Name: Esther Ajuwon

Composition & Aggregation

PART A: Conceptual Questions

1. Composition vs. Aggregation

Composition is a "has-a" connection in which one object completely owns another. If the parent is destroyed, so are the components.
Aggregation is a "has-a" relationship, but with shared ownership, allowing the child to exist independently of the parent.
Composition implies a stronger form of ownership than aggregation.

2. When to Use

Scenario where composition is more appropriate than inheritance
In a game, you have a PLayer class that requires various abilities (such as flying, shooting, and healing). Instead of using inheritance (which limits flexibility), use composition to give Player objects abilities. There is no strict hierarchy of classes, so you can mix and combine skills. It maintains classes loosely connected and encourages adaptability and reuse.
Scenario where aggregation is sufficient
A Bank class has a list of Customer objects; however, customers can exist independently and belong to other banks or systems. The Bank has references, but not actual ownership. If the Bank is destroyed, customer objects may still remain. This is a loser coupling that symbolizes partial ownership—suitable for things with longer lifespans or relationships.

3. Differences from Inheritance

How Composition/Aggregation Differ from Inheritance

- Composition and Aggregation imply a “has-a” relationship.

For example: A Car *has an* Engine (composition); a Team has Players(aggregation)

- Inheritance implies an “is-a” relationship

For example: A Dog *is a* Animal.

So, inheritance models identity and common behavior, whereas composition/aggregation models functionality and part-whole relationship.

In object-oriented design, composition is frequently preferred over inheritance because it provides greater flexibility, better encapsulation, and decreases the likelihood of difficulties such as the weak base class problem, in which modifications to a parent class unintentionally affect all subclasses. Composition also enables for more modular designs, as a class can integrate and combine behaviors from multiple components, which is very valuable in complicated or changing systems.

4. Real-World Analogy

A university system is a real-world example of both composition and aggregation: a department has courses through composition, which means the courses have an unbreakable connection to the department and would not exist independently; if the department ceases to exist, the courses go with it. Professors in the same department are aggregated, which means they can exist independently, teach in numerous departments, and are not destroyed when a department closes. These distinctions are important in programming because they define ownership and lifecycle responsibilities. Composition implies tight ownership, requiring the parent to govern the child's lifetime, whereas aggregation denotes loose coupling and shared or independent lifetimes. Understanding this helps to avoid memory leaks, dangling pointers, and design ambiguity—particularly in C++, where manual resource management is essential for developing dependable and stable software.

PART B: UML Diagram

```
+-------------------+           +--------------------+
|  Department  |◄◆-------|    Course       |
|--------------------| 1      1..*|---------------------|
| - name: string  |            | - title: string     |
| - code: string   |            | - credits: int      |
+--------------------+          +--------------------+
    Composition

+--------------------+          +-----------------------+
|  Department   |◄◇-------|  Professor       |
|--------------------| 1      0..*|-----------------------|
| - name: string  |             | - name: string    |
| - code: string   |             | - employeeId: int|
+-------------------+           +-----------------------+
    Aggregation
```

The relationship between Department and Course is composition because a course is inextricably linked to its department; if the department is removed, its courses are no longer relevant and are erased alongside it, demonstrating strong ownership. In contrast, the relationship between Department and Professor is aggregation since professors can operate across numerous departments and continue to exist even if one department is disbanded, showing looser coupling and shared responsibility.

PART C: Reflection & Short Discussion

1. Ownership & Lifecycle

In a composition relationship, when the parent object is destroyed or deallocated, the child object is automatically destroyed—this demonstrates strong ownership, in which the parent has complete control over the child's lifecycle.
In an aggregation connection, the kid object exists independently because it is not owned by the parent. The parent may refer to or use the kid, but it is controlled externally and can be shared with other objects or live beyond the parent's lifetime.

2. Advantages & Pitfalls

Advantage

Composition provides precise control over object lifecycles—when a parent is destroyed, its own components are automatically cleaned away, eliminating memory leaks and simplifying resource management.
Pitfall:
If you use composition instead of aggregation incorrectly, you may mistakenly remove shared or independently managed objects, resulting in data loss, dangling pointers, or violation of object ownership, particularly in systems where reuse and independence are critical.

3. Contrast with Inheritance

"Has-a" relationships, expressed by composition and aggregation, show how one object contains or refers to another, allowing for flexible and modular architectures with components that may be reused or replaced. In contrast, a "is-a" relationship, represented by inheritance, implies that a class is a specialized version of another, firmly binding the child to the parent's structure and behavior.
We frequently avoid inheritance when composition or aggregation might handle the problem because inheritance can result in rigid, unstable hierarchies, decreased code reuse, and unwanted side effects from changes in base classes. Composition encourages more flexibility, maintainability, and explicit ownership of responsibilities.