Samuel Edet Ukim

OOP HW6

Part A: Conceptual Questions

Composition vs. Aggregation

- **Composition:** In my own words, composition is a "has-a" relationship where one object (the "parent") exclusively owns and manages the lifecycle of another object (the "child"). The child object cannot exist independently of the parent, and when the parent is destroyed, the child is also typically destroyed.
    - **Concise Example:** A `CPU` is a component of a `Computer`. The `CPU` is created when the `Computer` is created and generally ceases to exist in the context of that `Computer` when the `Computer` is discarded.
- **Aggregation:** In my own words, aggregation is a "has-a" relationship where one object (the "container") has a reference to another object (the "part"), but the part can exist independently of the container. The container uses the part, but it doesn't exclusively own it, and the part's lifecycle is not strictly tied to the container's.
    - **Concise Example:** A `Library` has many `Book` objects. However, a `Book` can exist in another library or be owned by an individual. The `Library` uses `Book` objects but doesn't exclusively own their existence.
- **Stronger form of ownership:** Composition implies a stronger form of ownership because the parent object controls the creation and destruction of the child object.

When to Use

- **Composition over Inheritance:** Composition is more appropriate than inheritance in a UI scenario when building a complex UI element from smaller, distinct components that have their own behaviors. For example, a `UserProfilePage` might be composed of a `ProfileHeader`, a `FriendsList`, and a `PostFeed`. These components have different functionalities and lifecycles tightly bound to the `UserProfilePage`. Inheritance wouldn't be a natural fit as a `UserProfilePage` isn't inherently a specialized type of `ProfileHeader` or `FriendsList`.

- **Aggregation Sufficient:** Aggregation is sufficient in a banking scenario where a `Customer` has one or more `Account` objects. The `Account` objects can exist independently (e.g., a customer might close their account at one bank and open one at another). The `Customer` object uses the `Account` objects but doesn't exclusively own their existence or lifecycle. It's a looser coupling because the `Account` can exist even if the `Customer` record is removed, or the `Customer` can exist without any active accounts.

Differences from Inheritance

- Composition and aggregation represent "has-a" relationships, where one object contains or uses another. This differs from the "is-a" relationship implied by inheritance, where a derived class is a specialized type of its base class, inheriting its properties and behaviors. Inheritance focuses on code reuse through specialization, while composition/aggregation focuses on building complex objects by combining simpler, independent or tightly-coupled parts.

- An OOP design might favor composition over inheritance in certain cases to avoid the rigid hierarchies and potential issues of the "fragile base class problem" associated with inheritance. Composition promotes flexibility by allowing the behavior of an object to be determined at runtime by the objects it contains, and it leads to looser coupling, making the system easier to maintain and extend without deep, potentially brittle inheritance trees.

Real-World Analogy

- **Real-Life System:** A personal computer.
    - **Composition:** A computer *has a* motherboard. The motherboard is an integral part of the computer and is generally created and disposed of with the computer.
    - **Aggregation:** A computer *uses a* keyboard. The keyboard can exist independently of the computer and can be used with other computers. The computer has a relationship with the keyboard, but doesn't own its entire lifecycle.
- **Why these distinctions matter in code:** These distinctions matter in code because they dictate how objects are related and how their lifecycles are managed. Choosing the correct relationship impacts the coupling between classes, the flexibility of the design, and how memory and resources are handled. Using composition when strong ownership is needed ensures that related parts are managed together, while using aggregation allows for more independent and reusable components. Misunderstanding these relationships can lead to memory leaks, unexpected behavior, or overly rigid and difficult-to-maintain code.

Part B: Minimal Class Example (C++)

C++

```
#include <iostream>
#include <string>

class Address {
public:
    std::string street;
    std::string city;

    Address(std::string s, std::string c) : street(s),
city(c) {
        std::cout << "Address created: " << street << ", "
<< city << std::endl;
    }

    ~Address() {
        std::cout << "Address destroyed: " << street << ",
" << city << std::endl;
    }

    void displayAddress() const {
        std::cout << "Street: " << street << ", City: " <<
city << std::endl;
    }
};

class Person {
private:
    std::string name;
    Address homeAddress; // Composition: Person owns and
manages the Address

public:
    Person(std::string n, std::string s, std::string c) :
name(n), homeAddress(s, c) {
        std::cout << "Person created: " << name <<
std::endl;
    }

    ~Person() {
        std::cout << "Person destroyed: " << name <<
std::endl;
    }

    void displayPersonInfo() const {
        std::cout << "Name: " << name << ", Lives at: ";
        homeAddress.displayAddress();
    }
};

int main() {
    Person john("John Doe", "123 Main St", "Anytown");
    john.displayPersonInfo();
    // When john goes out of scope, his Address
(homeAddress) will also be destroyed.
    return 0;
}
```
Part B: Emphasis on Lifecycle Dependence (Composition)

In this example, the `Person` class has a member variable `homeAddress` of type `Address` directly within it. This demonstrates composition. The `Address` object is created when the `Person` object is created (in the `Person`'s constructor). The lifecycle of the `Address` object is tied to the `Person` object. When the `Person` object `john` is destroyed at the end of the `main` function, the destructor of `Person` is called, and as part of the object's destruction, the destructor of its member `homeAddress` is also automatically called. The `Address` doesn't exist independently of the `Person` in this design.

Part C: Reflection & Short Discussion

- **Ownership & Lifecycle:**
    - **Composition:** In a composition relationship, when the "parent" object is destroyed or deallocated, the "child" object that it owns is also typically destroyed or deallocated. This is because the parent has exclusive ownership and manages the child's existence.
    - **Aggregation:** In an aggregation relationship, the "child" object can continue to exist independently even if the "parent" object is destroyed or deallocated. The parent object merely holds a reference to the child, but doesn't control its lifecycle.
- **Advantages & Pitfalls:**
    - **Advantage of Composition:** Composition provides strong control over the lifecycle of contained objects. This can simplify resource management and ensure that related objects are cleaned up together, preventing resource leaks or dangling references.
    - **Potential Pitfall of Wrongly Using Composition:** If composition is used when a looser coupling (aggregation) is more appropriate, it can lead to unnecessary dependencies and restrictions on the reusability and independent lifecycle management of the "child" objects. For example, tightly coupling a `Student` object to a specific `Classroom` object through composition might make it difficult to represent scenarios where a student can exist outside of that specific classroom or move between classrooms.
- **Contrast with Inheritance:** "Has-a" relationships (composition and aggregation) focus on how objects are *related* and *interact* by containing or using each other. This contrasts with the "is-a" relationship of inheritance, which focuses on creating a *hierarchy* of classes where derived classes *specialize* the behavior and structure of a base class. "Has-a" emphasizes building complex behavior by combining objects, while "is-a" emphasizes code reuse through type specialization.

- **Why avoid inheritance for composition/aggregation:** We might avoid inheritance in situations that can be solved by composition or aggregation because inheritance can lead to tightly coupled and less flexible designs, especially with deep inheritance hierarchies. Composition and aggregation promote looser coupling and greater flexibility by allowing objects to collaborate without being bound by a strict "is-a" relationship. This often leads to more maintainable and reusable code.