Design Patterns

### 1) Types of Design Patterns:

1. ***Creational Patterns* :** This patterns talks about the way a class/object is created. Class-creational pattern use the inheritance in instantiation process and Object-Creational pattern use the delegation effectively to get the job done. Some of the *examples* are :  Factory Method, Abstract Factory, Builder, Singleton, Object Pool, Prototype and Singleton.
2. ***Behavioral Patterns :***These patterns are about identifying common communications between different objects and apply these patterns. Some of the *examples* are : Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor
3. ***Structural Patterns :***These patterns are about creating a bigger structure using the creational and behavioral patterns. Some of the *examples* are : Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

**Creational :**

1. ***Singleton:*** This pattern is used when we need only one object for a class throughout the application. The real time usage would creating DB configuration. In the following examples we will look at how to create a singleton and how can we exploit the singleton class and also how to prevent from exploitation.

Eager example show that it creates the singleton object when the class is compiled, while in Laxy the object gets created when only we call the getInstance method.

Ways to break Singleton :

1. MultiThreaded
2. Serializtion
3. Reflection
4. Clonable
5. Multiple Class Loaders

Example that covers all the above scenarios:



1. Factory method: Factory pattern gives us a ways to encapsulate the instantiations of concrete types. Factory Method pattern encapsulates the functionality required to instantiate a class inside a designated method referred to as Factory Method.

Example : Consider a company have an xml Parser which parsers the request sent to it and responses to it based on the request, could be an error, feedback or a response.



The following are the usage(s) of the Factory Method Pattern in JDK.

• java.util.Calendar#getInstance()

• java.util.ResourceBundle#getBundle()

• java.text.NumberFormat#getInstance()

1. Prototype:



1. Builder: This pattern encapsulates the logic which constructs the object. When we have a class with multiple variables it could make the object very heavy. So in order to avoid that we will create a Builder class which is used to create the object. Multiple classes can be created in order to create objects with different paramaters.

Builder Interface (CarBuilder)

public void buildCarType(String carType);

public void buildEngine(String engine);

public void buildFuelType(String fuelType);

public void buildPower(Integer engine);

public Car getCar();

Client

|  |
| --- |
| Builder Implementation (CarBuilderImpl) |
| **public void** buildCarType(String carType);  **public void** buildEngine(String engine);  **public void** buildFuelType(String fuelType);  **public void** buildPower(Integer engine);  **public** Car getCar(); |

|  |
| --- |
| Product(Car) |
| **private** String **carType**;  **private** String **engine**;  **private** Integer **power**;  **private** String **fuelType**; |

1. Abstract Factory Design Pattern: This pattern allows to create a family of familiar or dependent objects by encapsulating the concrete types. We create a factory of factories which in turn implement abstract factory interface.

Consider the example where we have a company which needs its client to parse a xml which could be a errorXML or ResponseXML, here the XML format is given by the company itself. But if the clients wants to use his own xml format then company need to first need to find out which xml he is using by creating a factory which would inturn contain factory method to decide which parser to use.

diagram:

|  |
| --- |
| NYParserFactory |
| getParserInstance(String parserType) |

|  |
| --- |
| <<Class>>  Abstract Factory |
| getFactory() |

|  |
| --- |
| TWParserFactory |
| getParserInstance(String parserType) |

|  |
| --- |
| <<Interface>>  XML Parser |
| parse() |

|  |
| --- |
| ResponseXMLParser |
| parse() |

|  |
| --- |
| ErrorXMLParser |
| parse() |



**Structural Design Pattern**

Adapter : This design pattern converts one interface into another interface that a client understands. Lets say we have an interface which is exposed but the client may not understand the interface so it needs to converted to client understandable interface.

For ex :- Lets say a company is implementing a third party gateway now if they want to have there own gateway so they need to change code that client is using in order to change gateway. In order to avoid this we need to create an adapter which converts the third party interface to match your own interface.

Adapter Pattern Diagram :

Client

Interface

Adapter

Adaptee



Bridge : This patterns separates the abstraction from implementation and client is exposed to only abstraction. This is used in case if we have some code which could be applied to multiple scenarios which little change. In this pattern the abstraction layer grows independently of the implementation layer.

For ex :- Let’s say we have a Car company which produces central locking system and gear locking system. But the company gets assignments from 2 different companies which are slightly different from each other. Here we need to write 4 different classes in order to implement all these, instead of that we separate the common functionality into different class.

****