Jadhav, Ajay - Dell Team

[Company name]  [Company address]

My OOPs Notes

Contents

[Design abstract classes or interface classes 2](#_Toc136183746)

[Use abstract classes when: 2](#_Toc136183747)

[Use interfaces when: 2](#_Toc136183748)

[Conclusion: 2](#_Toc136183749)

[Polymorphism 3](#_Toc136183750)

[Inheritance Polymorphism: 3](#_Toc136183751)

[Relationships: 3](#_Toc136183752)

# Design abstract classes or interface classes

The decision to design abstract classes or interface classes depends on the specific requirements and design goals of your application. Here are some considerations to help you make the choice:

## Use abstract classes when:

* You want to provide a common base implementation for a group of related classes.
* You want to define default behavior for certain methods.
* You need to access protected members or fields within the hierarchy of related classes.
* You want to create a class hierarchy that represents a "is-a" relationship, where subclasses are more specific types of the abstract class.
* You anticipate the need to add new methods or members in the future, while still providing a default implementation for existing methods.

## Use interfaces when:

* You want to define a contract or a set of method signatures that classes should implement, regardless of their inheritance hierarchy.
* You need to support multiple inheritance-like behavior, where a class can implement multiple interfaces.
* You want to enable loose coupling between classes, allowing different implementations to be easily swapped.
* You have unrelated classes that need to share common behavior.
* You want to enforce a certain level of abstraction and ensure adherence to a specific interface.

### Conclusion:

It's important to note that abstract classes and interfaces can be used together in a design. For example, you might have an abstract class providing a base implementation and implementing an interface that defines additional behavior. This combination can provide flexibility while maintaining a clear contract through interfaces.

Ultimately, the decision between abstract classes and interfaces depends on the specific needs and goals of your application's architecture, the relationship between classes, and the level of flexibility and abstraction required.

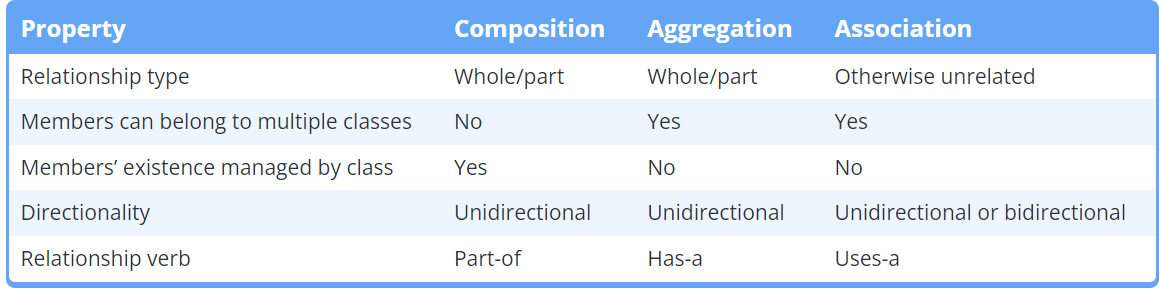
Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common superclass. It enables objects to exhibit different behaviors based on their specific class type or the context in which they are used.

Polymorphism can be achieved through two main mechanisms: inheritance and interfaces.

## Inheritance Polymorphism:

Inheritance polymorphism occurs when a subclass inherits from a superclass and overrides or extends its methods. It allows objects of the subclass to be used wherever objects of the superclass are expected. The specific implementation of the overridden methods in the subclass is executed at runtime based on the actual object type.

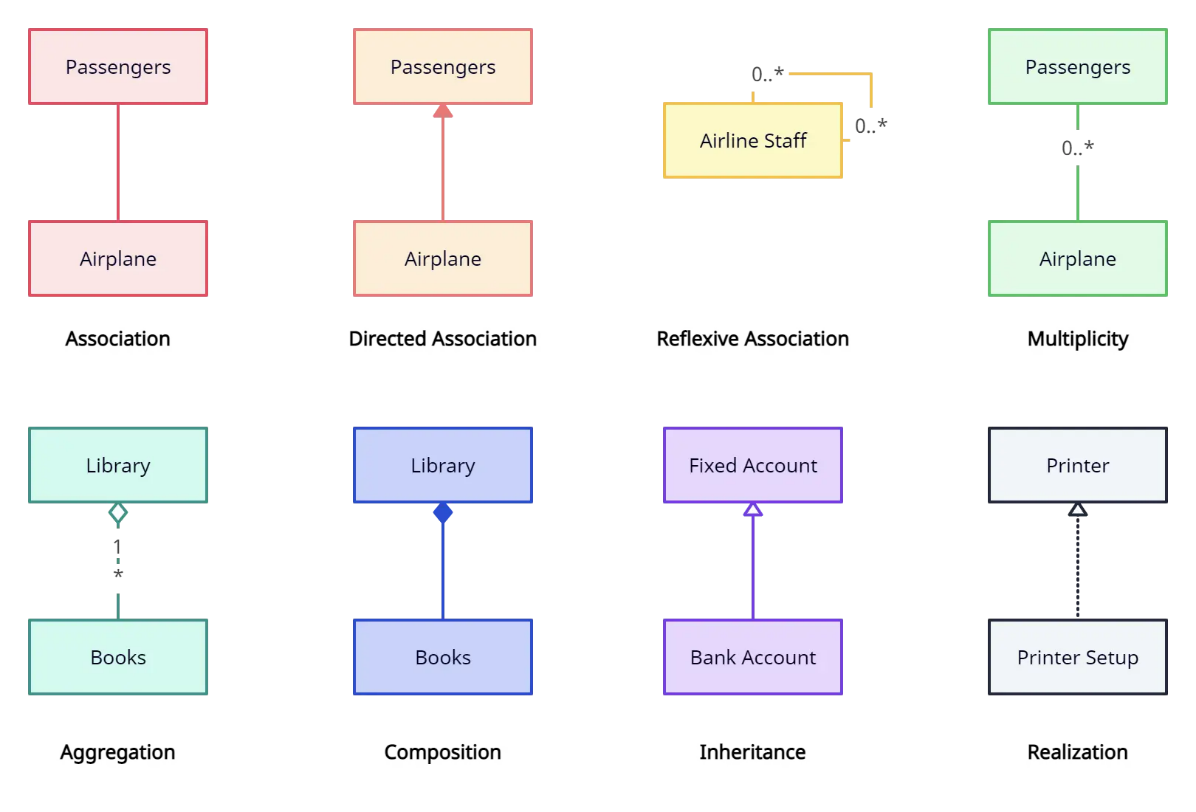
# Relationships:



Remember association with verb relationship among two classes like Dog eats, Driver driving car

a reflexive association relationship between the University class and the Student class. This means that each Student object is associated with a University object, and each University object maintains a collection of Student objects.

## UML diagrams:



## Code Examples :

class Car {

private:

string make;

Engine engine; // Composition relationship

Driver\* driver; // Association relationship

public:

Car(const string& carMake, const string& engineType, Driver\* carDriver)

: make(carMake), engine(engineType), driver(carDriver) {}

void startCar() {

cout << "Starting the " << make << " car." << endl;

engine.start();

driver->driveCar(make);

}

// Aggregate class

class Car {

private:

string make;

Engine\* engine; // Aggregation relationship

public:

Car(const string& carMake, Engine\* carEngine) : make(carMake), engine(carEngine) {}