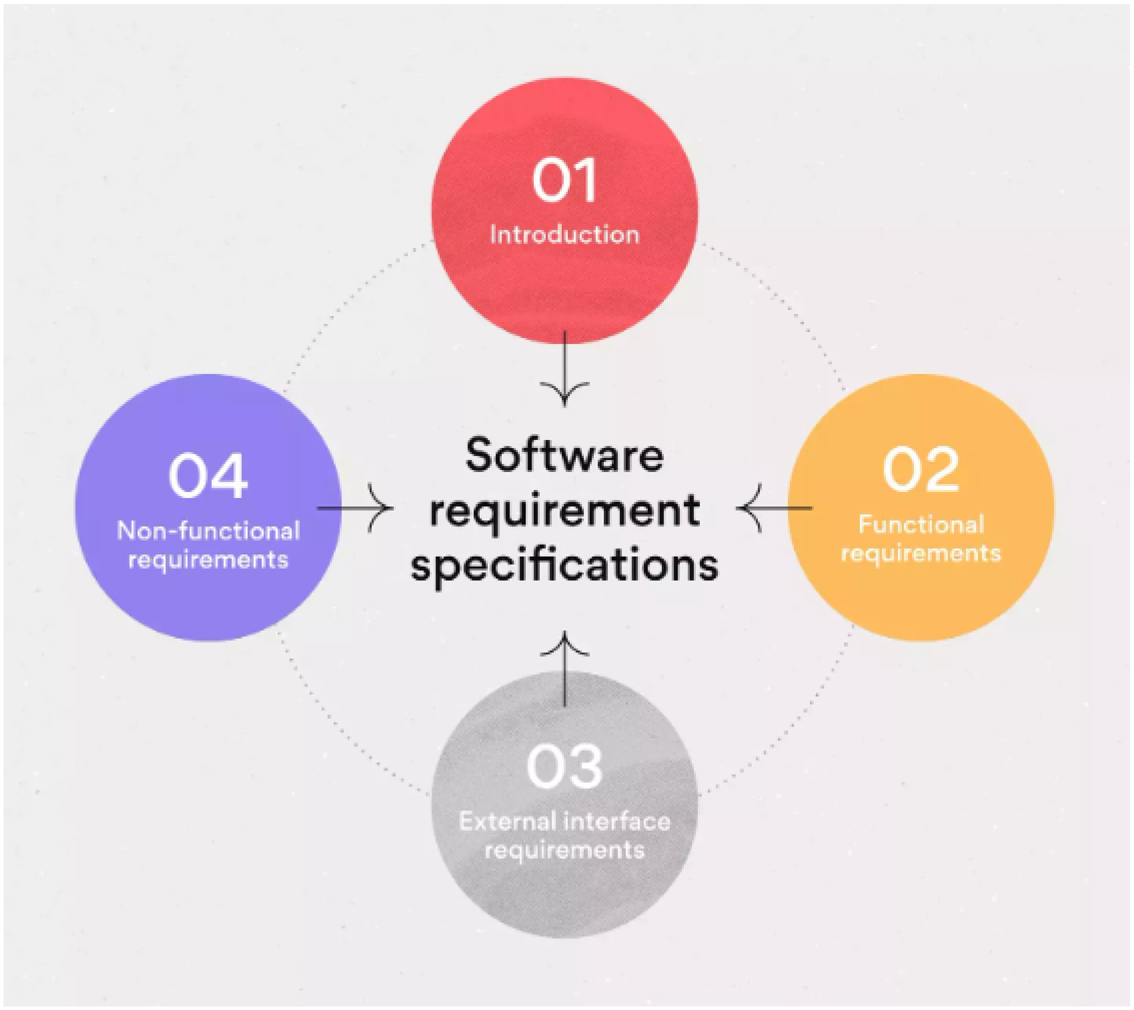
- ➢ A software requirement specifications (SRS) document lists the requirements, expectations, design, and standards for a future project.
- ➢ These include the high-level business requirements dictating the goal of the project, end-user requirements and needs, and the product's functionality in technical terms.
- ➢ To put it simply, an SRS provides a detailed description of how a software product should work and how your development team should make it work.
- ➢ Imagine you have a great idea for an app. You have a vision of what you want it to do and how you want it to look, but you know you can't just give a verbal description to a developer and expect them to match your expectations.
- ➢ This is where an SRS comes in.

Why use an SRS?

- ➢ If developers don’t have clear directions when creating a new product, you may end up spending more time and money than anticipated trying to get the software to match what you had in mind.
- ➢ Composing an SRS document helps you put your idea down on paper and set a clear list of requirements.
- ➢ This document becomes your product’s sole source of truth, so all your teams—from marketing to maintenance—are on the same page.
- ➢ Because software requirement specifications are living documents, they can also act as a communication point between every stakeholder involved in the product development process.
- ➢ Product iterations are bound to occur during any software development project—by noting changes in the SRS, all parties can validate them in the document. This will ease any confusion regarding product requirements.

What to include in an SRS document:

➢ A basic SRS document outline has four parts: an introduction, system and functional requirements, external interface requirements, and non-functional requirements.


01
Introduction
02
Functional requirements
03
External interface requirements
04
Non-functional requirements
Software requirement specifications

Introduction;

➢ An SRS introduction is exactly what you expect—it's a 10,000-foot view of the overall project. When writing your introduction, describe the purpose of the product, the intended audience, and how the audience will use it.

➢ In your introduction, make sure to include:

- **Product scope:** The scope should relate to the overall business goals of the product, which is especially important if multiple teams or contractors will have access to the document. List the benefits, objectives, and goals intended for the product.
- **Product value:** Why is your product important? How will it help your intended audience? What function will it serve, or what problem will it solve? Ask yourself how your audience will find value in the product.

    - **Intended audience:** Describe your ideal audience. They will dictate the look and feel of your product and how you market it.
    - **Intended use:** Imagine how your audience will use your product. List the functions you provide and all the possible ways your audience can use your product depending on their role. It’s also good practice to include use cases to illustrate your vision.
    - **Definitions and acronyms:** Every industry or business has its own unique acronyms or jargon. Lay out the definitions of the terms you are using in your SRS to ensure all parties understand what you’re trying to say.
    - **Table of contents:** A thorough SRS document will likely be very long. Include a table of contents to help all participants find exactly what they’re looking for.
- ➢ Make sure your introduction is clear and concise.
- ➢ Remember that your introduction will be your guide to the rest of the SRS outline, and you want it to be interpreted the same by everyone using the doc.

System requirements and functional requirements:

- ➢ Once you have your introduction, it’s time to get more specific.Functional requirements break down system features and functions that allow your system to perform as intended.
- ➢ Use your overview as a reference to check that your requirements meet the user’s basic needs as you fill in the details.
- ➢ There are thousands of functional requirements to include depending on your product. Some of the most common are:
    - If/then behaviors
    - Data handling logic
    - System workflows
    - Transaction handling
    - Administrative functions
    - Regulatory and compliance needs
    - Performance requirements
    - Details of operations conducted for every screen
- ➢ If this feels like a lot, try taking it one requirement at a time.
- ➢ The more detail you can include in your SRS document, the less troubleshooting you’ll need to do later on.

External interface requirements:

- ➢ External interface requirements are types of functional requirements that ensure the system will communicate properly with external components, such as:
- **User interfaces:** The key to application usability that includes content presentation, application navigation, and user assistance, among other components.

- **Hardware interfaces:** The characteristics of each interface between the software and hardware components of the system, such as supported device types and communication protocols.
- **Software interfaces:** The connections between your product and other software components, including databases, libraries, and operating systems.
- **Communication interfaces:** The requirements for the communication functions your product will use, like emails or embedded forms.

➢ Embedded systems rely on external interface requirements. You should include things like screen layouts, button functions, and a description of how your product depends on other systems.

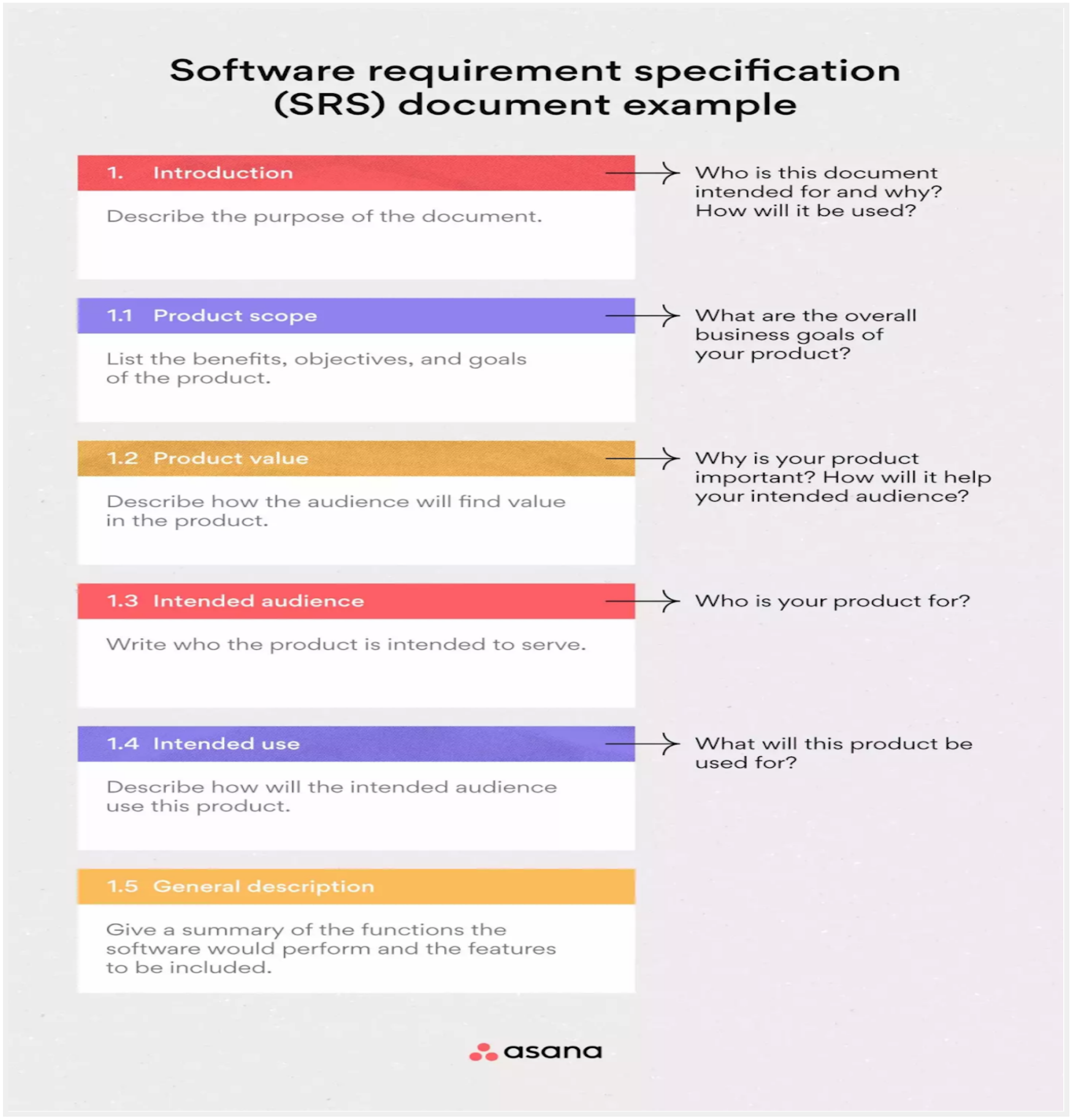
Non-functional requirements (NRFs):

➢ The final section of your SRS details non-functional requirements.
➢ While functional requirements tell a system what to do, non-functional requirements (NFRs) determine how your system will implement these features.
➢ For example, a functional requirement might tell your system to print a packing slip when a customer orders your product.
➢ An NFR will ensure that the packing slip prints on 4x6" white paper, the standard size for packing slips.
➢ While a system can still work if you don't meet NFRs,
➢ you may be putting user or stakeholder expectations at risk.
➢ These requirements keep functional requirements in check, so it still includes attributes like product affordability and ease of use.

The most common types of NFRs are called the 'Itys'. They are:

- **Security:** What's needed to ensure any sensitive information your software collects from users is protected.
- **Capacity:** Your product's current and future storage needs, including a plan for how your system will scale up for increasing volume demands.
- **Compatibility:** The minimum hardware requirements for your software, such as support for operating systems and their versions.
- **Reliability and availability:** How often you expect users to be using your software and what the critical failure time is under normal usage.
- **Scalability:** The highest workloads under which your system will still perform as expected.
- **Maintainability:** How your application should use continuous integration so you can quickly deploy features and bug fixes.
- **Usability:** How easy it is to use the product.

Other common types of non-functional requirements include performance, regulatory, and environmental requirements.



Software requirement specification (SRS) document example

1. Introduction → Who is this document intended for and why? How will it be used?

Describe the purpose of the document.

1.1 Product scope → What are the overall business goals of your product?

List the benefits, objectives, and goals of the product.

1.2 Product value → Why is your product important? How will it help your intended audience?

Describe how the audience will find value in the product.

1.3 Intended audience → Who is your product for?

Write who the product is intended to serve.

1.4 Intended use → What will this product be used for?

Describe how will the intended audience use this product.

1.5 General description

Give a summary of the functions the software would perform and the features to be included.

asana

Requirements Gathering:

- Requirements gathering is the process of understanding what you are trying to build and why you are building it.
- Requirements gathering is often regarded as a part of developing software applications or of cyber-physical systems like aircraft, spacecraft, and automobiles (where specifications cover both software and hardware).
- It can, however, be applied to any product or project, from designing a new sailboat to building a patio deck to remodeling a bathroom.

Three Main Sub Processes of Requirements Gathering:

➢ The key ingredients of the requirements gathering process are three overlapping subprocesses: requirements elicitation, requirements documentation, and requirements understanding.

➢ **Requirements elicitation** is the process of asking for and collecting top-level requirements from all relevant stakeholders. Effort should be made to account for the needs of customers, their users, the internal stakeholders within your own company, and your key suppliers.

➢ **Requirements documentation** organizes the input from the requirements elicitation process into whatever format is appropriate for your organization. This formatting may include:

- User stories
- Functional decompositions (especially for complex cyber-physical systems)
- Feature descriptions

➢ These will be collected in a top-level requirements specification like a product requirements document (PRD) or a system specification.

➢ The purpose of this top-level specification is to make those stories and descriptions available to all members of the project team.

➢ **Requirements confirmation** is the process of making sure all stakeholders and team members have a common understanding of what you're trying to build.

➢ This involves reviewing and refining the requirements. It will very likely require additional elicitation and revision of the documentation as well.

What are the Benefits of Requirements Gathering?

Beyond the obvious advantage of having requirements with which to work, a good requirements gathering process offers the following benefits:

- Greatly improves the chances that customers and users will get what they want. Stakeholders often have difficulty putting into words exactly what it is that they need. You're going to have to help them, and it's going to take some digging.
- Decreases the chances of a failed project. A frequently heard lament following unsuccessful projects is, "The requirements weren't clear."
- Reduces the overall cost of the project by catching requirements problems before development begins. Requirements that are ambiguous or not fully understood often result in costly scrap and rework. Numerous studies have shown that the cost of fixing requirements errors rises exponentially over subsequent phases of development.

Requirements gathering challenges and solutions:

Undocumented processes:

- ➢ In many organizations there is often no or very poor documentation available about existing processes.
- ➢ In this situation, requirements gathering becomes a two-step process.
- ➢ Firstly, back-engineering of existing processes, and then identifying areas for improvement and optimisation.
- ➢ To ensure requirements are full and correct, it's critical to identify key stakeholders and subject matter experts and engage with them directly.
- ➢ This helps eliminate any assumptions and provides a full picture.
- ➢ Drawing business process maps and visualizing workflows are effective techniques that can be used in this situation.

Conflicting requirements:

- ➢ Uncertainty about existing processes or different priorities for different stakeholders, often leads to conflicting requirements.
- ➢ If this is the case, the role of a business analyst is to document all requirements, identify contradictory requests and let stakeholders decide on priorities.
- ➢ As a business analyst you may have some recommendations about what should be prioritized, but it's still important to hear stakeholders' opinion.
- ➢ Setting up a poll can be one of the ways to get clarity about what is important to the majority of stakeholders.

Lack of access to end users:

- ➢ Unavailability of end users may occur due to a few reasons and requires appropriate resolution.
- ➢ Sometimes end users are too busy with their day-to-day work and unwilling to participate in requirements gathering activities.

- In such situations the best a business analyst can do is to minimize the number and length of engagements.
- Doing as much research as possible prior to the engagement will help to make the conversation more structured and insightful.
- It is almost like turning requirements gathering into requirements validation sessions. Defining focus groups and finding the most suitable end-users in each group will also help.
- Another common scenario is when IT or the project team ‘know’ what end-users want and discourage direct communication.
- To convince the team about the necessity of such engagements, try identifying and presenting examples of day-to-day tasks and problems that different users may encounter.
- Spotlight the differences, the environment they are working in, KPIs they have, technology they are using, etc. Such details often uncover unique requirements that could not be assumed or revealed by one group of users.

Focusing on visual aspects rather than on functional:

- Often stakeholders and end-users have a clear picture of how the new solution should look but don’t have an understanding of what it should do.
- The user interface is an important aspect of any system, but it should not define or hinder the functionality.
- Business analysts should make sure there is a clear distinction between design and functional requirements in their documentation.
- To keep focus on functional aspects use more abstract tools such as diagrams, user stories and low-fi prototypes rather than design drafts.

Stakeholder design:

- This is the case when the stakeholders or end-user have the urge to dictate how the system should work rather than providing details about what the system should do
- Listening to stakeholders about potential solutions can be insightful but may also divert from actual problems and better solution designs.
- To prevent such situations, validate each potential ‘false requirement’ by asking ‘why?’, eventually it will reveal the ‘true’ requirement.

Communication problems:

- This category included language barriers, wrong assumptions, unclearly defined vocabulary, and excessive use of professional terminology that can lead to misunderstandings between stakeholders and a business analyst.
- The best strategy to avoid such situation is to communicate often and establish two-way communication.

- ➢ Document gathering requirements and send them for review and feedback to multiple subject matter experts, create and share a glossary of terms, and always verify assumptions.

A 6-Step Requirements Gathering Process
The requirements gathering process consists of six steps. Three of those (steps three through five—the bulk of the process) we've already mentioned as the key subprocesses of requirements gathering. The full six steps are:

- Identify the relevant stakeholders
- Establish project goals and objectives
- Elicit requirements from stakeholders
- Document the requirements
- Confirm the requirements
- Prioritize the requirements

It is important to note that while these steps are typically initiated in the order listed, there is a great deal of overlap and iteration among them. It may, therefore, be better to think of them as sub processes rather than steps as we look at each one individually.

Identify the Relevant Stakeholders:
Find qualified representatives from each relevant stakeholder group. Depending on your project, these may include:

- Customers stakeholders
- Decision-makers
- Users
- System administrators
- Other impacted customer departments
- Internal stakeholders
- Executives
- Engineering
- Marketing
- Sales
- Customer support (including on-site maintenance teams, if applicable)
- Key suppliers
- Distributors and other partners

Remember to search for "hidden stakeholders." Ask probing questions in early meetings, before you begin eliciting requirements. Identify all stakeholder groups who need to be involved. Ideally, you want input from every group who has skin in the game; give them all the opportunity to state their needs.

Establish Project Goals and Objectives:

- ➢ What are you trying to achieve with this new product or project? What are the overall outcomes your customers want from the product? What are your company's business goals? What are the actionable, measurable objectives you need to achieve to realize those goals and outcomes?
- ➢ Write them down. State them clearly, and get all your stakeholders to sign-off on them. Your goals and objectives will act as a framework for your decision-making.
- ➢ Each requirement you write should help satisfy a project objective and accomplish a goal. If it doesn't, you should either discard it or make it a candidate for a future release.

Elicit Requirements From Your Stakeholders:

- ➢ This is the first of three requirements gathering subprocesses which are highly iterative and overlapping.
- ➢ Even in an agile environment, you are likely to go through several cycles of elicitation, documentation, and review/confirmation before you achieve a workable specification to begin development.
- ➢ Elicitation can be performed in several ways.
- ➢ Time-tested techniques include surveys, questionnaires, and interviews. Interviews and follow-up meetings will be prevalent during later iterations.
- ➢ Be sure to listen actively during interviews. Ask probing questions and take copious notes. Afterward, organize your notes and follow up as necessary. Document each exercise or encounter thoroughly.

Document the Requirements:

- ➢ As soon as requirements begin to emerge from your elicitation process, start documenting them.
- ➢ Write them down and collect them in whatever format your organization has agreed upon.
- ➢ That could be a product requirements document (PRD) of your company's design, a government-mandated system requirements specification, a vendor-supplied requirements management (RM) tool like Jama Connect, a spreadsheet, a database, or any other appropriate repository your entire team can access.

What's most important is the requirements documentation…

- Can be easily navigated and understood by your team
- Is available for review by all stakeholders
- Provides a facility for traceability to other documentation.

Templates are extremely useful, both for the specification as a whole and for individual requirements. Solid, battle-tested templates and standardized formats help provide clarity and aid navigation.

Confirm the Requirements:

- ➢ Review the requirements with all stakeholders. Make sure the requirements clearly capture what was intended and that all parties have a common understanding of each of them. If you find any ambiguity in a requirement, revise it.
- ➢ You should also validate your requirements through prototyping and testing, where possible. Modern prototyping tools make it fast and easy to create a working model of your specification.
- ➢ You can then use that model to perform feasibility, usability, and product concept testing.
- ➢ Get stakeholder sign-off on individual requirements as you get them nailed down. Do the same for the specification as a whole during a final review.

Prioritize the Requirements

- ➢ Most engineering development programs run into unexpected challenges along the way. Unanticipated obstacles are encountered. Schedules slip. Priorities change. It's important to be able to adapt to those challenges and changes.
- ➢ That's why it is crucial to prioritize your requirements based on how each will impact your goals and objectives for your release.
- ➢ Many product managers prioritize features by tagging them with labels, such as "must have," "high want," and "nice to have."
- ➢ But it's also important to rank order each requirement within those categories. There are two reasons for this:-
    - ○ The first is time to market. Schedules often slip. When they do, you may need to trim features and requirements to meet your release date. You don't want your team implementing the easiest requirements first only to find that there's not enough time to complete all your must-haves.
    - ○ The second reason is that requirements evolve. During implementation, you're likely to discover new needs. Some may be critically important and supersede existing requirements in terms of priority.

You need to know where those new requirements fit in your pecking order. If you don't, less important factors will determine what gets implemented first, which may have an adverse impact on your product's success.

What is the difference between software requirements and software specifications?

Understanding Software Requirements:

- ➢ Software requirements are a vital aspect of software development.
- ➢ They represent the needs, expectations, and constraints that a software system must fulfill to satisfy its stakeholders.
- ➢ Software requirements define the functionalities, features, and qualities that the software application should possess.
- ➢ These requirements are typically expressed in natural language, supplemented by diagrams, use cases, and user stories.
- ➢ Software requirements capture the "what" rather than the "how" of a software system. They focus on the high-level objectives and functionality, describing the intended behavior of the software from a user's perspective.
- ➢ Requirements are typically gathered through interactions with stakeholders, such as interviews, surveys, and workshops.
- ➢ They serve as a foundation for software development, guiding the design, implementation, and testing phases.

Key characteristics of software requirements include:

1. User-centric: Software requirements reflect the needs and expectations of the end-users and stakeholders of the software system.
2. High-level: Requirements provide a broad overview of the desired functionalities, features, and qualities of the software without delving into implementation details.
3. Subject to change: Requirements can evolve throughout the software development life cycle as stakeholders gain more clarity or encounter new insights or challenges.

Understanding Software Specifications:

Software specifications, on the other hand, delve into the detailed design and technical aspects of the software system. They bridge the gap between requirements and implementation, providing a clear blueprint for developers to follow. Specifications focus on the "how" of the software system, defining the technical details, architecture, interfaces, algorithms, and data structures required to realize the software requirements.

Software specifications often include:

1. Architectural Design: Specifications describe the overall system architecture, including the arrangement of modules, subsystems, and their interactions.
2. Detailed Functionality: Specifications outline the specific functions, operations, and behavior of the software system, often expressed through pseudo-code or flowcharts.
3. Data Models: Specifications define the structure and organization of data within the system, including data schemas, relationships, and constraints.
4. Interfaces: Specifications outline the interfaces between different system components, modules, and external systems, defining how they interact and exchange data.

5. Performance Requirements: Specifications specify the performance characteristics of the software, such as response times, throughput, and resource utilization.
6. Security and Quality Considerations: Specifications address security measures, error handling, exception handling, and any quality standards or guidelines that need to be followed.

Key characteristics of software specifications include:

1. Technical Details: Specifications provide a comprehensive and detailed description of the software system, addressing design choices, algorithms, data structures, and technical constraints.
2. Implementation-oriented: Specifications serve as a guide for developers, providing them with the necessary information to build the software system accurately.
3. Stability: Unlike requirements, specifications tend to be less prone to change once they are defined, as they form the foundation for development.

Distinguishing Between Software Requirements and Software Specifications:

While software requirements and software specifications share a common goal of guiding software development, they differ in their focus and level of detail:

1. **Focus:** Requirements concentrate on the desired functionalities and qualities of the software system from a user's perspective, while specifications dive into the technical aspects and implementation details.
2. **Level of Detail:** Requirements provide a high-level overview of the software system, allowing for flexibility and adaptability. Specifications, on the other hand, offer granular details and serve as a blueprint for developers to follow during implementation.
3. **Stakeholder Perspective:** Requirements capture the needs and expectations of stakeholders, including end-users, clients, and business owners. Specifications address the technical concerns and considerations of developers and system architects.
4. **Flexibility:** Requirements are subject to change as stakeholders gain more clarity or new insights arise. Specifications, once defined, tend to be more stable and provide a solid foundation for implementation.

Software requirements and software specifications play distinct roles in the software development process. Requirements focus on capturing the needs and expectations of stakeholders, while specifications provide detailed technical guidance for developers. Understanding the difference between these two terms is crucial for effective communication, collaboration, and successful software development. By properly defining and managing requirements and specifications, software engineers can build software systems that meet stakeholder expectations, adhere to technical standards, and deliver high-quality solutions.

Case Study 1:

Case Study: Blinkit App

Background:
"Blinkit" is a startup company aiming to provide a food delivery and grocery shopping platform. The company wants to develop a mobile application that allows users to order food from restaurants and purchase groceries from local stores, all in one app. The Blinkit app aims to offer convenience and a wide range of options to its users.

Key Stakeholders:
1. Blinkit Management Team
2. Software Development Team
3. Restaurant Partners
4. Grocery Store Partners
5. Delivery Drivers
6. Customers

Phase 1: Requirement Gathering

Objective:
Define the functional and non-functional requirements for the Blinkit app.

Functional Requirements:

1.User Registration and Authentication:
   - Users should be able to create accounts.
   - Users must log in to access personalized features.
   - Implement social media login options.

2. Food Ordering:

- Users can browse a list of restaurants.
- View restaurant menus with item details, prices, and images.
- Add food items to the cart.
- Specify customizations and special instructions for food items.
- Review and confirm orders.
- Track the delivery status in real-time.

3. Grocery Shopping:
   - Display a list of local grocery stores.
   - Provide product categories and search functionality.
   - Allow users to add grocery items to their cart.
   - Specify quantity and preferred brands.
   - Review and finalize grocery orders.

4. Payment Integration:
   - Support various payment methods (credit card, digital wallets, cash on delivery).
   - Implement secure payment processing.

5. Delivery Management:
   - Assign delivery drivers to orders.
   - Display estimated delivery times to customers.
   - Provide delivery drivers with navigation tools.
   - Allow customers to rate and review delivery drivers.

6. User Reviews and Ratings:
   - Enable users to rate and review restaurants, grocery stores, and products.
   - Display average ratings and reviews on relevant pages.

Non-Functional Requirements:

1. **Performance:**
   - Ensure fast loading times for the app and menu pages.
   - The app should be responsive even during peak usage.

2. **Security:**
   - Implement robust data encryption for user and payment information.
   - Regularly update and patch security vulnerabilities.

3. **Scalability:**
   - The system should be able to handle increased user traffic and restaurant/grocery store partnerships.

4. **Usability:**
   - Design an intuitive and user-friendly interface.
   - Conduct usability testing to gather feedback for improvements.

5. **Reliability:**
   - Minimize app downtime and ensure reliable order processing.

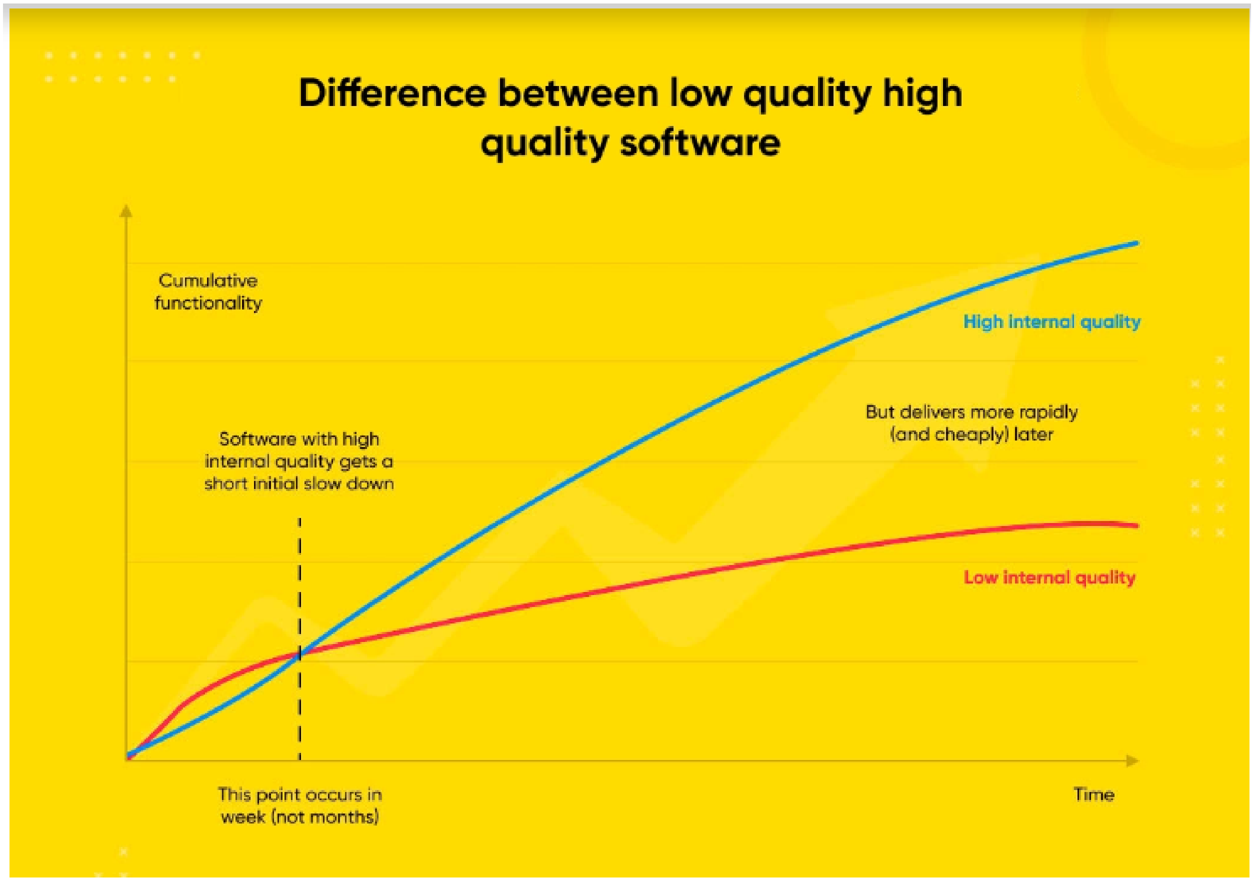
6. **Compliance:**
   - Comply with local food safety regulations and data protection laws.

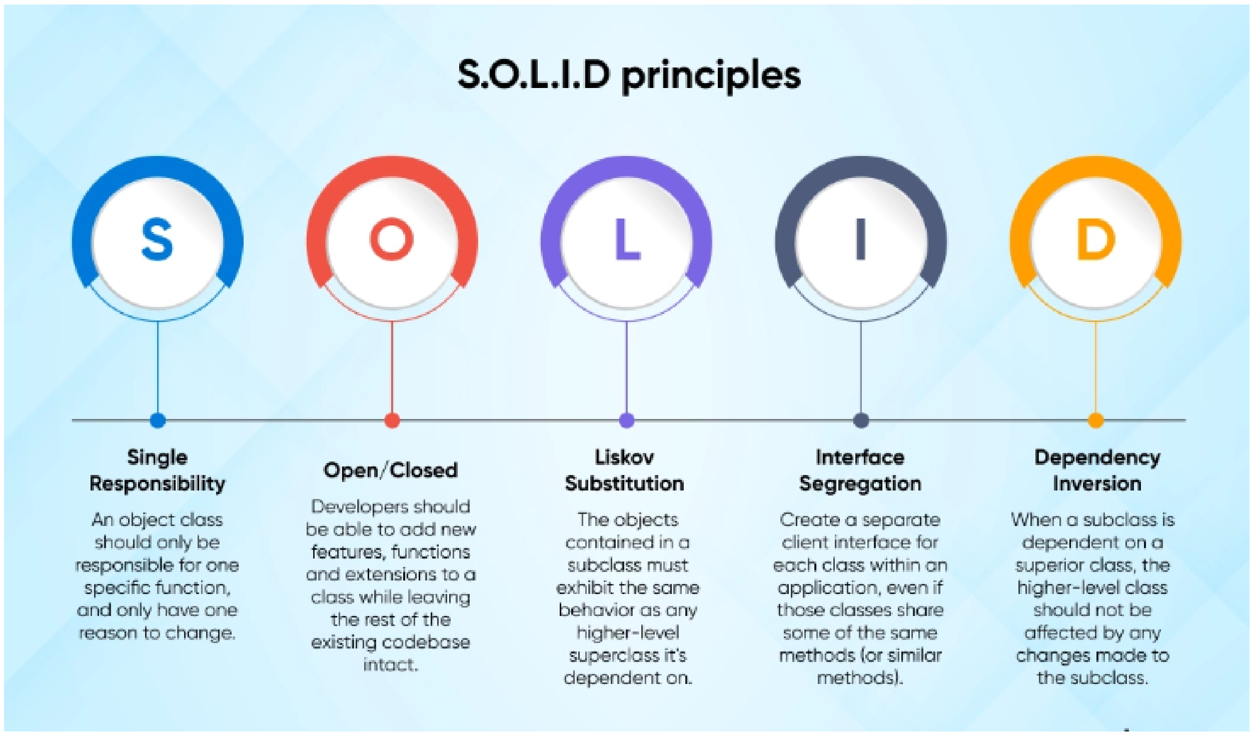
7. **Accessibility:**
   - Ensure the app is accessible to users with disabilities.

This case study outlines the initial software requirements and specifications for the Blinkit app. Further detailed documentation and iterative development will be necessary to bring the application to fruition.

Case Study 2:

Hospital manegnent





S.O.L.I.D principles

Single Responsibility
An object class should only be responsible for one specific function, and only have one reason to change.

Open/Closed
Developers should be able to add new features, functions and extensions to a class while leaving the rest of the existing codebase intact.

Liskov Substitution
The objects contained in a subclass must exhibit the same behavior as any higher-level superclass it's dependent on.

Interface Segregation
Create a separate client interface for each class within an application, even if those classes share some of the same methods (or similar methods).

Dependency Inversion
When a subclass is dependent on a superior class, the higher-level class should not be affected by any changes made to the subclass.

USE CASE

Simple Laundry Use Case

Use Case 1	Do laundry
Actor	Housekeeper
Basic Flow	On Wednesdays, the housekeeper reports to the laundry room. She sorts the laundry that is there. Then she washes each load. She dries each load. She folds the items that need folding. She irons and hangs the items that are wrinkled. She throws away any laundry item that is irrevocably shrunken, soiled or scorched.

Middleweight Laundry Use Case
Basic Flow

Use Case 1	Do laundry
Actor	Housekeeper
On Wednesdays, the housekeeper reports to the laundry room. She sorts the laundry that is there. Then she washes each load. She dries each load. She folds the items that need folding. She throws away any laundry item that is irrevocably shrunken, soiled or scorched.	

Alternative Flow 1	If she notices that something is wrinkled, she irons it and then hangs it on a hanger.
Alternative Flow 2	If she notices that something is still dirty, she rewashes it.
Alternative Flow 3	If she notices that something shrank, she throws it out.

Heavyweight Laundry Use Case:

Use Case 1	Housekeeper does laundry
Actor	Housekeeper
Use Case Overview	It is Wednesday and there is laundry in the laundry room. The housekeeper sorts it, then proceeds to launder each load. She folds the dry laundry as she removes it from the dryer. She irons those items that need ironing.
Subject Area	Domestics
Actor(s)	The housekeeper

Trigger	Dirty laundry is transported to the laundry room on Wednesday.
Precondition 1	It is Wednesday
Precondition 2	There is laundry in the laundry room.

Basic Flow: Do Laundry

Description	This scenario describes the situation where only sorting, washing and folding are required. This is the main success scenario.
1	Housekeeper sorts laundry items.
2	Housekeeper washes each load.
3	Housekeeper dries each load.
4	Housekeeper verifies that the laundry item does not need ironing, is clean and not shrunken.
5	Housekeeper verifies that the laundry item is foldable.
6	Housekeeper folds laundry item

7	Housekeeper does this until there are no more laundry items to fold
Termination outcome	Laundry is clean and folded

Alternative Flow 4A: Laundry item needs ironing.

Description	This scenario describes the situation where one or more items need ironing before or in lieu of folding
4A1	Housekeeper verifies that the laundry item needs ironing and is clean and not shrunken
4A2	Housekeeper irons the laundry item
4A3	Housekeeper puts laundry item on a hanger
Termination outcome	Laundry that needs ironing is ironed and hung up.

Alternative flow 4B: Laundry item is dirty.

Description	This scenario describes the situation where the laundry item did not get clean the first time through the wash.
4B1	Housekeeper verifies that the laundry item is not clean.
4B2	Housekeeper rewashes the laundry item

Termination outcome	Dirty laundry is rewashed.

Alternative flow 4C: Laundry item shrank.

Description	This scenario describes the situation where the laundry item shrank.
4C1	Housekeeper verifies that the laundry item shrank
4C2	Housekeeper disposes of laundry item.
Termination outcome	Laundry item no longer exists.

Alternative flow 5A: Laundry item needs hanger.

Description	This scenario describes the situation where the laundry item needs to be hung instead of folded.
5A1	Housekeeper verifies that the laundry item needs hanging.
5A2	Housekeeper puts laundry items on a hanger.
Termination outcome	Laundry that needs hanging is hung up.

Post conditions: All laundry clean and folded or hung up.

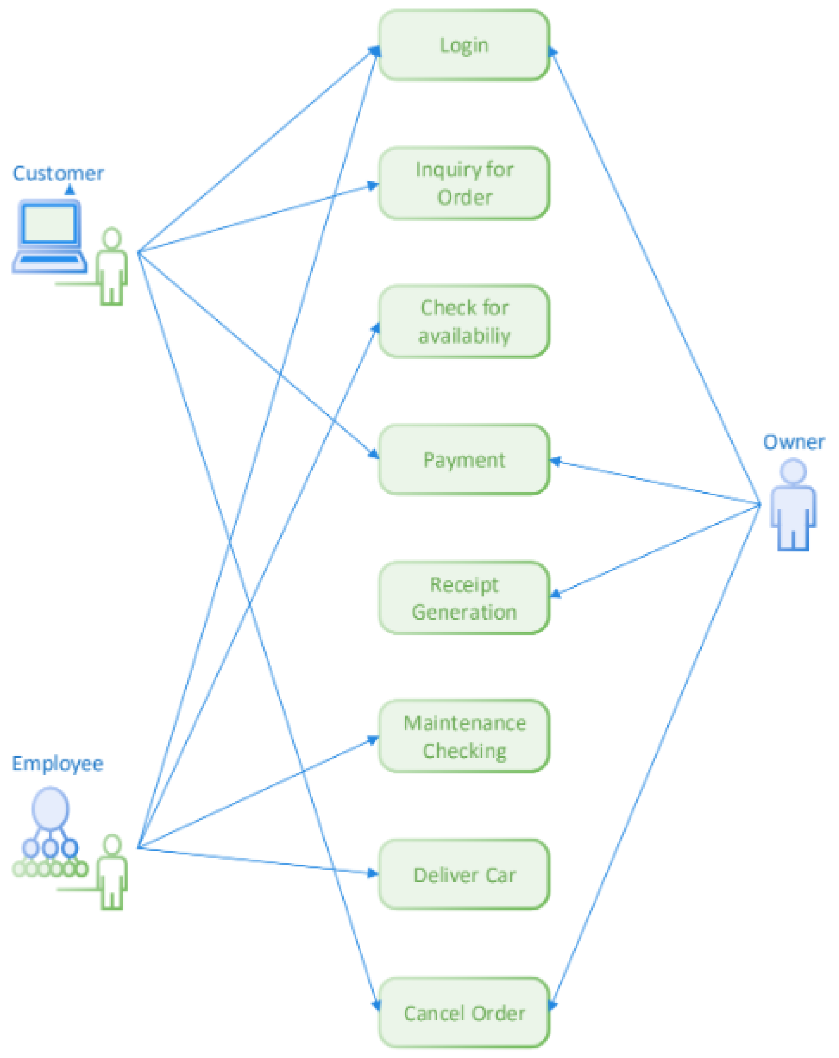


Figure 7-1: Use-case Diagram