Object-Oriented Programming

1. What are the 4 pillars?

- There are four major object-oriented principles. They are Encapsulation, Abstraction, Inheritance, and Polymorphism.

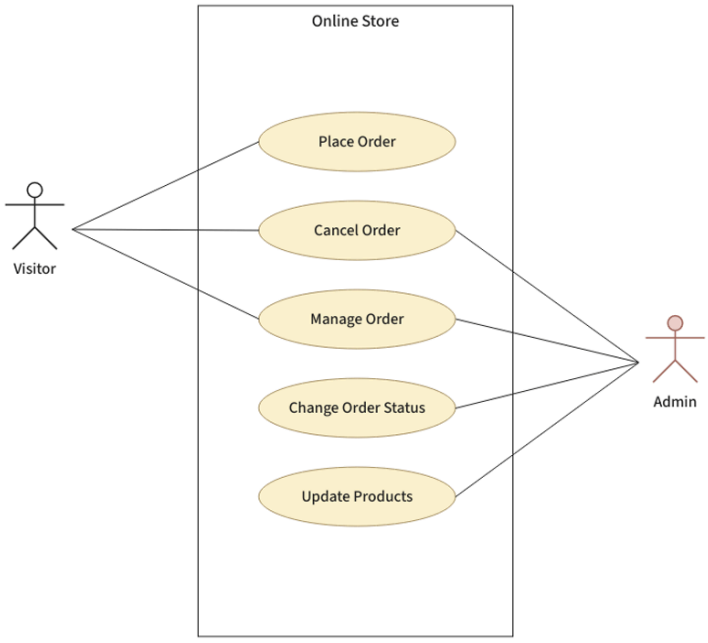
2. Why they are important and how they are applied in Java.

- The OOP pillars are important because they make programs more modular, reusable, and easier to debug and maintain. In Java, they applied through classes and objects (encapsulation), interfaces, and abstract classes (abstraction), extends and implements (inheritance), and method overloading/overriding (polymorphism).

Use Case Diagrams

1. What they are and how to make them.

- It provided a visual way to document user goals and explore possible functionality.



2. Benefits of Use Case Diagrams.

- It helps visualize how users interact with a system, making it easier to understand system functionality and requirements. They also improve communication among stakeholders and provide a clear boundary of the system's scope.

UML Diagrams

1. Be able to read UML diagrams.

2. Understand the different relationships between classes.

3. Be able to create a UML diagram from a set of classes.

Design Principles

1. SOLID principles.

- SOLID is a set of five object-oriented design principles that help developers write cleaner, more maintainable, and flexible code. These principles make large projects easier to manage and reduce bugs when the software grows.

2. Understand what they are and how to apply them.

S — Single Responsibility Principle (SRP)

A class should have **one job** and **one reason to change**. This keeps classes small, focused, and easier to maintain.

O — Open/Closed Principle (OCP)

Classes should be **open for extension but closed for modification**. You should be able to add new features by extending code, not rewriting existing code.

L — Liskov Substitution Principle (LSP)

Subclasses should be able to **replace** their parent classes without breaking the program. If class B extends class A, B should behave like A.

I — Interface Segregation Principle (ISP)

No class should be forced to implement **methods it doesn’t need**. It's better to create **small, specific interfaces** rather than large, general ones.

D — Dependency Inversion Principle (DIP)

Classes should depend on **abstractions, not concrete implementations**. This reduces tight coupling and makes code easier to test and change.

Design Patterns

1. Understand what the design patterns are.

- Design patterns are **proven, reusable solutions to common software design problems** that occur in real-world programming. They provide a **standardized way to structure code**, improve maintainability, scalability, and readability, and help developers communicate ideas more effectively.

2. Observer, Singleton, Factory, Adapter Patterns.

- **Observer Pattern:** This pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically. It is commonly used in event handling systems, like GUI updates or messaging apps.
- **Singleton Pattern:** Singleton ensures that a class has only one instance and provides a global point of access to it. It is often used for managing shared resources, such as database connections or configuration settings.
- **Factory Pattern:** The Factory pattern provides an interface for creating objects but lets subclasses decide which class to instantiate. It promotes loose coupling by encapsulating object creation, making code easier to extend and maintain.
- **Adapter Pattern:** The Adapter pattern allows incompatible interfaces to work together by converting one interface into another expected by clients. It is useful when integrating third-party libraries or legacy code with modern systems.

3. Facade, Decorator, Command, Iterator, State, Composite, Observer, etc.

- **Facade Pattern:** Facade provides a simplified interface to a complex subsystem, making it easier for clients to interact with. It helps reduce dependencies and hides the internal complexities of the system.

- **Decorator Pattern:** Decorator allows behavior to be added to individual objects dynamically without affecting other objects of the same class. It promotes flexible and reusable code by wrapping objects with additional functionality.
- **Command Pattern:** Command encapsulates a request as an object, allowing you to parameterize clients with queues, requests, or operations. It is useful for implementing undo/redo functionality and decoupling the sender from the receiver.
- **Iterator Pattern:** Iterator provides a way to access elements of a collection sequentially without exposing its underlying representation. It allows different traversal methods for a collection without modifying its structure.
- **State Pattern:** State allows an object to alter its behavior when its internal state changes, making it appear as if the object's class has changed. It is often used in scenarios like workflow engines or game character states.
- **Composite Pattern:** Composite lets you treat individual objects and compositions of objects uniformly. It is useful for representing hierarchical structures like file systems or UI components.
- **Observer Pattern:** Observer defines a one-to-many dependency so that when one object changes, all its dependents are notified. It is commonly used in event handling systems, like GUI updates or messaging apps.
- **Proxy Pattern:** It provides a substitute or placeholder object that controls access to another object, allowing you to add extra behavior like caching, security checks, or lazy initialization. It helps manage resource-heavy or sensitive objects by letting the proxy handle requests before forwarding them to the real object.
- **Visitor pattern:** It lets you add new operations to a group of related objects without modifying their classes by separating the operation logic into a separate "visitor" object. It's useful when you have many object types and want to apply different behaviors to each while keeping their classes clean and stable.

4. Be able to provide a simple example of the patterns above.

MVC

1. Know what it is.

MVC (Model-View-Controller): MVC is a design pattern that separates an application into three components: Model (manages data and business logic), View (handles the user interface), and Controller (processes user input and updates Model/View). This separation improves code organization, maintainability, and allows multiple views to use the same data.

2. Be able to provide an example with MVC.

Imagine a blog application where the Model handles storing and retrieving posts from a database, the View displays the list of posts to users, and the Controller processes user actions like creating, editing, or deleting a post. When a user submits a new post, the Controller updates the Model, which then updates the View to show the latest posts.

Testing

1. Know what TDD is.

TDD (Test-Driven Development): TDD is a software development approach where you write tests before writing the actual code. It ensures code correctness, encourages simpler designs, and helps catch bugs early in the development process.

2. Black Box vs Gray Box vs White Box testing.

Black Box Testing: In black box testing, the tester evaluates the software without any knowledge of its internal code or structure, focusing only on inputs and expected outputs. It is useful for validating functionality and user requirements.

White Box Testing: White box testing examines the internal logic, code structure, and implementation details of the software. It is used to ensure all code paths, conditions, and branches work correctly.

Gray Box Testing: Gray box testing combines both black box and white box approaches, where the tester has partial knowledge of the internal code. It is effective for identifying security issues, integration problems, and system vulnerabilities.

3. Know what unit testing is and basics of JUnit.

Unit Testing: Unit testing is the practice of testing individual components or functions of a program in isolation to ensure they work correctly. It helps detect bugs early and makes code easier to maintain and refactor.

JUnit Basics: JUnit is a popular Java framework for writing and running unit tests, allowing developers to define test methods with annotations like `@Test`. It provides assertions such as `assertEquals()` and `assertTrue()` to verify that code behaves as expected.

4. Understand what code coverage is.

Code Coverage: Code coverage measures the percentage of your code that is executed while running tests, showing how much of your program is tested. Higher coverage helps identify untested parts of the code, improving reliability and reducing the risk of bugs.

5. Understand what System, Integration, Acceptance Testing is.

System Testing: System testing evaluates the complete, integrated software to ensure it meets the specified requirements. It focuses on verifying the overall behavior and functionality of the system as a whole.

Integration Testing: Integration testing checks how different modules or components of a system work together. It aims to detect interface defects and ensure that combined parts function correctly.

Acceptance Testing: Acceptance testing determines whether the software meets the business requirements and is ready for delivery to users. It is usually performed by end-users or clients to validate that the system fulfills their needs.