AEC1 – Object Oriented Programming

ISMAEL HERNÁNDEZ CLEMENTE

Inheritance vs. Composition

Which Is the Best Design Strategy?

In the realm of Object-Oriented Programming (OOP), two fundamental strategies often emerge: **inheritance** and **composition**. While inheritance allows classes to inherit properties and behaviors from parent classes, composition promotes building complex classes through smaller, specialized classes. This investigation aims to delve deeper into the definitions, advantages, and disadvantages of both strategies, along with practical examples. The research further explores their implications on design flexibility, code reuse, and their relationship with key concepts such as **polymorphism**.

1. Introduction to Inheritance

Inheritance is a cornerstone of OOP that fosters code reuse by establishing hierarchical relationships among classes. For example, in a hypothetical company management application, one could create a base class called "Employee" and derive subclasses such as "Manager" and "Assistant." This structure allows for shared functionalities and properties, simplifying the codebase.

Advantages of Inheritance

- **Code Reusability**: Allows for the reuse of existing code, reducing redundancy.
- **Logical Hierarchy**: Encourages a clear organizational structure, making it easier to understand relationships among classes.
- **Ease of Maintenance**: Changes made in the parent class can propagate to child classes, easing maintenance.

Disadvantages of Inheritance

- **Rigidity**: Deep inheritance hierarchies can become cumbersome and inflexible, leading to challenges in modifying or extending functionality.
- **Fragile Base Class Problem**: Alterations in a parent class may unintentionally affect derived classes, causing unexpected behavior.
- **Limited Flexibility**: Inheritance tightly couples classes, making it harder to change implementations without affecting dependent classes.

2. Introduction to Composition

Composition offers a different approach, advocating for the construction of classes by combining simpler, reusable components. For instance, in a vehicle simulation project, rather than inheriting

from a generic "Vehicle" class, one could define components like "Engine," "Wheels," and "Brakes," which can then be assembled into various types of vehicles. This design fosters greater flexibility and modularity.

Advantages of Composition

- **Modularity**: Encourages the development of independent components that can be reused across different classes.
- **Flexibility**: Components can be easily added, removed, or replaced without affecting the overall system.
- **Encapsulation**: Promotes better encapsulation as each component can manage its own behavior and state.

Disadvantages of Composition

- **Complexity**: May introduce complexity when managing interactions among various components.
- **Overhead**: More classes and objects may lead to increased overhead in terms of performance and memory usage.

3. When to Use Inheritance vs. Composition

Inheritance: Typical Use Cases

Inheritance is best suited for scenarios where there is a clear "is-a" relationship. For instance, a "Student" is a type of "Person." This clear relationship justifies the use of inheritance, allowing for shared behaviors and properties that are common across subclasses.

Composition: Typical Use Cases

On the other hand, composition excels in scenarios where objects need to be more modular and interchangeable. For example, in a video game, characters may have components such as "Inventory," "Health," and "Abilities," which can vary independently. This allows for dynamic behavior changes without the constraints of an inheritance hierarchy.

4. Practical Examples

In a school management system, I would leverage **inheritance** to establish a structure among "Person," "Teacher," and "Student," facilitating clear relationships. However, in a gaming project, I

would opt for **composition** to define characters with dynamic attributes, allowing for changes in inventory or abilities without disrupting the core character logic.

5. Relationship with Polymorphism

Polymorphism is a powerful feature in OOP that allows for treating objects of different classes as objects of a common superclass. Inheritance naturally supports polymorphism, enabling methods to be overridden in subclasses, providing specific implementations while adhering to a common interface. Conversely, with composition, polymorphism can be achieved by utilizing interfaces or abstract classes for the individual components, allowing them to be interchangeable within a broader context.

In conclusion, the debate between inheritance and composition is nuanced and context-dependent. While inheritance provides a clear framework for establishing relationships between classes, it can introduce rigidity and maintenance challenges. Composition, in contrast, offers a more flexible and modular approach, fostering scalability and adaptability in evolving systems. Ultimately, the choice between these two strategies should align with the specific requirements of the project at hand, ensuring that the design remains robust, maintainable, and adaptable to change.