❖ *Assignment*

*1. <u>What is SDLC ?</u>

SDLC stands for Software Development Life Cycle.

It's a structured process used by software developers and teams to design, develop, test and deploy software efficiently and with high quality.

Stages of the SDLC

- **Planning**
  - Define goals, scope, resources, timeline
  - Feasibility studies and project planning.
- **Requirements Gathering & Analysis**
  - Understand what the users need.
  - Document functional and non-functional requirements.
- **Design**
  - Architecture and UI/UX design.
  - Choose tech stack, create data models, etc.
- **Development**
  - Actual coding of the software based on the design.
  - Front-end, back-end, databases, APIs, etc.
- **Testing**
  - Check for bugs, performance issues, security problems.
  - Includes unit testing, integration testing, system testing.
- **Deployment**
  - Release the product to a live environment.
  - Might be a full launch or a phased rollout.
- **Maintenance**
  - Ongoing updates, bug fixes, and improvements post-launch.
  - Could also include support and documentation updates.

- **SDLC Models**

- **Waterfall** – Linear and sequential.
- **Agile** – Iterative and flexible.
- **Spiral** – Risk-driven, combines iterative nature with systematic aspects.
- **DevOps** – Focuses on integration between development and operations.

*2. What is software testing?

Software testing is the process of evaluating and verifying that a software application or system does what it's supposed to do. The goal is to find bugs, errors, or gaps between actual and expected results, and to ensure the software is reliable, secure, and performs well under various conditions.

Types of software testing:

1. **Manual Testing**
   Performed by humans without the help of automation tools.
2. **Automated Testing**
   Uses tools/scripts to execute tests automatically (e.g., Selenium, JUnit).
3. **Functional Testing**
   Tests what the system **does** (e.g., can users log in?).
4. **Non-Functional Testing**
   Tests how the system behaves (e.g., speed, usability, security).
5. **Common levels of testing**
   - **Unit Testing:** Smallest testable parts (usually by developers).
   - **Integration Testing :** How components work together.
   - **System Testing :** The whole system is tested.

- Software Testing is a part of software development process.
- Software Testing is an activity to detect and identify the defects in the software.
- The objective of testing is to release quality product to the client**.**

*3. What is agile methodology?

Agile methodology is a way of managing projects — especially in software development — that focuses on **flexibility, collaboration, and customer feedback** rather than following a strict, step-by-step plan.

Instead of trying to build the entire product at once, Agile teams work in small, repeatable cycles called **iterations** or **sprints** (usually 1–4 weeks long). After each sprint, they deliver a working piece of the product, review it with stakeholders, and adjust their plans based on feedback. The idea is to stay adaptable and continuously improve the product as you go.

Some key principles of Agile:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

Popular Agile frameworks include **Scrum**, **Kanban**, and **Extreme Programming (XP)**.

point of Agile methodology

- **Being flexible**: Plans can change easily if new information or feedback comes in.

- **Delivering value early and often**: Instead of waiting months or years, customers start getting usable features quickly.

- **Improving continuously**: After each sprint, teams reflect on what went well and what could be better.

- **Keeping customers involved**: Frequent feedback means the final product is closer to what users actually want.

- **Reducing risk**: Problems are spotted and fixed early rather than at the end when it's expensive and messy.

*4. What is SRS ?

In software testing, SRS stands for **Software Requirements Specification**.

It’s a very important document that defines what the software should do — before any design, coding, or testing starts. In simple words, it acts like a contract between the client and the development team.

In the context of testing, the SRS is super important because:

- **Testers use it to design test cases**: They check if every requirement mentioned in the SRS is properly implemented and working.
- **It helps in requirement traceability**: Testers map test cases back to specific requirements to ensure full coverage.
- **It serves as a basis for acceptance testing**: To verify if the final product meets the needs described.

A good SRS typically includes:

- Functional requirements (what the system should do)
- Non-functional requirements (like performance, security, usability)
- Constraints (like tech stack, compliance standards)
- Assumptions and dependencies

Example
If the SRS says "The system must allow users to reset their password via email," then testers will create tests specifically checking if that feature works correctly.

- **Use in Testing**: Testers design test cases based on the SRS to verify all requirements are met.

- **Helps in Validation**: Ensures the developed software matches client expectations.

- **Supports Requirement Traceability**: Test cases are mapped to specific requirements for full coverage.

- **Basis for Acceptance Testing**: It is used to validate the final product before delivery.

- **Contents**: Functional requirements, non-functional requirements, constraints, and assumptions.

- **Importance**: A clear SRS = easier and more accurate testing = fewer bugs later.

*5. What is oops ?

In **software testing**, when people mention **OOPS**, they usually mean **Object-Oriented Programming Concepts** applied in testing — it's not a separate thing but connected to how testing is done when the software is written in an object-oriented way.

- If your software is built using **OOP principles** (like classes, objects, inheritance, polymorphism), your **testing** strategies need to adapt to that too.
- You focus on **testing objects**, their **interactions**, **states**, and **behaviors** instead of just testing functions one by one.

In OOPS-based testing, you often do:

- **Unit testing** for **methods of classes**.
- **Mocking objects** to isolate what you're testing.
- **Testing inheritance**: Making sure child classes behave properly.
- **Testing polymorphism**: Making sure overridden methods work as expected.
- **Encapsulation** testing: Verifying that private/internal data isn’t misused.

Frameworks like Java, Python, or .NET are used to easily create tests for object-oriented code.

example:

- Suppose you have a Car class with methods like start(), stop().
- In testing, you'd write unit tests to check if start() works correctly, maybe mock the Engine object it uses inside, and assert the expected outcomes.

Benefits of OOP

OOP has several advantages that make it a powerful paradigm for designing large software systems:

- **Modularity**: You can divide a software system into independent objects or modules, each responsible for a specific task.
- **Code Reusability**: Through inheritance, you can reuse existing code in new ways, reducing redundancy.
- **Maintainability**: Since code is structured in small, modular objects, it's easier to maintain and extend over time.
- **Flexibility and Scalability**: Objects can be easily modified, extended, or replaced without affecting the entire system.
- **Easy Collaboration**: Teams can work on different objects (classes) simultaneously without major conflicts, making OOP suitable for large projects.

*6. <u>Write Basic Concepts of oops ?</u>

1. Class

- A **class** is a blueprint or template for creating objects.
- It defines properties (attributes) and behaviors (methods/functions) that the objects created from it will have.

Example:

```
class Car { }
```

2. Object

- An object is an instance of a class.
- It represents a real-world entity with state (attributes) and behavior (methods).

Example:
Car myCar = new Car();

3. Encapsulation

- **Encapsulation** is the process of wrapping data (variables) and code (methods) together into a single unit (class).
- It restricts direct access to some of the object's components, protecting the integrity of the data.

Example:
Using private variables and public getter/setter methods.

4. Inheritance

- **Inheritance** allows a new class (child class) to inherit properties and behaviors from an existing class (parent class).
- It promotes code reusability.

Example:
class SportsCar extends Car { }

5. Polymorphism

- **Polymorphism** means "many forms".
- It allows objects to be treated as instances of their parent class rather than their actual class.
- It mainly occurs in two ways:
    - **Compile-time Polymorphism** (Method Overloading)
    - **Run-time Polymorphism** (Method Overriding)

6. Abstraction

- **Abstraction** means hiding complex implementation details and showing only the necessary features of an object.
- It helps in reducing programming complexity.

Example:
Abstract classes and interfaces in Java.

*7. <u>What is object ?</u>

- **In everyday language**: An object is just a thing you can see or touch — like a chair, a phone, or a book.

- **In grammar**: An object is the part of a sentence that receives the action of a verb. For example, in "She reads a book," *book* is the object.

- **In programming**: An object is a collection of data and behaviors (functions/methods) bundled together. It's a key part of object-oriented programming (OOP).

- **In philosophy**: An object is anything that can be thought about or perceived.

creating an object — defining its **properties** (data) and **methods** (functions or actions).

*8. What is class ?

a **class** usually refers to a **class in object-oriented programming (OOP)**.

In OOP, a **class** is like a **blueprint** for creating **objects**. It defines the properties (also called **attributes** or **variables**) and behaviors (**methods** or **functions**) that the objects will have.

When you're testing software, especially if it's written in an object-oriented language like Java, Python, or C#, you often test **classes** directly. Here's how it connects:

- You write **unit tests** to test **individual classes**.
- You test **methods** inside the class to make sure they behave correctly.
- You check if the **state** (data inside the class) changes the right way after actions.

*9. What is encapsulation ?

Encapsulation is a core concept in object-oriented programming (OOP) where the internal details of an object (like its data and how it works) are hidden from the outside world. Instead, you interact with the object only through a set of public methods (functions).

In simple words:
Encapsulation means wrapping the data and methods that operate on the data into a single unit (like a class), and controlling access to that data.

For example:

- You make some variables **private** (not directly accessible).
- You allow outside code to access or update them only through **public methods** (getters and setters).

Why use encapsulation?

- It protects the internal state of an object.
- It makes the code more organized and easier to maintain.
- It prevents outside code from messing with internal stuff directly.

*10. What is inheritance ?

Inheritance is a fundamental concept in **Object-Oriented Programming (OOP)** that allows one class (called a **subclass** or **child class**) to inherit properties and methods from another class (called a **superclass** or **parent class**). This concept allows for code reusability, organization, and easier maintenance of the code.

Types of Inheritance

1. **Single Inheritance:**
    - One class inherits from another class.
    - Example: A Dog class inherits from an Animal class.
2. **Multiple Inheritance:**
    - A class inherits from more than one parent class.
    - Example: A HybridCar class inherits from both ElectricCar and GasCar classes.

3. **Multilevel Inheritance:**

    - A class inherits from another class, which itself is a subclass of a parent class.
    - Example: A `Sparrow` class inherits from `Bird`, which inherits from `Animal`.

4. **Hierarchical Inheritance:**

    - Multiple subclasses inherit from a single parent class.
    - Example: Both `Cat` and `Dog` inherit from `Animal`.

5. **Hybrid Inheritance:**

    - A combination of two or more types of inheritance.

Benefits of Inheritance:

1. **Code Reusability:**
    - Inheritance promotes code reuse. You can define common functionality in a parent class, and any subclass can reuse that functionality.
2. **Extensibility:** It allows you to create new classes based on existing ones, adding or modifying behaviors as needed without changing the original code
3. **Organized Code:**
    - It helps in organizing code by creating relationships between classes that mimic real-world hierarchies.

4. **Improved Maintenance:**
    - Changes made in the parent class can be reflected across all subclasses that inherit from it, making it easier to maintain code.

*11. What is polymorphism ?

Polymorphism is one of the fundamental principles of **object-oriented programming** (OOP), along with **encapsulation**, **inheritance**, and **abstraction**.
The word "polymorphism" is derived from Greek, where "poly" means "many" and "morph" means "forms." So, polymorphism essentially means "many forms."

In programming, **polymorphism** allows objects of different classes to be treated as objects of a common superclass. More importantly, it lets a single interface or method act differently based on the object it is operating on. This makes programs more flexible and extensible, improving code readability, reusability, and maintainability.

- **There are mainly two types of polymorphism**:

1. **Compile-time Polymorphism (Static Polymorphism):**
    - This happens when the method call is resolved during compile time.

    - Achieved mainly through **method overloading** and **operator overloading**.
    - **Method Overloading** means having multiple methods in the same class with the same name but different parameters (type, number, or order).

2. **Run-time Polymorphism (Dynamic Polymorphism):**

- This occurs when the method call is resolved during runtime.
- Achieved through **method overriding**, where a subclass provides its own implementation of a method that is already defined in its parent class.
- It is typically used with **inheritance** and **interfaces**.

Why is Polymorphism Important?

- **Flexibility:** It allows writing more generic and flexible code.
- **Reusability:** The same method name can be used for different types.
- **Maintainability:** Code changes are easier because you don't need to modify existing code when adding new classes.

*12. Draw Usecase on online bill payment system (paytm) ?

Actors:

1. User (Customer)
2. Admin
3. Third-Party Service Providers (e.g., Electricity Board, Telecom Operators)
4. Bank/Payment Gateway

Use Cases:

- Register/Login
- View Bill
- Pay Bill
- Choose Payment Method
- Add Money to Wallet
- Receive Payment Confirmation
- Check Transaction History
- Manage Account
- Refund/Dispute Resolution
- Service Provider Bill API Integration
- Payment Processing (by Bank/PG)

*13. Draw Usecase on banking system for customers ?

Actors:

- **Customer**

Use Cases:

1. Open Account
2. Deposit Money
3. Withdraw Money
4. Transfer Funds
5. Check Account Balance
6. View Transaction History
7. Apply for Loan
8. Pay Bills
9. Contact Support.

*14. Draw Usecase on Broadcasting System. ?

Here's a simple Use Case Diagram for a Broadcasting System. This kind of system typically involves actors like administrators, content producers, and viewers. The system itself supports various functionalities such as content creation, scheduling, and broadcasting.

Actors:

1. **Administrator** – Manages users, schedules, and system configurations.
2. **Content Producer** – Uploads and manages media content.
3. **Viewer** – Watches broadcasted content.

Use Cases:

- **Manage User Accounts** (Administrator)
- **Schedule Broadcasts** (Administrator)
- **Upload Content** (Content Producer)
- **Edit Content** (Content Producer)
- **View Broadcasts** (Viewer)
- **Receive Notifications** (Viewer)
- **Monitor System Status** (Administrator)

*15. Write SDLC phases with basic introduction. ?

The **Software Development Life Cycle (SDLC)** is a structured process used by software developers to design, develop, test, and deploy high-quality software. It provides a framework for managing software projects and ensures that the end product meets customer requirements and expectations. Here are the main phases of the SDLC with a basic introduction to each:

1. Requirement Gathering and Analysis

This is the initial phase where all relevant information is collected from stakeholders and users to understand what the software needs to do. Analysts work to document requirements clearly and check for feasibility.

2. System Design: In this phase, the requirements are translated into a blueprint for the software. This includes defining architecture, system components, user interfaces, data models, and more. The design helps in specifying hardware and system requirements.

3. Implementation (or Coding)

Developers begin writing code based on the design documents. The system is built in small units and tested for functionality. This phase is typically the longest and involves both frontend and backend development.

4. Testing

The developed software is tested to find and fix bugs. Various types of testing (unit, integration, system, acceptance) ensure the software meets the defined requirements and works as expected.

5. Deployment

After successful testing, the software is deployed to the production environment for users to access. This can be done in stages, such as pilot releases, or in full-scale launches depending on the project strategy.

6. Maintenance

Once deployed, the software enters the maintenance phase. This includes fixing bugs not discovered during earlier testing, upgrading features, and making improvements based on user feedback.

*16. Write phases of spiral model. ?

The **Spiral Model** is a **risk-driven** software development process model that combines elements of both **iterative** and **waterfall** models. It emphasizes risk analysis and repeated **refinement** through multiple iterations (or spirals).

Each loop (or spiral) in the model represents a phase in the software development process.

Phases of the Spiral Model

Each spiral cycle is divided into four main phases:

1. Planning Phase

- Objectives, alternatives, and constraints of the project are identified.

- Requirements are gathered and understood.
- Initial estimates for cost, schedule, and resources are made.

Outputs:

- Requirements documents
- Project planning details

2. Risk Analysis Phase

- Identify, assess, and resolve potential risks (technical, management, operational).
- Prototyping may be done to reduce risk.
- This phase is unique to the Spiral Model and its most critical feature.

Outputs:

- Risk assessment reports
- Prototypes (if needed)
- Mitigation strategies

3. Engineering Phase

- Actual development and testing of the software product or prototype.
- Includes design, coding, and unit testing.
- Iterative development may be used in this phase.

Outputs:

- Working product versions
- Test results

4. Evaluation Phase

- Stakeholders evaluate the progress and provide feedback.
- Decide whether to proceed with another spiral (next phase), modify the plan, or abandon the project.

Outputs:

- Evaluation reports
- Stakeholder approval
- Decision for next iteration

Key Characteristics:

- Focuses heavily on risk management
- Supports iterative development and customer feedback
- Suitable for large, complex, and high-risk projects

*17. Write agile manifesto principles. ?

Deliver quality software early and continuously through frequent and effective testing that adds value to the customer.

1. **Embrace changing requirements** by designing flexible and maintainable test strategies that can evolve with the product.
2. **Test frequently and iteratively**, ensuring fast feedback and continuous improvement in every development cycle.
3. **Collaborate closely with developers, product owners, and stakeholders** to understand expectations, reduce ambiguity, and ensure shared quality goals.
4. **Empower testers as integral members of the agile team**, providing them autonomy and encouraging proactive involvement in the entire software lifecycle.
5. **Foster open and direct communication** within the team to quickly identify and resolve issues, ideally through face-to-face conversations or real-time communication tools.
6. **Working software with verified quality is the primary measure of progress**—not just software that "runs."
7. **Promote sustainable testing practices**, avoiding burnout by maintaining a steady pace and automating repetitive tasks where feasible.
8. **Pursue technical excellence in testing** by continuously improving test automation, test data management, and coverage strategies.
9. **Focus on simplicity** by writing just enough tests to cover business-critical paths, avoiding over-testing or unnecessary complexity.
10. **Encourage self-organizing teams where testers contribute to all phases**, from planning to delivery, and help guide quality ownership across roles.
11. **Reflect and adapt regularly** by inspecting test processes, tools, and outcomes, and seeking ways to improve effectiveness and team synergy.

*18. Explain working methodology of agile model and also write pros and cons. ?

The Agile model is a software development approach that emphasizes iterative development, collaboration, continuous feedback, and flexibility to change. It contrasts with traditional models like the Waterfall model by breaking the project into smaller, manageable units called iterations or sprints, each typically lasting 1–4 weeks.

1. **Requirements Gathering:**
    - Initial high-level requirements are collected.
    - Detailed requirements are defined incrementally with each sprint.
2. **Planning:**
    - The project is divided into small cycles (sprints).
    - A **backlog** (list of features/tasks) is created and prioritized.
3. **Design & Development:**

- Each sprint involves designing, developing, and testing a working feature.
- Agile encourages **cross-functional teams** where developers, testers, and designers collaborate closely.

4. **Daily Stand-Up Meetings:**
    - Short daily meetings help team members sync up on progress, roadblocks, and plans.
5. **Testing:**
    - Testing is integrated into every sprint (continuous testing).
    - Bugs are identified and resolved quickly.
6. **Delivery:**
    - A potentially shippable product increment is delivered at the end of each sprint.
7. **Review & Feedback:**
    - Sprint review meetings allow stakeholders to provide feedback.
    - Teams adapt based on feedback and adjust the backlog as needed.
8. **Retrospective:**
    - The team reflects on what went well and what can be improved for the next sprint.

Pros of agile model

1. **Customer Satisfaction:**
    - Continuous delivery of valuable software keeps stakeholders engaged and satisfied.
2. **Flexibility & Adaptability:**
    - Easily accommodates changes even late in the development cycle.
3. **Faster Delivery:**
    - Frequent releases ensure quicker time to market.
4. **Improved Quality:**
    - Regular testing and integration reduce bugs and improve stability.
5. **Better Collaboration:**
    - Regular communication fosters a stronger team environment.
6. **Transparency:**
    - Stakeholders can track progress and provide feedback throughout the process.

Cons of agile model

1. **Requires Experienced Team:**
    - Teams need strong collaboration skills and self-discipline.
2. **Poor Documentation:**
    - Agile focuses more on working software than comprehensive documentation, which may be problematic for future maintenance.
3. **Scope Creep:**
    - Constantly changing requirements may lead to uncontrolled growth in scope.
4. **Not Ideal for Fixed Contracts:**
    - It can be hard to estimate cost and timelines upfront.
5. **May Lack a Clear End Goal:**
    - Without strict initial planning, teams can lose sight of the end product.

*19. Draw usecase on OTT Platform. ?

simple **Use Case Diagram** for an **OTT (Over-the-Top) Platform**, such as Netflix, Amazon Prime, or Disney+.

Actors:

- **User (Viewer)**
- **Admin**
- **Content Provider**

Use Cases:

For User (Viewer):

- Register / Login
- Browse Content
- Search Content
- Watch Video
- Manage Subscription
- Rate / Review Content
- Add to Watchlist
- Download Content (if allowed)
- Receive Recommendations

For Admin:

- Manage Users
- Manage Content
- Manage Subscriptions
- View Analytics
- Handle Reports / Feedback

For Content Provider:

- Upload Content
- View Content Performance
- Manage Metadata (title, description, etc.)

*20. Draw usecase on E-commerce application. ?

Actors:

1. **Customer**
2. **Admin**

3. **Payment Gateway (external system)**

- **Use Cases:**

- Register / Login
- Browse Products
- Search Products
- Add to Cart
- Place Order
- Make Payment
- Track Order
- Manage Profile
- Manage Inventory (Admin)
- Process Orders (Admin)
- Manage Users (Admin)

Example: flipcart, amazon

***21. <u>Draw usecase on Online shopping product using payment gateway. ?</u>**

Actors:

1. **Customer**
2. **Admin**
3. **Payment Gateway**

Use Cases:

- **Customer:**
    - Register / Login
    - Browse Products
    - Add to Cart
    - Checkout
    - Make Payment
    - Track Order
    - View Order History
- **Admin:**
    - Manage Products

        - Manage Orders
        - Manage Users
    - **Payment Gateway:**
        - Validate Payment
        - Process Transaction
        - Send Payment Status

*22. <u>**Explain Phases of the waterfall model. ?**</u>

- **Requirement Gathering and Analysis**
    - In this phase, all possible requirements of the system to be developed are gathered from stakeholders.
    - The requirements are documented and analyzed to ensure clarity and completeness.
    - The output is a **Software Requirements Specification (SRS)** document.
- **System Design**
    - Based on the SRS, system architecture and design are prepared.
    - This includes both **high-level design** (architecture, components) and **low-level design** (detailed logic, data structures).
    - The goal is to define how the system will meet the requirements.
- **Coding**
    - Developers start writing the actual code based on the design documents.
    - The system is typically built in small modules or units and then integrated.
    - This phase produces the **working software**.
- **Integration and Testing**
    - Once all modules are developed, they are integrated and tested as a complete system.
    - Testing focuses on finding defects and ensuring the software meets the specified requirements.
    - Includes different types of testing like unit, integration, system, and acceptance testing.
- **Deployment**
    - The software is delivered or deployed to the customer or end-user environment.
    - It may be released in stages depending on the deployment plan.
- **Maintenance**

- After deployment, the software may require updates, bug fixes, or enhancements.
- Maintenance ensures the system continues to function correctly and adapts to changing user needs.