"A good design should exhibit high cohesion and low coupling."-Justify the assertion by considering two principles of SOLID.

1. Single Responsibility Principle (SRP):

- **High Cohesion:** SRP dictates that a class should have only one reason to change. This implies that all functionalities within the class are closely related and contribute to a single, well-defined responsibility. This results in classes with **high internal cohesion**, where elements work together tightly towards a specific goal.
- **Low Coupling:** By focusing on a single responsibility, classes become less dependent on other modules or external factors for completion. They don't need to access unrelated data or functionalities, thereby minimizing their external dependencies and achieving **low coupling** with other code components.

Consider an example: a class solely responsible for user authentication (login, sign-up, password reset). Under SRP, this class wouldn't handle additional functionalities like managing user profiles or sending emails. This focus on one responsibility keeps the class cohesive and minimizes its dependencies on other modules, exemplifying both high cohesion and low coupling.

2. Interface Segregation Principle (ISP):

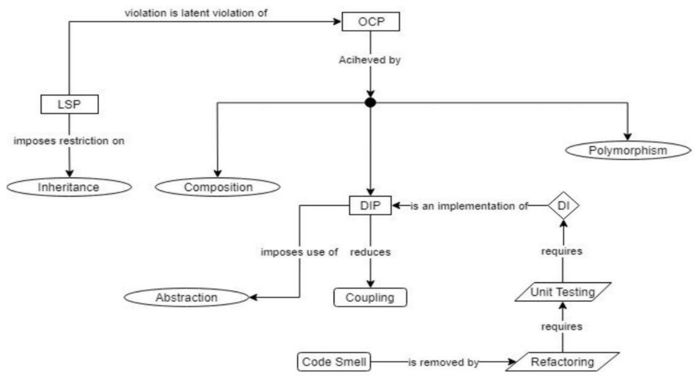
- **High Cohesion:** ISP suggests splitting fat interfaces into smaller, more specific ones. This ensures that clients only depend on functionalities they actually use. Each smaller interface represents a cohesive subset of functionalities, focusing on a clearly defined purpose. This leads to classes that are highly integrated within their specific interface context.
- **Low Coupling:** By decoupling clients from unused functionalities in large interfaces, ISP reduces unnecessary dependencies. Clients only rely on the specific interface that meets their needs, minimizing their coupling with other code components.

Imagine a "DatabaseAccess" interface containing all database operations (read, write, update, delete). Applying ISP would split this interface into smaller ones like "ReaderInterface" and "WriterInterface." Users accessing only read functionalities wouldn't be burdened with unnecessary methods for writing, reducing their coupling with write-related code.

These two SOLID principles demonstrate how high cohesion and low coupling work together to achieve good design. Cohesive classes are easier to understand, maintain, and reuse, while low coupling facilitates independent development and testing of modules. Ultimately, both principles contribute to code that is more robust, flexible, and adaptable to change - key characteristics of a well-designed system.

Draw a diagram to illustrate how SOLID principles are related with each other.

Interrelationships between OO concepts



1. OCP is the fundamental principle. This is because it is related to **change** in software.
   a. DIP helps implementing OCP
   b. LSP is a subset of OCP
2. Encapsulation is still missing in the graph

"Violation of Liskov Substitution principle (LSP) is a latent violation of open Close Principle (OCP)."- Explain the statement with an example.

The statement "Violation of Liskov Substitution principle (LSP) is a latent violation of Open-Close Principle (OCP)" means that if you break LSP in your code, it will inevitably lead to situations where you need to modify existing code to handle new types, violating OCP. Let's see how this plays out with an example of birds and ostriches in Java:

LSP Violation:

Imagine we have a base class `Bird` with methods like `fly()` and `sing()`. We then create a subclass `Ostrich` inheriting from `Bird`. Ostriches cannot fly, but they still have feathers and make sounds. Here's where LSP is violated:

1. `fly()` behavior is not substitutable: When expecting a `Bird`, you assume it flies. But if you receive an `Ostrich` through LSP, calling `fly()` will break your code as ostriches don't fly.

OCP Violation:

To handle ostriches, you must modify existing code:

1. Conditional checks: You need to add checks within Bird methods to differentiate between flying and non-flying birds, adding complexity and potentially affecting existing clients.
2. New exceptions: You might throw exceptions in fly() if an ostrich is passed, forcing clients to handle edge cases.

This violates OCP, as adding new types (ostriches) requires changing existing code (Bird methods). Ideally, accommodating new types shouldn't break existing clients.

```
interface Bird {
  void fly(); // Assumes all birds can fly
  void sing();
}

class Ostrich implements Bird {
  @Override
  public void fly() { // This breaks LSP as ostriches don't fly
    throw new UnsupportedOperationException("Ostriches can't fly!");
  }
```

Solution:

To fix this, we need to consider LSP from the start:

1. Separate flying birds: Create a FlyingBird interface containing fly() and inherit it by flying birds. Ostriches wouldn't inherit this interface.
2. Abstract common behavior: Keep common methods like sing() in the Bird class.

This way, clients expecting flying behavior rely on the FlyingBird interface, ensuring LSP. Adding ostriches doesn't require modifying Bird or its clients, upholding OCP.

A software development team, one of the developers encouraged others to perform refactoring by saying that "critical bugs will be fixed as a result of refactoring."
i) Explain why the statement above incorrectly uses the term refactoring.
ii) Based on the definition of refactoring, explain how unit testing is related to it.

Part i) Incorrect Use of Refactoring:
The statement "critical bugs will be fixed as a result of refactoring" incorrectly uses the term refactoring for several reasons:

1. **Purpose of refactoring:** Refactoring aims to improve the design and code structure without changing its external behavior. Its primary goal is not bug fixing, although improved structure can make bugs easier to find and fix in the future.
2. **Unpredictable bug fixing:** While cleaner code structure might reveal previously obscure bugs, refactoring itself doesn't guarantee fixing critical bugs. It's more likely to improve code maintainability and readability, enabling easier future bug detection and resolution.
3. **Risk of regressions:** Introducing even well-intended changes through refactoring carries the risk of introducing new bugs, potentially worsening the situation instead of fixing critical ones.

Therefore, using refactoring as a primary solution for fixing critical bugs is misleading and potentially dangerous. While improved code structure can contribute to better bug detection and handling in the long run, it's not a substitute for dedicated bug fixing efforts.

Part ii) Unit Testing and Refactoring:
Unit testing plays a crucial role in refactoring for several reasons:

1. **Regression prevention:** Before refactoring, existing unit tests ensure the current code functionality is correct. These tests act as a safety net, verifying that the refactored code maintains the intended behavior.
2. **Confidence in changes:** Running unit tests after refactoring provides confidence that the changes haven't inadvertently introduced new bugs. This feedback loop ensures the refactoring process remains controlled and predictable.
3. **Improved maintainability:** By writing unit tests alongside refactoring, you create living documentation that clarifies the code's behavior and purpose. This improves long-term code maintainability and facilitates future understanding and modification.

In essence, unit testing and refactoring go hand-in-hand. Unit tests provide a safety net during the refactoring process, while refactoring can improve the testability of the code, making future maintenance and updates easier.

Briefly explain each of the following code smells with an example:
1. Refused Bequest
2. Middle Man

1. Refused Bequest:

This smell occurs when a subclass inherits but then ignores or actively hides functionality provided by its parent class. It indicates that the inheritance relationship might be inappropriate or misleading.

Example: Imagine a `Shape` class with a `draw()` method and a `Circle` subclass inheriting it. However, the `Circle` class overrides `draw()` to always draw a square instead of a circle. This is Refused Bequest, as the subclass rejects the parent's intended behavior.

2. Middle Man:

This smell involves a class that serves as a useless intermediary between another class and its intended receiver. It adds no value and only clutters the code, potentially hindering performance and understanding.

Example: Let's say you have a `Sensor` class and a `Logger` class, and both communicate directly. Introducing a `DataProcessor` class solely to pass data from `Sensor` to `Logger` without any processing would be a Middle Man smell.

How is encapsulation different from information hiding? Justify your answer.

Encapsulation:

- **Broader Scope:** Encapsulation refers to the bundling of data and operations together within a unit, such as a class in object-oriented programming. It's about creating self-contained modules that manage their internal state and expose interfaces for interaction with the outside world.
- **Focus on Functionality**: The emphasis is on providing a unified and clear interface for manipulating the data within the unit. Encapsulation aims to simplify interaction and protect internal details from unintended modification.
- **Implementation:** Techniques like access modifiers (public, private, etc.) and methods are used to achieve encapsulation.

Information Hiding:

- **Narrower Focus:** Information hiding specifically focuses on restricting access to certain data within a unit. It emphasizes protecting internal data from unauthorized or improper manipulation.
- **Emphasis on Security:** The primary goal is to prevent code outside the unit from directly accessing or modifying sensitive or critical data.
- **Part of Encapsulation:** Information hiding is a key technique used to achieve encapsulation. By restricting access to sensitive data, the overall unit becomes more robust and reliable.

Understanding the difference between these concepts is important for writing more secure and maintainable code.

- **Separation of concerns:** Encapsulation allows developers to separate the "how" (implementation) from the "what" (interface) of a unit. This makes code easier to understand, modify, and reuse.
- **Data integrity:** Information hiding protects sensitive data from accidental or malicious modification, enhancing program stability and security.
- **Improved maintainability:** By clearly defining boundaries and interfaces, both concepts make code easier to maintain and evolve over time.

What is clean code? Write four practices to ensure clean code?

Clean code is code that is easy to understand, maintain, and extend. It prioritizes clarity, readability, and efficiency. Clean code isn't just about functionality; it's about making the code as clear and transparent as possible for other developers, including your future self.

Here are four key practices to ensure clean code:

1. **Write meaningful and descriptive names:** Variables, functions, and classes should have names that accurately reflect their purpose and behavior. Don't use abbreviations or obscure terms. Choose names that are informative and unambiguous, like `calculateDiscount` instead of `calcD`.

2. **Focus on small, single-responsibility functions:** Break down complex tasks into smaller, focused functions that perform a single, well-defined action. This makes the code easier to understand, test, and reuse. Avoid functions that do too much or attempt multiple unrelated things.

3. **Keep it concise and consistent:** Use appropriate levels of indentation and white space to improve readability. Follow a consistent coding style (e.g., curly brace placement, naming conventions) throughout your project to avoid confusion and enforce maintainability. Strive for concise code that expresses everything clearly without unnecessary complexity.

4. **Leverage comments strategically:** Use comments sparingly to explain non-obvious logic or complex sections of code. Avoid redundant comments that simply restate the code itself. Focus on explaining "why" something is done, not just "what" it does.

Briefly describe the retention policy of custom annotation.

Custom annotations in programming languages like Java can have three different retention policies that determine their lifespan during program execution:

1. **SOURCE:** The annotation is only present in the source code and not included in the compiled class file. It's useful for documentation purposes and code analysis during compilation but has no effect at runtime.

2. **CLASS:** The annotation is stored in the compiled class file but is not accessible at runtime. Its information can be used by class loaders or reflection tools during class loading or analysis, but the application code itself cannot access it at runtime.

3. **RUNTIME:** The annotation is stored in the compiled class file and is also accessible at runtime. This allows application code to read and utilize the annotation information using reflection mechanisms. This is the most versatile option but can incur performance overhead and increase class file size.

Describe the final keyword? Difference between variable final and object final.

The final keyword in many programming languages, like Java, has different applications depending on the context. Here's a breakdown:

For variables:

- **Declares a constant value:** When used with a variable, `final` makes its value unchangeable once initialized. This is useful for constants like `PI` or configuration settings.
- **Reference can be final, not value:** If the variable type is a reference type (e.g., object), the `final` keyword makes the reference itself unchangeable, meaning you can't reassign the variable to a different object. However, the object's internal state can still be modified unless declared final within the object itself.

For methods:

- **Prevents overriding in subclasses:** Declaring a method as `final` prevents subclasses from **overriding** that method. This is useful for core functionality you want to keep consistent across the inheritance hierarchy.

For classes:

- **Prevents subclassing:** Making a class `final` prohibits any other class from **extending** it. This can be useful for utility classes or singletons where having multiple instances is undesirable.

Final Variable:

- **Immutability:** When used with a variable, final makes its value unchangeable. This applies to both primitive and reference types. For primitives (int, double, etc.), the actual value becomes locked. For references, the reference itself cannot be reassigned to point to a different object, although the object's internal state might still be mutable.

```
final String NAME = "John"; // Unchangeable string reference
```

Final Object:

- **Unmodifiable reference:** When used with an object, final makes the reference to the object itself unchangeable. You cannot reassign the variable to point to a different object. But, the internal state of the object can still be modified unless declared final within its class for specific fields.

```
final Person person = new Person("John", 30); // Reference to 'person'
is final

person.setName("Alice"); // This is allowed as object's state is
mutable
```