Chapter - 2

Date
Page

Design Patterns

Pattern :

→ A design pattern is general repeatable solution to a commonly occuring problem in software design.

→ A design pattern isn't a finished design that can be transformed directly into code.

→ It is a description or template for how to solve a problem that can used in many different situations.

Types of Pattern :

→ 3 types :-

1. Creational Pattern
2. Structural Pattern
3. Behavioral Pattern

1. Creational Pattern :-

→ Provide object creation mechanisms that increase flexibility and reuse of code.

→ Types :-

a. Singleton Pattern
b. Factory Method Pattern
c. Abstract factory Pattern
d. Builder Pattern
e. Prototype Pattern.

Date
Page

(2) Structural Pattern :

→ Gen. explain how classes and objects can be composed, to form large structures.

→ It simplifies the structure by identifying the relationships.

→ focus on how classes inherit from each other & how they are composed from other classes.

→ Types :-

(a) Adapter Pattern
(b) Bridge Pattern
(c) Composite Pattern
(d) Decorater Pattern
(e) Proxy Pattern
(f) Facade Pattern.

(3) Behavioral Pattern :

→ take care of effective communication and the assignment of responsibilities between objects.

→ Types :

(a) Chain of Responsibility.
(b) Command Pattern
(c) Observer Pattern
(d) Memento Pattern
(e) Mediator Pattern
(f) Template Pattern
(g) Strategy Pattern.

Date
Page

Singleton Pattern :

Importance of Design Pattern :-

→ Importance of Design Pattern :-

(1) It provides tried and tested solution for repeating common problem is software design.

(2) It defines common language so that we can communicate with our team-members more efficiency

(3) It can speed up the development process by providing tried and tested solution.

(4) It provides reusable solution.

Singleton Pattern :

→ Singleton Pattern is a creational pattern which ensures that the class have only one instance, while providing global Access point to this instance.

→ The implementation involves a static member in the "Singleton" Class, a private constructor, and static public method that returns reference of the static members.

→

Singleton
- instance : Singleton
- Singleton();
+ getInstance(): Singleton

fig :- Implementation of Singleton using UML Class diagram

→ get Instance () is responsible for creating class unique instance in case it is not created yet. and returns that instance

→
```
public class SingletonClass{
    private static SingletonClass = new SingletonClass();
    private SingletonClass (){ }
    public static SingletonClass getInstance(){
        return instance ;
    }
    public void showMessage (){
        system.out.println(" I'm single object");
    }
}
```

{Pethanni porno vamey naina}

→ Here,
This class is creating object itself, which represents the global instances.
By providing the private constructor, the class cannot be instantiated.
the get Instance () method is used as global access point for the rest of the application.

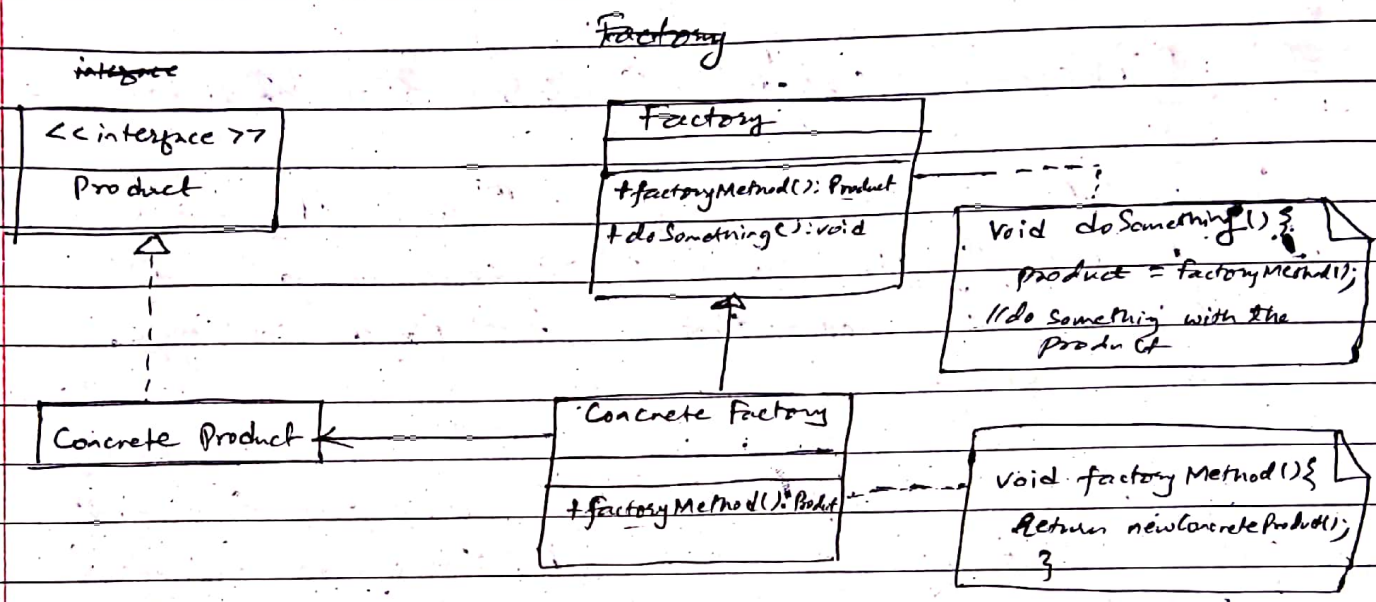
```
public class main (){
    public static void main (Str [] args){
        SingletonClass SingletonClass = SingletonClass.
                                        getInstance();
        SingletonClass. showMessage();
    }
}
```
O/P → I'm Single object

Date
Page

Factory Pattern:

→ It is a creational pattern, that provides interface for creating an objects for superclass, but allows the subclass to alter the ~~type of~~ type of objects that will be ~~Creat~~ created.

→



→ Product defines the interface for ϕ objects the factory method creates.

→ Concrete Product implements a product interface

→ Creator (i.e, Factory because it creates Product objects) declare the method 'factoryMethod', which returns a Product object.

→ ConcreteCreater overrides the generating methods for creating ConcreteProduct objects.

→ The need for implementing factoryMethod is very frequent.

Date
Page

- Some Cases are as follows :-

(1) When classes cannot anticipate (predict) the type of objects it is supposed to create.

(2) When class wants its subclass to be one of the type of newly created objects.

** Difference between Singleton & Factory Method :-

	Singleton Pattern		Factory Pattern
(1)	It generally gives same instance of whatever type we are retrieving.	(1)	It generally gives different instance of each type.
(2)	The purpose of Singleton Pattern is to want all calls to go through same instance.	(2)	The purpose of factory Pattern is to create and return new instance
Definition (3)	It ensures that the class have only one instance, while providing global access point to this instance	(3)	Definition
UML figure (4)	UML diagram		UML diagram.
(5)	less Scalable	(5)	More Scalable
(6)	Same instance of different types	(6)	different instance for same types.

Date
Page

Dependency Injection and Inversion of Control :

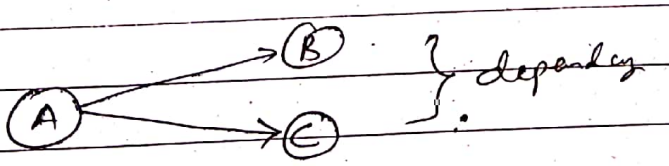
→ The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependency from the code.

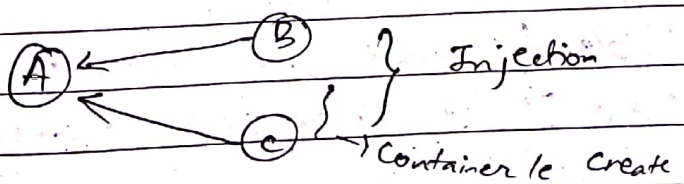
→ A class X has dependency to class Y, if class X uses class Y as a variable.

```
Class X {
    Y y;
    ----
    ----
}
```

→ Dependency Injection is a coding pattern in which the class ~~received~~ receives its dependencies from external sources rather than creating them itself.

to understand





→ Example :

Say your app has a text editor component and you want to provide spell checking. Then standard code is

```
public class TextEditor {
    private SpellChecker checker;
    public TextEditor() {
        checker =
```

Date
Page

```
public class TextEditor {
    private SpellChecker checker;
    public TextEditor()
    {
        checker = new SpellChecker();
    }
}
```

Here, we have created dependency between TextEditor and SpellChecker. So, if we made changes in SpellChecker then the TextEditor must also have to change itself.
So, to remove this dependency we use Inverse of Control (IoC). which is given below:

```
public class TextEditor {
    private ISpellCheck checker;
    public TextEditor(ISpellChecker checker){
        this.Checker = checker;
    }
}
```

→ IoC is the principle by which the control of an object is transferred to a container or framework.

→ DI is a pattern through which IoC is implemented.

→ "Injecting" an objects into another objects is done by container rather than by object itself.

→ Benefits of DI & IoC:

① Allows us to remove hard-coded dependencies.

Date
Page

(2) Make our application loosely coupled.
(3) Makes ~~extensible~~ extendable and Maintainable.

→ Disadvantage :-
(1) If overused, it can lead to maintenance issues.
(2) It hides service class dependencies that can lead to runtime error.

→ Types of DI :-
(1) Constructor ~~Depen~~ DI
(2) Setter (Property) DI
(3) Method DI :-

(1) Constructor DI :- (CDI)
→ Constructor DI is the process of ~~passing~~ injecting dependencies using the constructor of a class.
→ For eg :-

```
public class GfG {
    IGreek greet; // object of the interface IGreek
    GfG (IGreek gseek) { // Constructor to set the CDI
        this.gseek = gseek;
    }
}
```

→ Provides good readability as it ~~provides~~ seperately present in the code.

(2) Setter (property) DI :-
→ It is one of the simplest method.
→ Dependecies will be injected with the help of setter and /or getter method.
→ For eg :-

```
public class GfG{
    IGreek greek; // object of the Interface IGreek
    // setter method for property greek
    public void setGreek(IGreek greek){
        this.greek = greek;
    }
}
```

(c) Method DI :-

→ In this injection, client class implements an interface which declare the methods to supply the dependencies

→ For example :-

```
public class GfG{
    IGreek greek;
    public void saveGreek(IGreek greek)
    public GfG(){}

    public void saveGreek(IGreek greek)
    {
        greek.length();
    }
}
```

Date
Page

Lazy Initialisation :

→ Lazy Initialization is the technique that initializes a variable (in OO content usually a field of class) on it's first access.

→ Lazy initialization is useful when calculating the value of the field is time consuming and you don't want to do it until you actually need.

→ We should not use this technique, until we have some real problem to solve.

→ Lazy initialization is the technique of delaying the creation of an object, the calculation of a value, or some other expensive process, until it is first time needed.

{website ma lazy loading}.

See note

→ for example :-

Lazy Initialisation in Singleton pattern

```
public class LazySingleton {
    public static volatile LazySingleton instance = null;
    // private constructor
    private LazySingleton() { }
    public static LazySingleton getInstance() {
        if (instance == null)
            { synchronized (LazySingleton.class) {
                instance = new LazySingleton();
            }
        }
        return instance;
    }
```

Date
Page

Convention Vs Configuration :-

→ Convention over Configuration is a simple concept that is primarily used in programming.

→ It means that the environment in which we work (system, libraries, language, etc.) assumes many logical situations by default.

→ So, if you adapt to them instead of creating your own rules each time, programming becomes more easier and more productive.

→ The main goal is to decrease the number of decisions the programmer has to take, gaining simplicity and not losing flexibility.

→ The immediate result is that we can create many more things in less time.

→ Software frameworks that supports Convention over Configurations are Ruby on Rails, JavaBeans, CakePHP, etc.

→ We can ignore convention, by replacing them with our own code. But convention is not arbitary, they are established by community of high level programmer, so it doesnot make sense to waste time overwriting them.

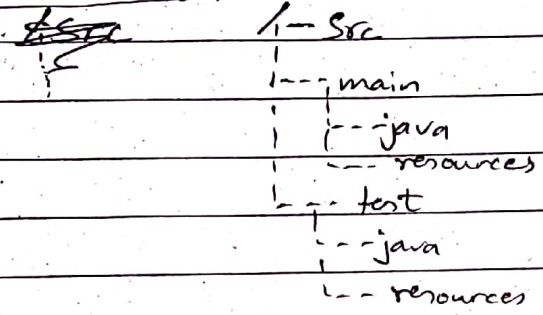
Date
Page

→ for eg :-

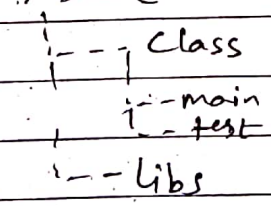
~~In context of 'Gnits',~~

In a Java project build with Gradle, with a single line of code we can apply java plugins to this project.

Gradle then automatically looks for source code in /src/main/java.



It creates a /build folder where the compiled class and JAR files are saved.



This is all by convention. We don't have to tell Gradle these details via configuration files.