State the fundamental concept of the following design principles with examples.
i. Single-Responsibility Principle (SRP)
ii. Liskov Substitution Principle (LSP)
iii. Dependency Inversion Principle (DIP)

Single-Responsibility Principle (SRP):

- **Concept:** A class should have **one and only one reason to change**. In other words, each class should focus on a single, well-defined responsibility. This promotes modularity, reusability, and maintainability.
- **Example:**

  User class is separated into classes like UserManager, AuthenticationService, and EmailService. Each class has a single responsibility, resulting in cleaner, more manageable code.

Liskov Substitution Principle (LSP):

- **Concept:** Subtypes should be substitutable for their base types without altering the correctness of the program. In simpler terms, objects of a subclass should seamlessly replace objects of its parent class without unexpected behavior.
- **Example:**

  Imagine a base class Shape with a draw() method. A subclass Triangle overrides draw() but draws a square instead. This violates LSP, as using a Triangle where a Shape is expected leads to incorrect behavior.

  Ensure sub-classes like Triangle adhere to the expected behavior of the base class Shape in terms of draw(), preventing unexpected outcomes.

Dependency Inversion Principle (DIP):

- **Concept:** High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions. This promotes loose coupling, making the code more flexible and adaptable.
- **Example:**

  A complex OrderProcessor class introduces an abstraction like a DatabaseInterface (high-level) that defines database operations regardless of implementation. Both OrderProcessor and the concrete database classes depend on this interface, leading to loose coupling and easier adjustments when changing database implementations.

Both SRP and ISP deal with Cohesion. Describe how they are different.

Single Responsibility Principle (SRP):

- **Focus:** SRP focuses on the internal cohesion of a class. It emphasizes that a class should have one and only one reason to change. In other words, each class should focus on a single, well-defined responsibility. This ensures that changes to the class are always motivated by a single concern, making the code more modular and maintainable.
- **Impact on Cohesion**: By ensuring a class has a single responsibility, SRP promotes high internal cohesion. All members of the class work together towards a single goal, making the class more unified and understandable.

Interface Segregation Principle (ISP):

- **Focus:** ISP focuses on the external cohesion of a class. It emphasizes that clients should not be forced to depend on features they don't need. This means creating smaller, more focused interfaces that cater to specific client needs, instead of one large interface that encompasses everything.
- **Impact on Cohesion:** By creating specialized interfaces, ISP promotes higher cohesion between clients and interfaces. Clients only depend on the features they need, reducing unnecessary dependencies and improving code understandability and efficiency.

Explain how code smell is removed by refactoring and refactoring requires Unit Test. Justify your answer.

Refactoring is the process of improving the internal structure of your code without changing its external behavior. This includes removing code smells, which are indicators of potential problems with the code structure, maintainability, or readability.

Each code smell has specific refactoring techniques associated with it. For example:

- Large Class: You can refactor a large class into smaller, more focused classes with single responsibilities.
- Duplicate Code: You can extract the duplicated code into a reusable function or class.
- Long Method: You can break the method down into smaller, more meaningful methods.

By applying these techniques, you clean up the code structure, improve its understandability, and make it less prone to errors.

Why Unit Tests are Necessary for Refactoring:

Refactoring changes the internal structure of your code, even though the external behavior should remain the same. This makes unit tests crucial for several reasons:

- Safety Net: Unit tests act as a safety net, ensuring that the refactoring doesn't unintentionally introduce bugs or change any existing functionality. They verify that the code still behaves as expected after the refactoring process.
- Confidence Boost: Running unit tests after each refactoring step provides confidence that the changes haven't caused regressions. This allows you to refactor with greater peace of mind.
- Improved Maintainability: Refactoring often involves writing new code or reorganizing existing code. Writing unit tests alongside the refactoring process helps document the expected behavior of the code and improves its maintainability in the long run.