Contents

[Open/Closed Principle (OCP): 1](#_Toc136190619)

[Liskov Substitution Principle (LSP): 2](#_Toc136190620)

[Interface Segregation Principle (ISP): 3](#_Toc136190621)

[Dependency Inversion Principle (DIP): 3](#_Toc136190622)

[3 ways: 3](#_Toc136190623)

[Dependency Injection (DI): 3](#_Toc136190624)

[Abstract Factory Pattern: 3](#_Toc136190625)

[Inversion of Control (IoC) Containers: 3](#_Toc136190626)

# 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.