Contents

[Open/Closed Principle (OCP): 1](#_Toc136259344)

[Liskov Substitution Principle (LSP): 3](#_Toc136259345)

[Interface Segregation Principle (ISP): 3](#_Toc136259346)

[Dependency Inversion Principle (DIP): 3](#_Toc136259347)

[3 ways: 4](#_Toc136259348)

[Dependency Injection (DI): 4](#_Toc136259349)

[Abstract Factory Pattern: 4](#_Toc136259350)

[Inversion of Control (IoC) Containers: 4](#_Toc136259351)

[DRY Don’t repeat yourself 4](#_Toc136259352)

[Here are a few guidelines to apply the DRY principle in the LLD design using OOP: 4](#_Toc136259353)

[Encapsulate Reusable Logic: 4](#_Toc136259354)

[Use Inheritance: 4](#_Toc136259355)

[Extract Common Functionality: 5](#_Toc136259356)

[Apply Design Patterns: 5](#_Toc136259357)

[Use Composition over Inheritance: 5](#_Toc136259358)

[Modularize Code: 5](#_Toc136259359)

[where to use Inheritance over Composition in c++ 5](#_Toc136259360)

["Is-a" Relationship: 5](#_Toc136259361)

[Polymorphism: 6](#_Toc136259362)

[Reusing Base Class Functionality: 6](#_Toc136259363)

[Inheriting Interface or Abstract Base Classes: 6](#_Toc136259364)

["YAGNI" (You Aren't Gonna Need It): 6](#_Toc136259365)

[Curly's Law: 6](#_Toc136259366)

[Keep It Simple Stupid (KISS) 6](#_Toc136259367)

[Don't make me think 7](#_Toc136259368)

# Open/Closed Principle (OCP):

the **Account** class is open for extension but closed for modification.

The **InterestCalculato**r interface allows for different interest calculation strategies. The Account class depends on the abstraction (InterestCalculator) rather than concrete implementations. It can be easily extended to accommodate new interest calculation strategies without modifying the existing Account class.

class **InterestCalculator** {

public:

virtual double calculateInterest(double balance) const = 0;

};

class **FixedDepositInterestCalculator** : public **InterestCalculator** {

public:

double calculateInterest(double balance) const override {

// logic to calculate interest for fixed deposit account

}

};

class **SavingsAccountInterestCalculator** : public InterestCalculator {

public:

double calculateInterest(double balance) const override {

// logic to calculate interest for savings account

}

};

class **Account** {

private:

string accountNumber;

double balance;

**InterestCalculator\* interestCalculator;**

// ...

public:

double calculateInterest() const {

**return interestCalculator->calculateInterest(balance);**

}

// ...

};

# Liskov Substitution Principle (LSP):

derived classes should be able to be used in place of their base classes without causing unexpected behavior.

For example, class Bird has fly method, but ostrich birds cannot fly causing this to create a new interface FlyClass.

# Interface Segregation Principle (ISP):

Just to create as many interfaces as possible.

Clients should not be forced to depend on interfaces they do not use. This principle suggests that interfaces should be fine-grained and focused on specific client requirements. Clients should not be required to depend on interfaces that contain methods they don't need, as this can lead to unnecessary coupling and dependencies.

the interfaces **(IAccount, IDepositAccount, IWithdrawalAccount)** are segregated based on their specific client requirements. The SavingsAccount class implements all three interfaces, but other account types can choose to implement only the relevant interfaces.

# Dependency Inversion Principle (DIP):

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Both should be loosely coupled.

Base class should not depend on concrete subtypes.

## 3 ways:

### Dependency Injection (DI):

Constructor Injection: Dependencies are passed to a class through its constructor.

Setter Injection: Dependencies are set using setter methods of a class.

Interface Injection: Dependencies are injected through specialized methods defined by an interface.

### Abstract Factory Pattern:

An abstract factory interface or class is used to create instances of concrete classes that implement a common interface or inherit from a common base class. This allows the client code to depend on the abstract factory rather than concrete classes directly.

### Inversion of Control (IoC) Containers:

IoC containers manage the creation and resolution of dependencies automatically. They provide mechanisms to register dependencies and their configurations, and the container handles the creation and injection of instances when needed.

# DRY Don’t repeat yourself

## Here are a few guidelines to apply the DRY principle in the LLD design using OOP:

Encapsulate Reusable Logic: Identify common functionalities or algorithms that are used in multiple parts of your design. Encapsulate them within reusable components such as classes, modules, or functions. This way, you can avoid duplicating the same logic in multiple places.

Use Inheritance: Inheritance is a key feature of OOP that allows you to create a hierarchy of classes where child classes inherit properties and methods from their parent classes. By defining common behavior in a base class and extending it with specific behavior in derived classes, you can reuse code across related classes.

Extract Common Functionality: If you find similar code segments in different classes, consider extracting the common functionality into separate utility methods or helper classes. This way, you can centralize the logic and use it wherever needed, reducing duplication.

Apply Design Patterns: Design patterns provide proven solutions to common design problems. They often promote reusable and maintainable code structures. Familiarize yourself with various design patterns, such as Singleton, Factory, Strategy, or Observer, and apply them where appropriate to avoid repeating design decisions.

Use Composition over Inheritance: While inheritance can be useful for code reuse, it can also lead to inflexible designs and inheritance hierarchies that become difficult to maintain. In such cases, prefer composition, where classes are composed of instances of other classes, to achieve flexibility and code reuse without deep inheritance hierarchies.

Runtime Configuration: **Composition enables runtime configuration of objects by dynamically changing their behavior or components.** This is particularly useful in scenarios where you want to switch or modify the behavior of a class at runtime without modifying its structure or using inheritance.

Modularize Code: Split your codebase into modular components that are responsible for specific tasks. Each module should have a clear responsibility and minimal dependencies on other modules. This way, you can develop and maintain individual modules independently, reducing the need for repeating code across different modules.

By adhering to the DRY principle in your LLD design, you can create a more maintainable, flexible, and reusable codebase, which can lead to improved productivity, reduced bugs, and easier maintenance in the long run.

# where to use Inheritance over Composition in c++

In C++, there are certain scenarios where using inheritance is more appropriate than composition. Here are some situations where inheritance is commonly used:

"Is-a" Relationship: If there is an "is-a" relationship between two classes, where one class represents a specialized version of the other, inheritance is typically used. The derived class inherits properties and behaviors from the base class, extending or modifying them as needed.

Polymorphism: Inheritance is crucial for achieving polymorphic behavior. When you have a base class pointer or reference, you can use it to refer to objects of derived classes and achieve dynamic dispatch or late binding. This allows for more flexible and extensible code.

Reusing Base Class Functionality: If you have a set of classes that share common functionality or attributes, you can define them in a base class and derive specialized classes from it. This way, you avoid duplicating code and promote code reuse.

Inheriting Interface or Abstract Base Classes: Inheritance is often used to define interfaces or abstract base classes that provide a common interface for derived classes. This allows you to enforce a specific set of methods that derived classes must implement.

# "YAGNI" (You Aren't Gonna Need It):

The YAGNI principle suggests that you should not add functionality or features to your software until they are necessary. It encourages simplicity and avoiding speculative implementation of code that might be needed in the future but isn't currently required. This principle aims to prevent unnecessary complexity, wasted effort, and potential maintenance issues associated with implementing functionality that may never be used.

Curly's Law: Curly's Law is about choosing a single, clearly defined goal for any particular bit of code: Do One Thing.

Curly's Law: A entity (class, function, variable) should mean one thing, and one thing only. It should not mean one thing in one circumstance and carry a different value from a different domain some other time. It should not mean two things at once. It should mean One Thing and should mean it all of the time.

# Keep It Simple Stupid (KISS)

The KISS principle states that most systems work best if they are kept simple rather than made complicated; therefore, simplicity should be a key goal in design, and unnecessary complexity should be avoided.

Simple code has the following benefits:

less time to write

less chances of bugs

easier to understand, debug and modify

Do the simplest thing that could possibly work.

## Don't make me think

Code should be easy to read and understand without much thinking. If it isn't then there is a prospect of simplification.