# Programming

Design patterns and design principles

Design patterns and design principles, specifically the SOLID principles, are two different concepts in software development, but they are related and complementary to each other.

Design Principles (SOLID): SOLID is an acronym that represents a set of five design principles for writing maintainable and scalable software. These principles were introduced by Robert C. Martin (Uncle Bob) and have become widely adopted in the industry. Each letter in SOLID stands for a specific principle:

Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one responsibility or purpose.

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. In other words, you should be able to add new functionality without modifying existing code.

Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. This principle promotes the idea of small, cohesive interfaces.

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

These principles guide developers in writing code that is modular, flexible, and easy to maintain and extend over time.

Design Patterns: Design patterns are reusable solutions to common software design problems. They provide templates or blueprints for solving specific design issues in a structured and proven way. Design patterns capture best practices and design principles that have been discovered and refined over time. They help in achieving better code organization, separation of concerns, and code reuse.

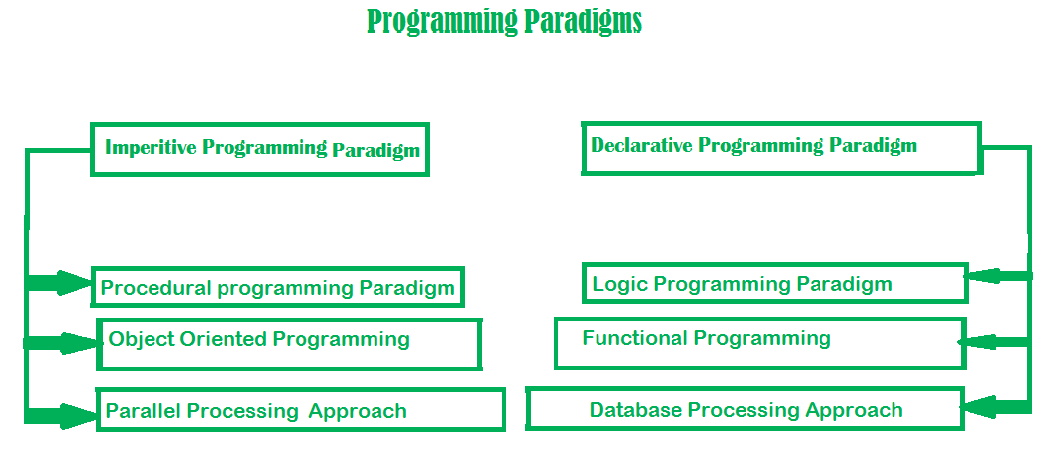
Design patterns are classified into several categories, including creational patterns, structural patterns, and behavioral patterns. Examples of design patterns include the Singleton pattern, Factory pattern, Observer pattern, and many more.

Design patterns can be used to implement the SOLID principles and make the design of a software system more robust and maintainable. By using design patterns, developers can leverage proven solutions to design problems rather than reinventing the wheel.

In summary, design principles (SOLID) provide guidelines for writing clean and maintainable code, while design patterns offer reusable solutions to common design problems. Design patterns can be applied to implement the principles effectively and create well-designed software systems. Both concepts are essential for professional software development.

# Introduction of Programming Paradigms

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand. They are discussed below:



1. **Imperative programming paradigm:**   
   It is one of the oldest programming paradigms. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The focus is on how to achieve the goal. The paradigm consists of several statements and after execution of all the result is stored.

Advantage:

* Very simple to implement
* It contains loops, variables etc.

Disadvantage:

* Complex problem cannot be solved
* Less efficient and less productive
* Parallel programming is not possible

Examples of Imperative programming paradigm:

int marks[5] = { 12, 32, 45, 13, 19 } int sum = 0;

float average = 0.0;

for (int i = 0; i < 5; i++) {

    sum = sum + marks[i];

}

average = sum / 5;

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

1. **Procedural programming paradigm –**   
   This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

Examples of Procedural programming paradigm:

#include <iostream>

using namespace std;

int main()

{

    int i, fact = 1, num;

    cout << "Enter any Number: ";

    cin >> number;

    for (i = 1; i <= num; i++) {

        fact = fact \* i;

    }

    cout << "Factorial of " << num << " is: " << fact << endl;

    return 0;

}

1. **Object oriented programming –**   
   The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.

Advantages:

* Data security
* Inheritance
* Code reusability
* Flexible and abstraction is also present

Examples of Object Oriented programming paradigm:

import java.io.\*;

  class GFG {

    public static void main(String[] args)

    {

        System.out.println("GfG!");

        Signup s1 = new Signup();

        s1.create(22, "riya", "riya2@gmail.com", 'F', 89002);

    }

}

class Signup {

    int userid;

    String name;

    String emailid;

    char sex;

    long mob;

    public void create(int userid, String name,

                        String emailid, char sex, long mob)

    {

        System.out.println("Welcome to

                 GeeksforGeeks\nLets create your account\n");

        this.userid = 132;

        this.name = "Radha";

        this.emailid = "radha.89@gmail.com";

        this.sex = 'F';

        this.mob = 900558981;

        System.out.println("your account has been created");

    }

}

1. **Parallel processing approach –**   
   Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them. This approach seems to be like divide and conquer. Examples are NESL (one of the oldest one) and C/C++ also supports because of some library function.
2. **Declarative programming paradigm:**   
   It is divided as Logic, Functional, Database. In computer science the declarative programming is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic.It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code is doing. It just declares the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.
3. **Logic programming paradigms –**   
   It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.   
   In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

sum of two number in prolog:

predicates

sumoftwonumber(integer, integer)

clauses

sum(0, 0).

sum(n, r):-

n1=n-1,

sum(n1, r1),

r=r1+n

1. **Functional programming paradigms –**   
   The functional programming paradigms has its roots in mathematics and it is language independent. The key principle of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions. The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like Perl, JavaScript mostly uses this paradigm.
2. **Database/Data driven programming approach –**   
   This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

CREATE DATABASE databaseAddress;

CREATE TABLE Addr (

PersonID int,

LastName varchar(200),

FirstName varchar(200),

Address varchar(200),

City varchar(200),

State varchar(200)

);

A programming paradigm is a way to classify a programming language based on some features. Basically, a programming paradigm is a way to write programs.

Some languages use only 1 paradigm, while others can use multiple paradigms. For example, Haskell uses a single paradigm: functional programming. JavaScript uses multiple paradigms, as we can write programs using procedural programming, "object-oriented" programming (OOP for short), functional programming, or even reactive programming.

For single paradigm languages, since there's only 1 way of writing programs, it's easy to "stay on the track". For multi-paradigms languages, it's harder to stay on the same track, as the frontier between 2 paradigms depends solely on the developer writing the program.

**Types of programming languages**

1. **Procedural programming languages:**

A procedural language follows a sequence of statements or commands in order to achieve a desired output. Each series of steps is called a procedure, and a program written in one of these languages will have one or more procedures within it. Example - C and C++, Java, Pascal, BASIC.

1. **Functional programming languages:**

Rather than focusing on the execution of statements, functional languages focus on the output of mathematical functions and evaluations. Each function–a reusable module of code–performs a specific task and returns a result. The result will vary depending on what data you input into the function. Example – Scala, Erlang, Haskell, Elixir, F#

1. **Object-oriented programming languages:**

This type of language treats a program as a group of objects composed of data and program elements, known as attributes and methods. Objects can be reused within a program or in other programs. This makes it a popular language type for complex programs, as code is easier to reuse and scale. Example – Java, Python, PHP, C++, Ruby.

1. **Scripting languages:**

Programmers use scripting languages to automate repetitive tasks, manage dynamic web content, or support processes in larger applications. Example- PHP, Ruby, Python, bash

1. **Logic programming languages:**

Instead of telling a computer what to do, a logic programming language expresses a series of facts and rules to instruct the computer on how to make decisions. Example- Prolog, Absys, Data log.

# Functional Programming

**Side effect (computer science):**

In computer science, an operation, function, or expression is said to have a side effect if it modifies some state variable value(s) outside its local environment, which is to say if it has any observable effect other than its primary effect of returning a value to the invoker of the operation. Example side effects include modifying a non-local variable, modifying a static local variable, modifying a mutable argument passed by reference, performing I/O or calling other functions with side-effects.

**What is Functional Programming?**

Functional programming is a paradigm through which developers write programs using a combination of pure functions, which are developed in such a way that they don’t have side effects (more on that later). Functional languages emphasis on expressions and declarations rather than execution of statements. Therefore, unlike other procedures which depend on a local or global state, value output in FP depends only on the arguments passed to the function.

**Characteristics of Functional Programming:**

1. Functional programming method focuses on results, not the process.
2. Emphasis is on what is to be computed.
3. Data is immutable.
4. Functional programming Decompose the problem into ‘functions.
5. It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation.
6. It does not support iteration like loop statements and conditional statements like If-Else.

**History of Functional programming:**

1. The foundation for Functional Programming is Lambda Calculus. It was developed in the 1930s for the functional application, definition, and recursion
2. LISP was the first functional programming language. McCarthy designed it in 1960
3. In the late 70’s researchers at the University of Edinburgh defined the ML(Meta Language)
4. In the early 80’s Hope language adds algebraic data types for recursion and equational reasoning
5. In the year 2004 Innovation of Functional language ‘Scala.’

**Type of functional language :**

1. **Pure Functional Languages** − These types of functional languages support only the functional paradigms. For example − Haskell.
2. **Impure Functional Languages** − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

## Pure function

Pure functions have two important properties:

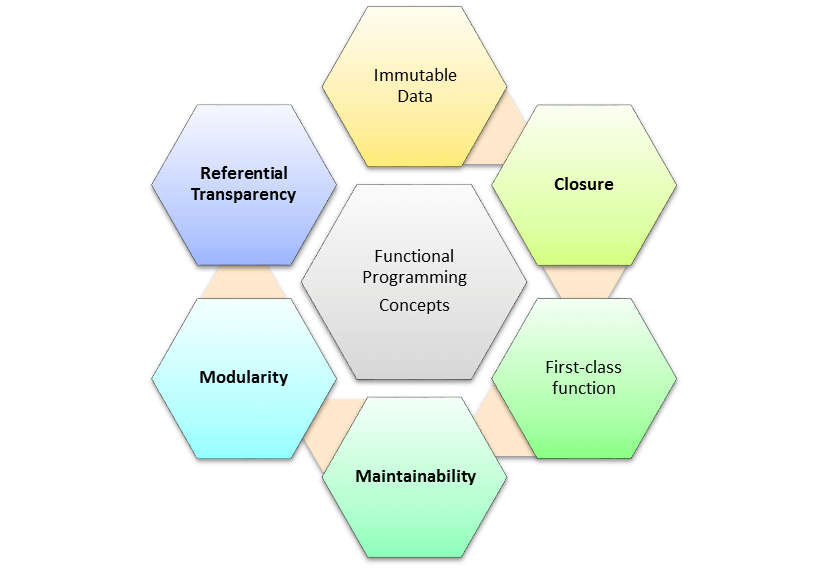
1. They always produce the same output with the same arguments irrespective of other factors.
2. They are deterministic. Pure functions either give some output or modify any argument or global variables i.e., they have no side effects.

Because pure functions have no side effects or hidden I/O, programs built using functional paradigms are easy to debug. Moreover, pure functions make writing concurrent applications easier.

When the code is written using the functional programming style, a capable compiler can:

1. Memorize the results
2. Parallelize the instructions
3. Wait for evaluating results

## Basic Functional Programming Terminology and Concepts



1. **First-Class Functions:** A first-class citizen is an entity of the language that supports the following operations:
2. Storing the entity in a variable
3. Passing the entity as an argument of a function
4. Returning the entity when calling a function

JavaScript’s functions are first-class citizens-

1. They can be stored in variables.
2. They be passed as arguments of other functions.
3. They be returned when calling other functions.

Therefore, functions are first-class citizens in JavaScript, meaning we can write FP programs using this language.

1. **Higher-Order Functions:**
2. Higher-order functions take functions as an argument or return the functions.
3. Higher-order functions are also essential for building programs in functional programming.
4. Higher-order functions allow partial applications or currying.
5. This technique applies a function to its arguments one at a time, as each application returning a new function which accepts the next argument.

Code:

Function show(a,b){

Return function(){

Console.log(a + b);

}

}

1. **Variables are Immutable:**
2. In functional programming, we can’t modify a variable after it’s been initialized.
3. We can create new variables – but we can’t modify existing variables, and this really helps to maintain state throughout the runtime of a program.
4. Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.
5. **Recursion:** There are no “for” or “while” loop in functional languages. Iteration in functional languages is implemented through recursion. Recursive functions repeatedly call themselves, until it reaches the base case.   
   example of the recursive function:

fib(n){

if (n <= 1)

return 1;

else

return fib(n - 1) + fib(n - 2);

}

**Function Composition:** Function composition allows you to combine pure functions to create more complicated ones.

Code:

Const trim = str => str.trim();

Const wrap = str => `<div> ${str} </div>`;

Const result = wrap(trim(“ .. shuvo … “));

**Referential Transparency:** Functional programs should perform operations just like as if it is for the first time. So, you will know what may or may not have happened during the program’s execution, and its side effects. In FP term it is called Referential transparency.

**Object-Oriented Programming vs Functional Programming**

1. OOP uses the imperative programming model, meaning functions are invariably coded in every step needed to solve a problem. You code each operation with the code itself specifying how to solve the problem. This model requires the programmer to know which functions are necessary to solve a problem instead of relying on models that can solve the problems.
2. FP uses the declarative programming model, meaning it relies on the underlying concepts of a programming language to execute the necessary steps to reach the predetermined outcome.
3. Imperative programs focus on the step-by-step process of solving a problem, whereas declarative programs focus on the result of solving a problem.
4. Another critical difference is mutability: OOP uses mutable data while FP uses immutable data. You can alter (or mutate) mutable objects after creation, whereas you can’t for immutable objects. In FP, you’ll need to make a copy of the object and use that copy to write the rest of your code.
5. Overall, immutable code is easier to update, more efficient to manage, and easier to test and debug. And because variables are constant, the resulting code is easier to understand and reason about. Many programmers and software developers prefer to work with FP models.
6. Ultimately, the right programming paradigm for you will depend on your intended application. OOP works best for standardized and straightforward projects, whereas FP works best for projects that require scalability and flexibility.

# Design pattern

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. Rather, it is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages. Some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

1. Creational,
2. Structural
3. Behavioral patterns

**Creational patterns:**

1. Abstract factory
2. Factory method
3. Builder
4. Prototype
5. Singleton
6. Object Pool

**Structural :**

1. adapter,
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy
8. Private Class Data

**Behavioral patterns**

1. Blackboard
2. Chain of responsibility
3. Command
4. Iterator
5. Interpreter
6. Mediator
7. Memento
8. Observer or Publish/subscribe
9. Strategy
10. State
11. Template method
12. Visitor
13. Null Object

# Design pattern in Java

**Java core design pattern:** there are many design pattern. But in core java we learn something important that we need more.

1. **Creational patterns:**
2. Singleton
3. Abstract factory
4. Builder
5. Factory method
6. Prototype
7. **Structural:**
8. adapter,
9. Bridge Pattern
10. Composite
11. Decorator
12. Facade
13. Flyweight
14. Proxy
15. **Behavioral patterns**
16. Chain of Responsibility Pattern
17. Command Pattern
18. Interpreter Pattern
19. Iterator Pattern
20. Mediator Pattern
21. Memento Pattern
22. Observer Pattern
23. State Pattern
24. Strategy Pattern
25. Template Pattern
26. Visitor Pattern

**Advance java design pattern:**

1. DOA
2. DTO
3. MVC
4. ORM
5. AOP & IOC
6. DI

# Singleton design pattern in Java:

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. Example- for DB connection we create one object of database connectivity then use this object when we need it.

Create singleton class:

1. Create a private constructor of the class to restrict object creation outside of the class.
2. Create a private attribute of the class type that refers to the single object.
3. Create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

Code:

class Singleton {

    private static Singleton obj;

    private Singleton () {}

    public static Singleton getInstance(){

        if (obj==null) {

            obj = new Singleton ();

}

        return obj;

    }

}

# Facade Design Pattern

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants. You, as client want access to different menus of different restaurants. You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the facade, as he hides the complexities of the system hotel.

**Code:**

1. **Interface of Hotel:**

package structural.facade;

public interface Hotel {

     public Menus getMenus();

}

1. **NonVegRestaurant.java:**

package structural.facade;

 public class NonVegRestaurant implements Hotel {

     public Menus getMenus() {

        NonVegMenu nv = new NonVegMenu();

        return nv;

    }

}

1. **VegRestaurant.java**

package structural.facade;

 public class VegRestaurant implements Hotel {

     public Menus getMenus()  {

        VegMenu v = new VegMenu();

        return v;

    }

}

1. **HotelKeeper.java**

package structural.facade;

 public interface HotelKeeper {

     public VegMenu getVegMenu();

   public NonVegMenu getNonVegMenu();

   public Both getVegNonMenu();

}

1. **HotelKeeperImplementation.java**

package structural.facade;

public class HotelKeeperImplementation implements HotelKeeper {

public VegMenu getVegMenu(){

        VegRestaurant v = new VegRestaurant();

        VegMenu vegMenu = (VegMenu)v.getMenus();

        return vegMenu;

 }

public NonVegMenu getNonVegMenu()    {

        NonVegRestaurant v = new NonVegRestaurant();

        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();

        return NonvegMenu;

}

}

1. **Client use façade class:**

package structural.facade;

public class Client{

    public static void main (String[] args) {

        HotelKeeper keeper = new HotelKeeper();

        VegMenu v = keeper.getVegMenu();

        NonVegMenu nv = keeper.getNonVegMenu();

    }

}

# Dependency injection

**Dependency:**

Before know dependency injection we need to know what is dependency. Dependency or dependent means relying on something for support. As example, if we want to go somewhere we depend on car.

In programming we say When class A uses some functionality of class B, then it’s said that class A has a dependency of class B. if we want to use other class method, we need to create the object of that class. Without create object we cannot use method of other class. Here class A create the object of class B and then use class B method.

Code:

Class engine{

Function show(){

Count<<”this is goo”;

}

}

Class Car {

Car yahamaEngine = new Engine();

YahamaEngine.show();

}

In above, Car use show method by create object of Engine class. So, we called Car is dependence of engine class.

Here is some problem of class dependency:

1. Class is not testable
2. Code is not extensible
3. Single responsibility
4. Lifetime of Object:

To solve this type of problem we use dependency injection.

**Dependency injection:**

In dependency injection dependence class object are create by someone else and dependable class use this object. because dependencies can be injected at runtime rather than at compile time

The 3 Types of Dependency Injection---

1. constructor injection,
2. method injection,
3. property injection.

**Constructor Injection:**

Constructor injection is the process of using the constructor to pass in the dependencies of a class. You should use constructor injection when your class has a dependency that the class requires in order to work properly.

If your class cannot work without a dependency, then inject it via the constructor.

you should use constructor injection when the dependency in question has a lifetime longer than a single method. Dependencies passed into the constructor should be useful to the class in a general way, with its use spanning multiple methods in the class. If a dependency is used in only one spot, method injection

Checking for null is necessary and is boilerplate code. Protecting against null being passed as a parameter is called the guard pattern

public class CustomerBusinessLogic{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic(ICustomerDataAccess custDataAccess){

\_dataAccess = custDataAccess;

}

public CustomerBusinessLogic(){

\_dataAccess = new CustomerDataAccess();

}

public string ProcessCustomerData(int id){

return \_dataAccess.GetCustomerName(id);

}

}

public interface ICustomerDataAccess{

string GetCustomerName(int id);

}

public class CustomerDataAccess: ICustomerDataAccess{

public CustomerDataAccess(){

}

public string GetCustomerName(int id){

return "Dummy Customer Name";

}

}

**Property Injection (setter injection):**

You should use property injection in case the dependency is truly optional

Property Injection however causes Temporal Coupling and when writing Line of Business applications, your dependencies should never be optional: you should instead apply the Null Object pattern.

property injection is considered bad in 98% of all scenarios because it hides dependencies and there is no guarantee that the object will be injected when the class is created.

The built-in IoC container does not support property injection. You will have to use a third-party IoC container.

Code:

public class CustomerBusinessLogic{

public CustomerBusinessLogic(){

}

public string GetCustomerName(int id){

return DataAccess.GetCustomerName(id);

}

public ICustomerDataAccess DataAccess { get; set; }

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

\_customerBL.DataAccess = new CustomerDataAccess();

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

**Method Injection:**

Thus, method injection is useful in two scenarios: when the implementation of dependency will vary, and when the dependency needs to be renewed after each use. In both cases, it’s up to the caller to decide what implementation to pass to the method.

interface IDataAccessDependency{

void SetDependency(ICustomerDataAccess customerDataAccess);

}

public class CustomerBusinessLogic : IDataAccessDependency{

ICustomerDataAccess \_dataAccess;

public CustomerBusinessLogic() { }

public string GetCustomerName(int id){

return \_dataAccess.GetCustomerName(id);

}

public void SetDependency(ICustomerDataAccess customerDataAccess){

\_dataAccess = customerDataAccess;

}

}

public class CustomerService{

CustomerBusinessLogic \_customerBL;

public CustomerService(){

\_customerBL = new CustomerBusinessLogic();

((IDataAccessDependency)\_customerBL).SetDependency(new CustomerDataAccess());

}

public string GetCustomerName(int id) {

return \_customerBL.GetCustomerName(id);

}

}

**Inversion of Control (IoC)**