A common principle in Object-Oriented Concepts is "prefer composition over inheritance".
How do you choose one over another? Justify your preference with an example.

Composition:

- **Concept:** Utilizes existing objects to achieve desired functionality. This involves creating a class and embedding instances of other classes within it to leverage their behavior.
- **Strengths:**
  - **Flexibility:** Allows composing different functionality from various independent classes, leading to flexible and adaptable designs.
  - **Code Reuse:** Promotes code reuse through existing classes without modifying their internal logic.
  - **Decoupling:** Encourages loose coupling between classes, making them more independent and easier to change and test.

Inheritance:

- **Concept:** Extends behavior of an existing class (superclass) to create a new subclass that inherits its properties and adds additional functionalities.
- **Strengths:**
  - **Code Sharing:** Enables direct code reuse from the superclass, reducing code duplication and promoting conciseness.
  - **Is-A Relationship:** Clearly represents "is-a" relationships between classes, such as Square is-a Shape.
  - **Liskov Substitution Principle:** Adheres to Liskov Substitution Principle (LSP) if the subclass faithfully upholds the superclass behavior with extensions.

Choosing Between Them:

Here's a general guideline for choosing between composition and inheritance:

- **Prefer composition for:**
  - Aggregating functionalities from unrelated classes, especially when the relationship is not "is-a".
  - Promoting flexibility and adaptability as requirements change.
  - Decoupling classes to ease modification and testing.
- **Prefer inheritance for:**
  - "Is-a" relationships, where the subclass logically extends the superclass behavior.
  - Code reuse where sharing functionalities from the superclass makes sense.
  - Following Liskov Substitution Principle when extending and modifying behaviors.

Example:

Imagine a scenario where you need to create a FlyingCar class.

- **Composition Approach:**
  - Create a Car class and a FlyingVehicle class with their respective functionalities (engine, wheels, fly, land).
  - The FlyingCar class can then compose instances of both Car and FlyingVehicle to get its combined behavior.
- **Inheritance Approach:**
  - Create a Car class and a FlyingCar class that inherits from Car.
  - The FlyingCar class can override or extend functionalities from Car while adding its own flying capability.

Briefly explain each of the following code smells with an example:

1. **Feature Envy**
2. **Shotgun Surgery**
3. **Primitive Obsession**

Code Smells and Examples:

Feature Envy:

A class's method accesses data or methods of another class more frequently than its own data, indicating a misplaced responsibility. The envious class seems more interested in the other class's functionality.

Example: Imagine a `UserInterface` class displaying information from a `UserManager`. Instead of accessing individual user details directly, the `UserInterface` uses methods like `UserManager.getFirstName(user)` for every user property it needs. This makes the `UserInterface` overly dependent on the `UserManager` and hinders flexibility.

Shotgun Surgery:

When modifying the same code in multiple unrelated places to fix a bug or add a feature. This duplication indicates a lack of abstraction and can lead to inconsistencies and maintenance difficulties.

Example: Adding validation logic for a specific field might require changing code in several functions across different classes, like login, registration, and profile editing. This violates DRY (Don't Repeat Yourself) and makes updates prone to errors and inconsistencies.

Primitive Obsession:

Excessive use of primitive data types (int, string, etc.) within class fields, parameters, and return values. This can lead to tight coupling, inflexible design, and difficulty in handling complex data representations.

Example: A `Money` class representing financial amounts might use simple `int` variables for storing sums. This lacks features like currency, precision, and operations specific to money, making it less robust and prone to data manipulation errors.

Explain the "Incomprehensibly Concise" and "shameless green"Shameless Green' approaches to implementing a requirement for the 99-bottles problem. Which one is considered a better practice? Justify your choice.

Incomprehensibly Concise:

- **Strengths:** Aims for minimal code, resulting in a very short program. This can be appealing for its simplicity and compactness.
- **Weaknesses**: Often sacrifices readability and maintainability in favor of brevity. Obscure tricks, cryptic code, and unorthodox practices can make it difficult for others to understand and modify. This can hinder future maintenance and collaboration.

Shameless Green:

- **Strengths:** Focuses on passing tests without necessarily adhering to good coding practices. This can be helpful for quickly meeting basic requirements and achieving functional code.
- **Weaknesses:** Encourages shortcuts and potentially bad practices like code duplication, lack of clarity, and disregard for design principles. This can lead to messy, brittle code that's difficult to maintain and evolve in the long run.

Neither approach is ideal, and striking a balance between conciseness and clarity is crucial. Here's why:

- **Maintainability and Readability:** Code should be understandable not just to the original writer but also to anyone who might need to work on it later. Obfuscating logic or ignoring good practices creates a hurdle for collaboration and future modifications.
- **Clean Design:** While minimal code might seem attractive, elegant and organized code with clear structure and proper abstraction is more sustainable and reusable.
- **Test Coverage:** Passing tests is essential, but solely focusing on this while neglecting code quality will result in fragile and hard-to-adapt code in the long run.