GoF Patterns

Sunday, December 25, 2022 6:19 PM

Why do we need design patterns?

Gang Of Four - Design Patterns Elements of Reusable Object-Oriented Software.

* Importance
- Flexible for customization
- The GoF pattern describe 23 design pattern

CREATIONAL	STRUCTURAL	BEHAVIORAL
• Singleton • Prototype • Factory Method • Builder • Abstract Factory	• Proxy • Flyweight • Composite • Bridge • Façade • Decorator • Adapter	• Observer • Strategy • Template Method • Command • Iterator • Memento • State • Mediator • Chain of Responsibility • Visitor • Interpreter

Creational Patterns

Instantiation process - that deals with object creation pattern.
- Independent of how object are composed or created.
- They control object creation

Structural Pattern

Focus of how classes and object can be composed to form large class.
They use inheritance to compose interface for interface

BEHAVIORAL PATTERN

Put stress on algorithm and how objects interact with each other.
Process or a flow by way object interact with each other.

DIFFERENCES BETWEEN STRUCTURAL & CREATIONAL PATTERN

Creational patterns - provides a way to create objects while hiding the creational logic, provides the flexibility for creating objects based on different use cases.
Structural pat. - provides use options how to collapse object or class together to form large patterns.

CLASS & Object Patterns

Class & Object Patterns

Scope	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Singleton Prototype Builder Abstract Factory	Proxy Flyweight Composite Bridge Façade Decorator Adapter (object)	Observer Strategy Command Iterator Memento State Mediator Chain of Responsibility Visitor

CLASS PATTERN

Patterns made based on object or class SCOPE

CLASS PATTERN

deals with class of step class that uses inheritance which are fix at run time

Object Pattern

Deals with object that can change at runtime.

DIFFERENCE BETWEEN CLASS & OBJECT PATTERNS

Difference between Class and Object patterns

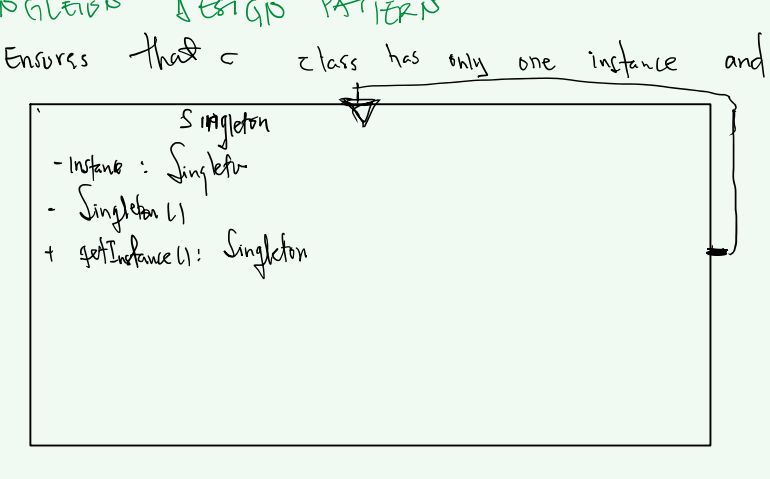
	Class Patterns	Object Patterns
Creational	Can defer object creation to subclasses	Can defer object creation to another object
Structural	Focuses on the composition of classes (primarily uses the concept of inheritance)	Focuses on the different ways of composition of objects
Behavioral	Describes the algorithms and execution flows	Describes how different objects can work together and complete a task

CREATIONAL PATTERN

- Singleton
- Prototype
- Factory Method
- Builder
- Abstract Factory
- Factory Method VS Abstract Factory

SINGLETON DESIGN PATTERN

Ensures that a class has only one instance and provide a global point of access to it.




```java
package org.code;

3 usages
public class OrderManagementServiceSingleton {
    3 usages
    private static OrderManagementServiceSingleton instance = null;
    1 usage
    private OrderManagementServiceSingleton() {
    }
    no usages
    public static OrderManagementServiceSingleton getInstance() {
        if (instance == null) {
            instance = new OrderManagementServiceSingleton();
        }
        return instance;
    }
}
```

NOTE:
We can include Synchronized - Key word to ensure there is Synchronization with thread or object creation is persisted in thread.

Test Print Output for any instance of this object created should be same
A test code is shown below.

```java
public class Main {
    no usages
    public static void main(String[] args) {

        var orderManagementServiceSingleton = OrderManagementServiceSingleton.getInstance();
        var orderManagementServiceSingleton2 = OrderManagementServiceSingleton.getInstance();

        System.out.println(orderManagementServiceSingleton==orderManagementServiceSingleton2);
        System.out.println("orderManagementServiceSingleton!");
    }
}
```

NOTE BETTER
- Define a private static field in the 'Singleton' class
* Make constructor private.
• Implement public accessor function
• Implement lazy initialization - that is creation of first use - Inside the accessor function
* Make sure that accessor function is synchronized in case if is going to be used in multithreading environment.

Prototype

Simplify object instantiation faster. Creating complex object faster without using "new" keyword

```java
package org.code.prototype;

import java.io.File;

10 usages  2 implementations
public interface Archiver {
    1 usage  2 implementations
    void archive(File directory);
    2 implementations
    Archiver clone();
}
```

- RarArchiver
  - Implements the archiver class and clones

```java
package org.code.prototype;


import org.apache.commons.lang3.SerializationUtils;

import java.io.File;
import java.io.Serializable;

1 usage
public class RarArchiver implements Archiver, Serializable  {
    1 usage
    @Override
    public void archive(File directory) {
        System.out.println("RarArchiver Archiving directory " + directory.getName() + " to zip");
    }

    @Override
    public Archiver clone() {
//        return new RarArchiver();
        return (Archiver) SerializationUtils.clone( object: this);
    }
}
```

The archiver clones this class instance and returns it when requested

- ZipArchiver.
  - Does same like the rarArchiver. The logic holds same just that we return different instance of the class

```java
package org.code.prototype;


import org.apache.commons.lang3.SerializationUtils;

import java.io.File;
import java.io.Serializable;

1 usage
public class ZipArchiver implements Archiver, Serializable  {
    1 usage
    @Override
    public void archive(File directory) {
        System.out.println("ZipArchiver Archiving directory " + directory.getName() + " to zip");
    }

    @Override
    public Archiver clone() {
//        return new RarArchiver();
        return (Archiver) SerializationUtils.clone( object: this);
    }
}
```

FINALLY ARCHIVERFACTORY

In the ArchiverFactory we can create or return the instance of the object (class) needed by the parameter passed

```java
package org.code.prototype;

import java.util.Map;

1 usage
public class ArchiverFactory {
    1 usage
    private  static Map<String, Archiver> archivers = Map.of(
            k1: "zip", new ZipArchiver(),
            k2: "rar", new RarArchiver()
    );

    1 usage
    public static Archiver getArchiver(String type) {
        return archivers.get(type).clone();
    }

    //use switch case
    no usages
    public static Archiver getArchiver2(String type) {
        switch (type) {
            case "zip":
                return new ZipArchiver().clone();
            case "rar":
                return new RarArchiver().clone();
            default:
                throw new IllegalArgumentException("Unknown type");
        }
    }
}
```

We can use HashMap or just switch case to check the type.

We can simple use an instance of the type provided and then return the appropriate object type from the factory.

A PROTOTYPE TEST - FOR GIVEN TYPES

```java
package org.code.prototype;

import java.io.File;

no usages
public class PrototypeCreationalPatternTest {
    no usages
    public static void main(String[] args) {
        String[] archiveTypes = {"zip", "rar"};
        for (String archiveType : archiveTypes) {
            Archiver archiver = ArchiverFactory.getArchiver(archiveType);
            archiver.archive(new File( pathname: "C:\\Users\\User\\Desktop\\test"));
        }

    }
}
```

NOTE
- Add clone() method to the hierarchy of our Objects.
- Design a registry that maintains a cache of prototypical Objects
- Design API that allows to clone object inside the factory and return new object as a result.
- Use factory API instead of NEW keyword to instantiate the object.

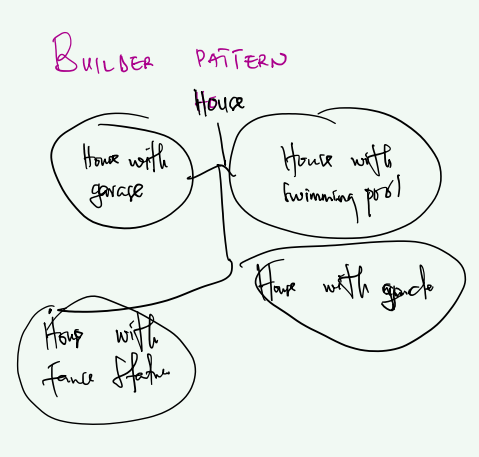
* FACTORY METHOD

The idea of factory pattern is to create interface to create object and let subclass to decide what can create ~~methods~~ object to choose ~~and~~ what class can instantiate.
- When we want to standardize what object is to instantiate.
- Used in abstract method usually.

Factory Method or pattern generates different object each time the factory is called.
This means the references to object from the factory points to different memory location.
Initial Interface

Factory Method Pattern
- Define hierarchy of objects.
- Design the argument for the factory method. Think about qualities or characteristics that are necessary and sufficient to identify the correct derived class to instantiate
- Call factory method during the runtime to instantiate the object that you need.

* BUILDER PATTERN



BUILDER PATTERN

Checklist:

To implement builder pattern with director we have to:

- Declare builder interface
- Create multiple builders
- Create director that can work with different builders
- In client code instantiate director and concrete implementation of builder to build new objects

To implement builder pattern with chain method calls we have to:

- Create inner Builder class inside type that we want to build
- Implement method in type that returns reference to the builder object
- Declare setters in builder that returns the reference to this builder object
- Implement build() method to terminate method chain and return target object
- In client code call method to get reference to builder and build object step-by-step

* ABSTRACT FACTORY

We need abstract factory when we need to create a group of inter-dependant class

Factory Method VS Abstract Factory