Questions

▼ Objects have crisply defined boundaries

Objects with crisply defined boundaries mean that each object is self-contained, encapsulating its data and behavior and providing a clear and controlled interface for interaction with other objects, contributing to the principles of encapsulation and abstraction in object-oriented programming.

▼ Delegation and Composition

- Composition is a design principle where one class (the container or composite class) contains an instance of another class (the component or delegate class) as a part of its internal structure. The container class has a direct reference to the component class and can invoke its methods to achieve specific functionality.

  Delegation is a broader concept that involves one object (the delegating object) passing on a task or responsibility to another object (the delegate) to perform on its behalf. In the context of OOP, delegation often involves composition, but it can also involve other mechanisms.

- In composition, the focus is on creating complex objects by assembling simpler objects as their parts. The container class directly contains an instance of the component class and uses it to provide specific functionality. Composition is a fundamental part of object-oriented design and is used to achieve code reuse and modular design.

  Delegation is a more abstract concept that encompasses various ways of assigning responsibility to other objects. While composition is one form of delegation (using a component object to perform part of the task), delegation can also involve other mechanisms, such as method delegation (one object invokes a method on another object to perform a task), event delegation (routing events to event handlers), and more.

▼ Re exporting and delegation

Delegation sometimes involves re-exporting.

"Re-exporting" in the context of delegation means that the delegating object doesn't just pass the task to the delegate but also exposes the delegate's capabilities or

results as its own. In other words, it allows the delegating object to "re-export" the functionality provided by the delegate as if it were its own.

Delegation with re-exporting:

```
#include <iostream>

class DelegateClass {
public:
    std::string performTask() {
        return "Task completed by DelegateClass";
    }
};

class DelegatingClass {
private:
    DelegateClass delegate;

public:
    std::string doTask() {
        // Delegating the task to the delegate
        std::string result = delegate.performTask();
        return result;  // Re-exporting the result as if it's the DelegatingClass's own
    }
};

int main() {
    DelegatingClass delegator;
    std::string result = delegator.doTask();
    std::cout << result << std::endl;  // Output: "Task completed by DelegateClass"
    return 0;
}
```

Note that there is no re-exporting in case of composition:

```
#include <iostream>

class ComponentClass {
public:
    std::string performTask() {
        return "Task completed by ComponentClass";
    }
};

class ContainerClass {
private:
    ComponentClass component;

public:
    std::string doTask() {
        // Delegating the task to the component
        std::string result = component.performTask();
        return result;  // The result is used internally but not re-exported
```

```
    }
};

int main() {
    ContainerClass container;
    std::string result = container.doTask();
    std::cout << result << std::endl;  // Output: "Task completed by ComponentClass"
    return 0;
}
```

▼ Without polymorphism, developer ends up writing code with large number of switch and case statements

```
enum ShapeType { Circle, Square, Triangle };

class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    void draw() override {
        // Draw a circle
    }
};

class Square : public Shape {
public:
    void draw() override {
        // Draw a square
    }
};

class Triangle : public Shape {
public:
    void draw() override {
        // Draw a triangle
    }
};

int main() {
    ShapeType type = /* Some logic to determine the type */;
    Shape* shape;

    switch (type) {
        case Circle:
            shape = new Circle();
            break;
        case Square:
            shape = new Square();
            break;
```

```
            case Triangle:
                shape = new Triangle();
                break;
        }

        shape->draw();  // Draw the selected shape

        // Clean up memory
        delete shape;

        return 0;
    }
```

▼ Mix-in and aggregate classes

Aggregate classes, also known as composite classes, are classes that combine or aggregate the functionality of multiple other classes or objects. They serve as a container for other objects, allowing them to work together as a single unit.

Mixin classes are classes that provide a specific set of behaviors or functionalities that can be "mixed in" with other classes. They are often used to add common or reusable functionality to multiple classes without the need for inheritance hierarchies.

▼ Coupling is the measure of level of accessibility

Coupling measures the level of interdependence and interaction between software components, while accessibility relates to how easily components can interact.

- **Low Coupling and High Accessibility:** When modules have low coupling, it often means that they are designed to interact through well-defined and minimal interfaces. This can lead to high accessibility, as it's clear how components should interact, and there are fewer dependencies to manage.
- **High Coupling and Low Accessibility:** Conversely, in systems with high coupling, it can be challenging to access or modify one component without affecting others. This can lead to lower accessibility, as changes may have unintended consequences

Reducing coupling often leads to more accessibility.

▼ Association

Association is the semantic relationship between two classes. Association represents a generic, bi-directional relationship between two or more classes or objects. It is a simple, loosely coupled relationship that doesn't imply ownership.

A teacher and a student are associated, but neither owns the other, and there can be many students associated with one teacher.

▼ Association can be bidirectional as well as unidirectional

Bidirectional (mutual awareness and interaction) or unidirectional (one-way awareness).

- In a bidirectional association, two classes or objects are aware of each other and can reference each other.
- Changes made in one class or object can affect the other, and both classes have a relationship with each other.
- Person and address or student and school

```
class Student {
    // ...
    School school;
}

class School {
    // ...
    List<Student> students;
}
```

- In a unidirectional association, one class or object is aware of the other, but the reverse is not true.
- Changes made in one class or object can affect the other, but the second class or object doesn't have a direct relationship with the first.

```
class Library {
    // ...
    List<Book> books;
}

class Book {
    // ...
}
```

▼ Binary association

A binary association in object-oriented modeling represents a relationship between two classes or objects, and it can be either bidirectional or unidirectional. A binary association specifically involves two classes or objects, hence the term "binary."

▼ Aggregation and Composition

- Aggregation is a specific type of association that represents a "whole-part" relationship between classes or objects. It implies that one class is a part of another class and has a weaker relationship than composition.

  Example : A university and its departments have an aggregation relationship, as a university consists of multiple departments, but departments can exist independently and can be shared among universities.

- Composition is a strong form of aggregation where one class (the whole) is composed of other classes (the parts), and the parts are tightly bound to the whole. It implies a "whole-part" relationship with strong ownership.

  A car and its engine have a composition relationship. The engine is an integral part of the car, and when the car is scrapped, the engine typically goes with it.

Association represents a generic relationship between classes or objects with no specific ownership or multiplicity.

Aggregation represents a "whole-part" relationship with weaker ownership and is often used for modeling situations where parts can be shared among multiple wholes.

Composition also represents a "whole-part" relationship but with strong ownership and lifecycle management, where the parts are tightly bound to the whole and usually destroyed when the whole is destroyed.

▼ Destroying the part, and effect on whole in Aggregation and Composition

- In an aggregation relationship, the part is considered a component of the whole, but it does not have strong ownership ties to the whole.
- When you destroy a part in an aggregation, it does not necessarily lead to the destruction of the whole.
- The part can exist independently of the whole and may be shared among multiple wholes.
- Deleting a part in aggregation typically involves setting the reference to null or removing it from the list of parts, but the whole remains unaffected.

```
class University {
    List<Department> departments;
}

class Department {
    // ...
}
```

- In a composition relationship, the part is considered an integral component of the whole, and there is a strong ownership relationship.
- When you destroy a part in a composition, it typically leads to the destruction of the whole.
- The whole manages the lifecycle of its parts, and when the whole is destroyed or deleted, it takes care of destroying its parts as well.
- Deleting a part in composition involves removing it from the whole, and the part's resources (memory, objects, etc.) are typically released or destroyed as part of this process.

```
class Car {
    Engine engine;
}

class Engine {
    // ...
}
```

▼ Destroying the whole and effect on part in Aggregation and Composition

- In aggregation, the part can often continue to exist independently even if the whole is destroyed. The part may have its own lifecycle that is not tightly bound to the whole.

  Example: If a library (whole) is destroyed, the books (parts) in its collection can still exist, and you can potentially use or move those books to another library or location.

- In composition, the whole takes strong ownership of its parts, and the part's lifecycle is tightly bound to the whole. When the whole is destroyed, it typically ensures the destruction or deallocation of its parts.

  Example: If a car (whole) is destroyed, the engine (part) is often destroyed or rendered unusable because the engine is an integral part of the car, and the

car is responsible for managing the engine.

▼ Note that Library-book system is a example of association and university-department is an example of Aggregation Yet both have same structure in code

```
class Library {
    List<Book> books; // Association with books
}

class Book {
    // ...
}
```

```
class University {
    List<Department> departments; // Aggregation of departments
}

class Department {
    // ...
}
```

The distinction between whether a relationship should be modeled as an association, aggregation, or composition often depends on the specific semantics and requirements of the system being modeled. While the structure of the classes may appear similar in both cases, the choice between association and aggregation (or composition) reflects the intended relationship and the level of ownership and responsibility between the classes.

▼ Aggregation does not guarantee physical containment but composition does it

If there is whole/part relationship, then there will be aggregation always. As the aggregation is loosely coupled association between classes, it does not restrict the parts to be entirely within the whole. Parts may be shared across others.

Aggregation is a modeling concept that represents a logical or conceptual relationship between objects without imposing strict rules about their physical containment or management

In object-oriented modeling, composition represents a strong "whole-part" relationship between classes or objects, and one of its defining characteristics is

that the whole (the composite) has exclusive ownership and responsibility for the creation, management, and destruction of its parts (components).

▼ Composition is a special case of Association

Both composition and aggregation represent "whole-part" relationships between classes or objects, but they differ primarily in terms of the strength of ownership and the lifecycle management of the parts by the whole. Composition is the stronger form of aggregation, and it imposes stricter rules on the relationship.

▼ Quality class and object design