***Design Patterns[23]***

**Creational Patterns**

1. **Abstract Factory :** Creates an instance of several families of classes
2. **Builder :** Separates object construction from its representation
3. **Factory Method** : Creates an instance of several derived classes
4. **Prototype** : A fully initialized instance to be copied or cloned
5. **Singleton** : A class of which only a single instance can exist

**Structural Patterns**

1. **Adapter** : Match interfaces of different classes
2. **Bridge** : Separates an object’s interface from its implementation
3. **Composite** : A tree structure of simple and composite objects
4. **Decorator :** Add responsibilities to objects dynamically
5. **Façade :**  A single Interface that represents an entire subsystem
6. **Flyweight :** A fine-grained instances used for efficient sharing
7. **Proxy :** An object representing another object

**Behavioural Patterns**

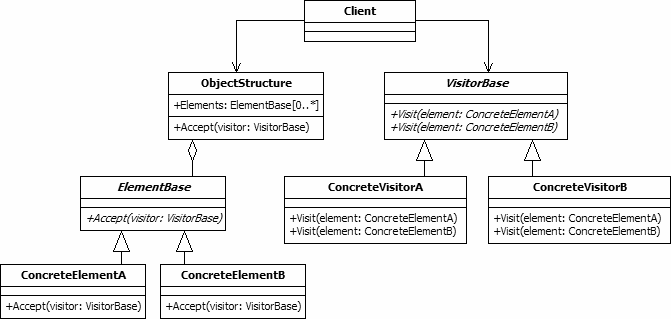
1. **Chain of Resp**: A way of passing a request between a chain of objects
2. **Command :** Encapsulate a command request as an object
3. **Interpreter** : A way to include language elements in a program
4. **Iterator :** *Sequentially access the elements of a collection*
5. **Mediator :** Defines simplified ***communication between classes***
6. **Memento :** Capture and restore an object's internal state
7. **Observer :** A way of notifying change to a number of classes
8. **State :** Alter an object's behaviour when its state changes
9. **Strategy:** Encapsulates an algorithm inside a class
10. **Template Method** : Defer the exact steps of an algorithm to a subclass
11. **Visitor**: Defines a new operation to a class without change

***Behavioural Design Patterns[11]*** *- deals with communication among the objects.*

1. **Visitor [usage-High]**

**Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates**

*Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects.* With the help of visitor pattern, we can move the operational logic from the objects to another class.



**ConcreteElementA{**

**public void accept (VisitorBase visit){**

**visitor.visit(this);**

**}**

**}**

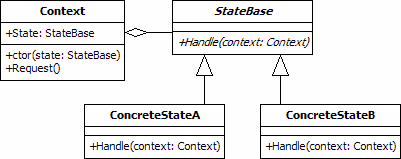
**Example**: think of a Shopping cart where we can add different type of items (Elements), when we click on checkout button, it calculates the total amount to be paid. Now we can have the calculation logic in item classes or we can move out this logic to another class using visitor pattern.

1. **State [usage-High]**

**Allow an object to alter its behaviour when its internal state changes.**

**State design pattern is used when an Object change its behaviour based on its internal state. Also, by implementing it, the code should remain cleaner *without many if/else statements.***

State pattern is used to provide a systematic and lose-coupled way to achieve this through **Context** and **State** implementations*.* ***Context*** *is the class that has a State reference to one of the concrete implementations of the State and forwards the request to the state object for processing*.



***The benefits of using State pattern to implement polymorphic behaviour is clearly visible, the chances of error are less and it’s very easy to add more states for additional behaviour making it more robust, easily maintainable and flexible. Also, State pattern helped in avoiding if-else or switch-case conditional logic in this scenario.***

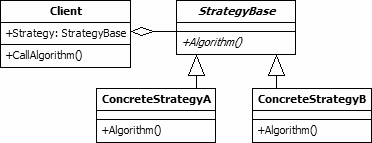
**Example:** Suppose we want to implement a TV Remote with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV

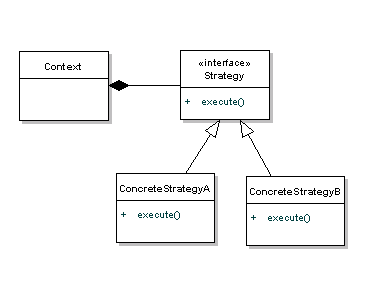
1. **Strategy [usage-High]**

**The strategy pattern is used to create *an interchangeable family of algorithms* from which the required process is chosen at run-time.**

Strategy pattern is also known as **Policy Pattern**. We define multiple algorithms and let client application pass the algorithm to be used as a parameter.

One of the best example of this pattern is ***Collections.sort()* method that takes Comparator parameter**. Based on the different implementations of Comparator interfaces, the Objects are getting sorted in different ways.





***Strategy Pattern is very similar to State Pattern. One of the difference is that Context contains state as instance variable and there can be multiple tasks whose implementation can be dependent on the state whereas in strategy pattern strategy is passed as argument to the method and context does not hold any strategy state variable.***

Strategy pattern is useful when we have multiple algorithms for specific task and we want our application to be flexible to choose any of the algorithm at runtime for specific task.

**Example**: Implement a simple Shopping Cart where we have **two payment strategies – using Credit Card or using PayPal.**

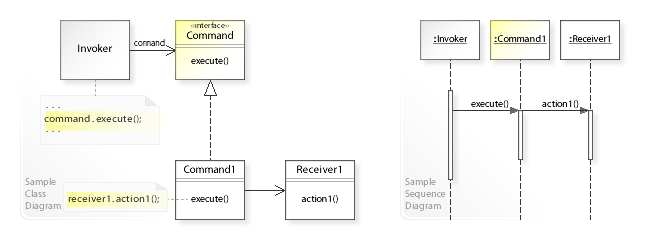
1. **Command [usage-High]**

***Command pattern is a behavioural*** [***design pattern***](https://howtodoinjava.com/gang-of-four-java-design-patterns/) ***which is useful to abstract business logic into discrete actions which we call commands. This command object helps in loose coupling between two classes where one class (invoker) shall call a method on other class (receiver) to perform a business operation.***

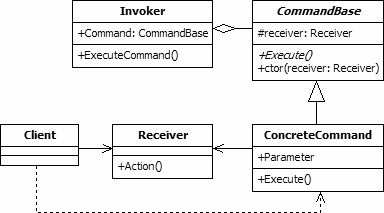
It’s used to implement **lose coupling** in a request-response model.

*In command pattern, the request is send to the invoker and invoker pass it to the encapsulated command object. Command object passes the request to the appropriate method of Receiver to perform the specific action. The client program create the receiver object and then attach it to the Command. Then it creates the invoker object and attach the command object to perform an action*. ***Now when client program executes the action, it’s processed based on the command and receiver object***.

### UML class and sequence diagram

[](https://en.wikipedia.org/wiki/File:W3sDesign_Command_Design_Pattern_UML.jpg)

A sample UML class and sequence diagram for the Command design pattern. [[3]](https://en.wikipedia.org/wiki/Command_pattern#cite_note-3)



**Example**:

We will look at a real life scenario where we can implement command pattern. Let’s say we want to provide a File System utility with methods to open, write and close file and it should support multiple operating systems such as Windows and Unix. To implement our File System utility, first of all we need to create the receiver classes that will actually do all the work. Since we code in terms of java interfaces, we can have FileSystemReceiver interface and it’s implementation classes for different operating system flavours such as Windows, Unix, Solaris etc.

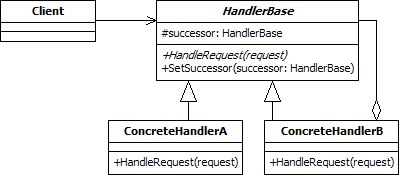
[Runnable interface](https://www.journaldev.com/1016/java-thread-example) (java.lang.Runnable) and Swing Action (javax.swing.Action) uses command pattern.

1. ***Chain of responsibility* [usage-high]**

**Gives more than one object an opportunity to handle a request by linking receiving objects together.**

*The request is passed to the first handler in the chain, which will either process it or pass it on to its successor. This continues until the request is processed or the end of the chain is reached. The handler responsible for the final processing of the request need not be known beforehand.*

Let’s see the example of chain of responsibility pattern in JDK and then we will proceed to implement a real-life example of this pattern. We know that we can **have multiple catch blocks in a try-catch block code**. Here every catch block is kind of a processor to process that particular exception. So, when any exception occurs in the try block, its send to the first catch block to process. If the catch block is not able to process it, it forwards the request to next object in chain i.e next catch block. If even the last catch block is not able to process it, the exception is thrown outside of the chain to the calling program.



1. **Observer [usage-High]**

**Defines one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.**

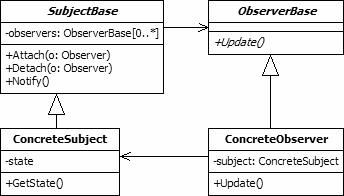
**Java provides inbuilt platform for implementing Observer pattern through *java.util.Observable* class and *java.util.Observer* interface**. However, it’s not widely used because the implementation is really simple **and most of the times we don’t want to end up extending a class just for implementing Observer pattern as java doesn’t provide multiple inheritance in classes**.

Java Message Service (JMS) uses **Observer pattern** along with **Mediator pattern** to allow applications to subscribe and publish data to other applications.

**Model-View-Controller (MVC)** frameworks also use Observer pattern where

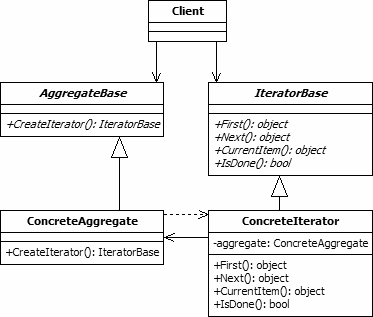
**Model is the Subject** and **Views are observers** that can register to get notified

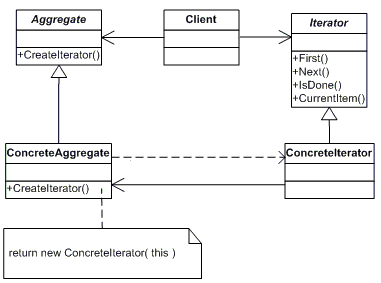
of any change to the model.



1. **Iterator [usage-High]**

**Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.**





Iterator pattern is widely used in Java Collection Framework where Iterator interface provides methods for traversing through a collection.

1. **Mediator [usage-Medium]**

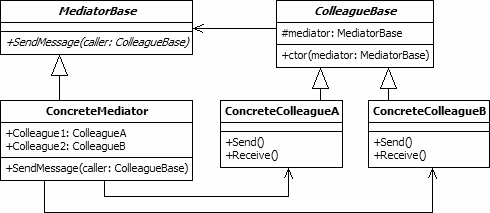
**Mediator design pattern is used to provide a centralized communication medium between different objects in a system**

Mediator design pattern is very helpful in an enterprise application where

multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects.

Air traffic controller is a great example of mediator pattern where the airport

control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have it’s own logic to provide way of communication.



Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.

Java Message Service (JMS) uses Mediator pattern along with **Observer pattern** to allow applications to subscribe and publish data to other applications.

**We should not use mediator pattern just to achieve lose-coupling because if the number of mediators will grow, then it will become hard to maintain them.**

**Example :** we will try to implement a chat application where users can do group chat. Every user will be identified by its name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.

1. **Template Method [usage-Medium]**

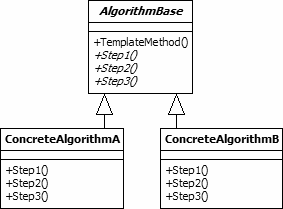
***Define the skeleton of an algorithm in an operation, deferring some implementation to the subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure*.**

*All non-abstract methods of java.io.InputStream,*

*java.io.OutputStream, java.io.Reader and java.io.Writer.*

*All non-abstract methods of java.util.AbstractList,*

*java.util.AbstractSet and java.util.AbstractMap.*



Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. ***Template method should be final.***

Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as Hollywood Principle – “don’t call us, we’ll call you”.

Methods in base class with default implementation are referred as **Hooks** and they are intended to be overridden by subclasses**, if you want some of the methods to be not overridden, you can make them final.**

**Example:** we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that we can’t change the order of execution because we can’t build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house.

Now building the foundation for a house is same for all type of houses, whether it’s a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it’s common for all the types of houses.

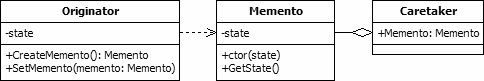
1. **Memento** **[usage-Low]**

***Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later*.**

Memento pattern is implemented with two objects – **Originator** and **Caretaker**. Originator is the object whose state needs to be saved and restored and it uses an inner class to save the state of Object. The inner class is called **Memento** and ***its private, so that it can’t be accessed from other objects.***

*Caretaker is the helper class that is responsible for storing and restoring the Originator’s state through Memento object*. *Since Memento is private to Originator, Caretaker can’t access it and* ***it’s stored as a Object within the caretaker.***

One of the best real life example is **the text editors where we can save it’s data anytime and use undo to restore it to previous saved state**.

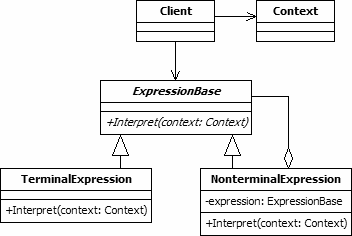


1. **Interpreter [usage-Low]**

**Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.**

Interpreter pattern is used to define a grammatical representation for a language and provides an interpreter to deal with this grammar.

The best example of this pattern is java compiler that interprets the java source code into byte code that is understandable by JVM. Google Translator is also an example of interpreter pattern where the input can be in any language and we can get the output interpreted in another language.

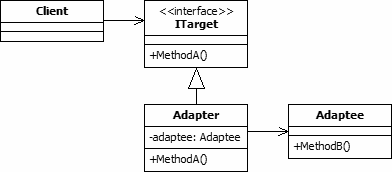


Interpreter pattern can be used when we can create a syntax tree for the grammar we have. Interpreter pattern requires a lot of error checking and a lot of expressions and code to evaluate them, it gets complicated when the grammar becomes more complicated and hence hard to maintain and provide efficiency**. *java.util.Pattern* and subclasses of *java.text.Format* are some of the examples of interpreter pattern used in JDK.**

***Structural Design Patterns***

1. **Adapter [usage-Medium]**

***Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.***



**Adapter design pattern** is one of the **structural design pattern** and it’s used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**.

**Example** : As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket.

1. **Composite [usage- Medium]**

**Compose objects into tree structures** **to represent part-whole hierarchies**. **The composite pattern is used to create hierarchical, recursive tree structures of related objects where any element of the structure may be accessed and utilised in a standard manner.**

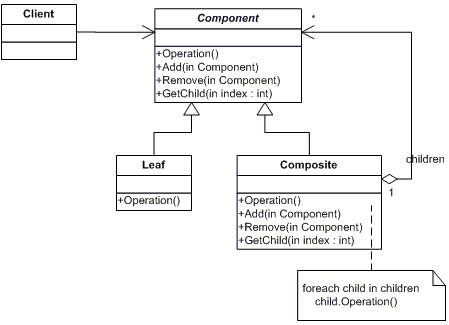
java.util.Arrays#asList()

java.io.InputStreamReader(InputStream) (returns a Reader)

java.io.OutputStreamWriter(OutputStream) (returns a Writer)

**1: Component** is an Interface here and both **Composite and Leaf Class implements Component Interface**

**2: Composite Class** has a ***Collection of Component(Composite or Leaf)****.* Example tree.



An example use of the composite pattern is for representing management structures in an organisation. In such a structure, each employee object can be the manager or zero or more subordinates.

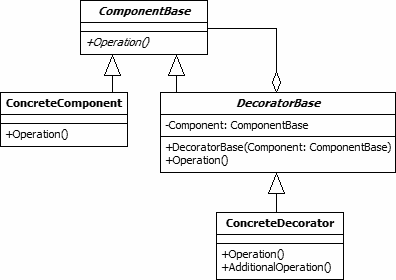
Composite pattern should be applied only when the group of objects should behave as the single object.

Composite pattern can be used to create a tree like structure.

1. **Decorator [usage -Medium]**

**Attach additional responsibilities to an object dynamically without altering the Object structure. Decorator design pattern is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behaviour.**

When we use inheritance or composition to extend the behaviour of an object but this is done at compile time and its applicable to all the instances of the class. We can’t add any new functionality of remove any existing behaviour at runtime – this is when Decorator pattern comes into picture.



***Composite Vs Decorator***

1. **Composite** gives an unified interface to a leaf and composite.

**Decorator** decorator gives additional feature to leaf, while giving unified interface.

1. Composite ***is a class*** which extends Component Interface and has a collection of Component (Leaf+Composite).**Where as DecoratorBase *is an abstract* class which extends ComponentBase** (Similar to Component interface) and also has an instance variable of type interface ComponentBase which gets initialized using ComponentBase constructor.
2. DecoratorBase sublasses can add additional features.

https://howtodoinjava.com/design-patterns/structural/decorator-design-pattern/

**Example**- Suppose we want to implement different kinds of cars – we can create

interface Car to define the assemble() method and then we can have a Basic car, furthermore we can extend it to Sports car and Luxury Car But if we want to get a car at runtime that has both the features of Sports car and Luxury car, then the implementation gets complex and if furthermore we want to specify which features should be added first, it gets even more complex. Now image if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern.

**public class DecoratorPatternTest** {

**public static void main**(String[] args) {

Car sportsCar = **new** SportsCar(**new** BasicCar());

sportsCar.assemble();

System.out.println("\n\*\*\*\*\*");

Car sportsLuxuryCar = **new** SportsCar(**new** LuxuryCar(**new**

BasicCar()));

sportsLuxuryCar.assemble();

}

}

Output of above test program is:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports

Car.

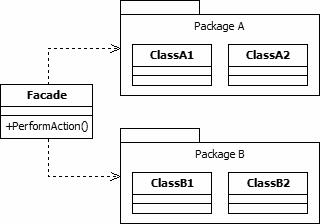
Decorator pattern is helpful in providing runtime modification abilities and hence more flexible. It’s easy to maintain and extend when the number of choices are more. The disadvantage of decorator pattern is that it uses a lot of similar kind of objects (decorators).

Decorator pattern is used a lot in Java IO classes, such as FileReader, BufferedReader etc.

1. **Façade [usage -Medium]**

The facade pattern **is used to define a simplified interface to a more complex subsystem.** **Façade defines a higher-level interface that makes the subsystem easier to use. It helps client applications to easily interact with the system**.

**Example:** Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces to get the required database connection and generate reports. But when the complexity increases or the interface behaviour names are confusing, client application will find it difficult to manage it. So**, we can apply Facade pattern here and provide a wrapper interface on top of the existing interface to help client application.**



Facade interface is a lot easier and cleaner way and avoid having a lot of logic at client side**. JDBC Driver Manager Class to get the database connection is a wonderful example of facade pattern**.

*Facade pattern is more like a helper for client applications,* ***it doesn’t hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.***

*Facade pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex****. Subsystem interfaces are not aware of Facade and they shouldn’t have any reference of the Facade interface.***

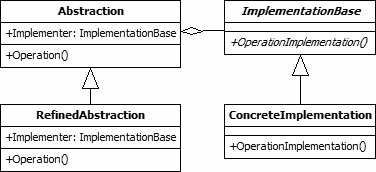
***Facade pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.***

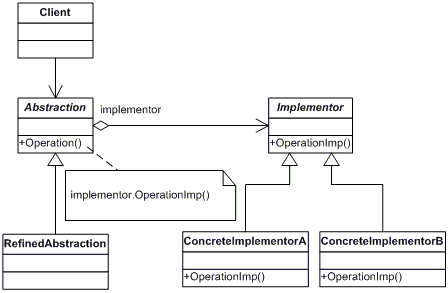
*We can use Factory pattern with Facade to provide better interface to client systems.*

1. **Bridge[usage- Medium]**

**Decouple an abstraction from its implementation so that the two can vary independently.**

1. **ImplementationBase is a bridge interface which decouples Abstraction interface and and ConcreteImplementation**
2. **Abstraction interface aggregates ImplementationBase**





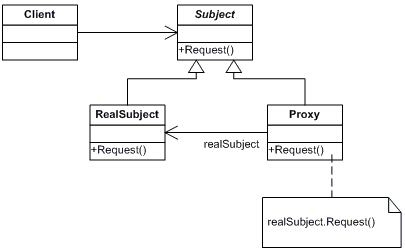
Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

**Example:** Suppose thereare switches to start a number equipment which uses two interface namely, Switch and Equipment interface. To make these two interfaces independent of each other , Switch interface cab have Equipment interface as composition.

https://www.youtube.com/watch?v=bxOY0uBvz38

1. **Proxy [usage- Medium]**

**Provide a surrogate or placeholder for another object to control access to it.**



**Example:** Let’s say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don’t want. Here a proxy class can be created to provide controlled access of the program. Now, we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed.

Proxy pattern common uses are to control access or to provide a wrapper implementation for better performance. **Java RMI** whole package uses proxy pattern.

1. **Flyweight [usage- Low]**

**Use sharing to** **support large numbers of fine-grained objects efficiently**.

Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects.

Before we apply flyweight design pattern, we need to consider following factors:

*The number of Objects to be created by application should be huge.*

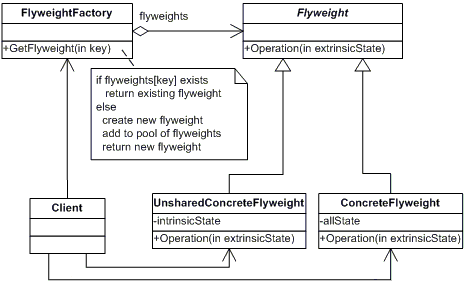
*The object creation is heavy on memory and it can be time consuming*

*too.*

*The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of a Object should be defined by the client program.*

To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. All the wrapper classes valueOf() method uses cached objects showing use of Flyweight design pattern. The best example is Java String class String Pool implementation.



Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade of between memory and time, so we need to use it judiciously based on our requirements.

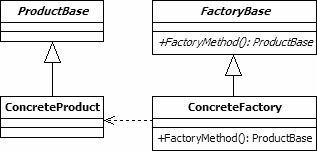
Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

***Creational Design Patterns* [5]**

1. **Factory Method [usage - high]**

**Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.**

Factory design pattern is used when we have a super class with multiple subclasses and based on input, we need to return one of the sub-class. This pattern takes out the responsibility of instantiation of a class from client program to the factory class.



We can keep Factory class Singleton or we can keep the method that returns the subclass as static. Based on the input parameter, different subclass is created and returned.

*Factory pattern provides approach to code for interface rather than implementation.*

*Factory pattern removes the instantiation of actual implementation classes from client code, making it more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.*

Factory meyhod pattern are used in jdk like java.util.Calendar, ResourceBundle and NumberFormat getInstance() methods uses Factory pattern. valueOf() method in wrapper classes like Boolean, Integer etc.

1. **Abstract Factory [usage- high]**

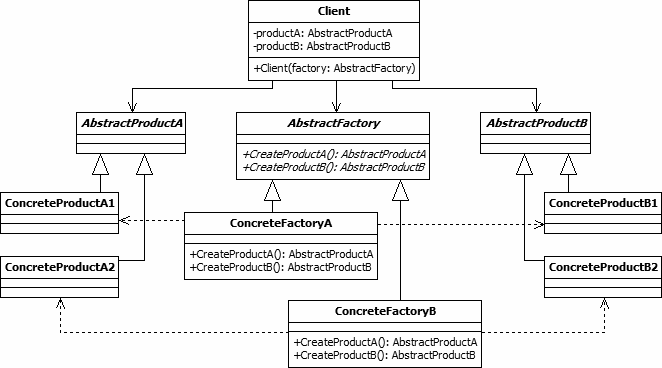
**Provide an interface for creating families of related or dependent objects without specifying their concrete classes.**

It is almost like **Factory Pattern** except the fact that it’s more like factory of factories.

Whenever you need another level of abstraction over a group of factories, you should consider using abstract factory pattern.

If you are familiar with **factory design pattern in java**, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

***In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class***.



Computer pc = com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2 GB","500 GB","2.4 GHz"));

javax.xml.parsers.DocumentBuilderFactory#newInstance()

javax.xml.transform.TransformerFactory#newInstance()

1. **Builder [usage-high]**

**Separate the construction of a complex object from its representation so that the same construction process can create different representations.**

Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will return the final Object**.**

* Create a static nested class and then copy all the arguments from the outer class to the Builder class
* The Builder class should have a public constructor with all the required attributes as parameters
* Builder class should have methods to set the optional parameters and it should ***return the same Builder object after setting the optional attribute***.
* The final step is to provide a *build()* method in the builder class that will return the Object needed by client program***. For this we need to have a private constructor in the Class with Builder class as argument****.*

java.lang.StringBuilder#append() (unsynchronized)

java.lang.StringBuffer#append() (synchronized)

1. **Prototype [usage-low]**

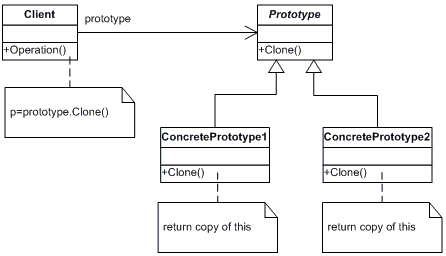
**The prototype pattern is used to instantiate a new object by copying all of the properties of an existing object, creating an independent clone**

Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. *This pattern uses java cloning to copy the object.*

***It would be easy to understand this pattern with an example, suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it’s not a good idea to create the Object using new keyword and load all the data again from database. So the better approach is to clone the existing object into a new object and then do the data manipulation.***

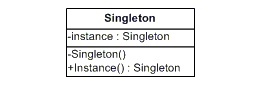
Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should not be done by any other class. ***However, whether to use shallow or deep copy of the Object properties***

***depends on the requirements and it’s a design decision.***



1. **Singleton [usage-high]**

***Ensure a class has only one instance and provide a global point of access to it*.**



The singleton class must provide a global access point to get the instance of the class. Singleton pattern is used for logging, driver objects, caching and thread pool.

To implement Singleton pattern, we have different approaches but all of them have following common concepts.

* Private constructor to restrict instantiation of the class from other classes.
* Private static variable of the same class that is the only instance of the class.
* Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern

implementation and design concerns with the implementation.

* **Eager Initialization**
* **Static block initialization**
* **Lazy Initialization**
* **Bill Pugh Singleton Implementation -**Prior to Java 5, java memory model had a lot of issues and above approaches used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using an **inner private static helper class**.
* **Using Reflection to destroy Singleton Pattern**
* **Enum Singleton -**To overcome this situation with Reflection, Joshua Bloch suggests the use of Enum to implement Singleton design pattern as Java ensures that any enum value is instantiated only once in a Java program. Since Java Enum values are globally accessible, so is the singleton.

## So what's broken about DCL?

class SomeClass {

private Resource resource = null;

public Resource getResource() {

if (resource == null) {

synchronized {

if (resource == null)

resource = new Resource();

}

}

return resource;

}

}

DCL relies on an unsynchronized use of the resource field. That appears to be harmless, but it is not. To see why, imagine that thread A is inside the synchronized block, executing the statement resource = new Resource(); while thread B is just entering getResource(). Consider the effect on memory of this initialization. Memory for the new Resource object will be allocated; the constructor for Resource will be called, initializing the member fields of the new object; and the field resource of SomeClass will be assigned a reference to the newly created object.

However, since thread B is not executing inside a synchronized block, it may see these memory operations in a different order than the one thread A executes. It could be the case that B sees these events in the following order (and the compiler is also free to reorder the instructions like this): allocate memory, assign reference to resource, call constructor. Suppose thread B comes along after the memory has been allocated and the resource field is set, but before the constructor is called. It sees that resource is not null, skips the synchronized block, and returns a reference to a partially constructed Resource! Needless to say, the result is neither expected nor desired.

When presented with this example, many people are skeptical at first. Many highly intelligent programmers have tried to fix DCL so that it does work, but none of these supposedly fixed versions work either. It should be noted that DCL might, in fact, work on some versions of some JVMs -- as few JVMs actually implement the JMM properly. However, you don't want the correctness of your programs to rely on implementation details -- especially errors -- specific to the particular version of the particular JVM you use.

Other concurrency hazards are embedded in DCL -- and in any unsynchronized reference to memory written by another thread, even harmless-looking reads. Suppose thread A has completed initializing the Resource and exits the synchronized block as thread B enters getResource(). Now the Resource is fully initialized, and thread A flushes its local memory out to main memory. The resource's fields may reference other objects stored in memory through its member fields, which will also be flushed out. While thread B may see a valid reference to the newly created Resource, because it didn't perform a read barrier, it could still see stale values of resource's member fields.

#### Volatile doesn't mean what you think, either

A commonly suggested nonfix is to declare the resource field of SomeClass as volatile. However, while the JMM prevents writes to volatile variables from being reordered with respect to one another and ensures that they are flushed to main memory immediately, it still permits reads and writes of volatile variables to be reordered with respect to nonvolatile reads and writes. ***That means -- unless all Resource fields are volatile as well -- thread B can still perceive the constructor's effect as happening after resource is set to reference the newly created Resource.***

## Alternatives to DCL

The most effective way to fix the DCL idiom is to avoid it. The simplest way to avoid it, of course, is to use synchronization. Whenever a variable written by one thread is being read by another, you should use synchronization to guarantee that modifications are visible to other threads in a predictable manner.

Another option for avoiding the problems with DCL is to drop lazy initialization and instead use eager initialization. Rather than delay initialization of resource until it is first used, initialize it at construction. The class loader, which synchronizes on the classes' Class object, executes static initializer blocks at class initialization time. That means that the effect of static initializers is automatically visible to all threads as soon as the class loads.

<http://www.dofactory.com/net/design-patterns>3: <https://www.youtube.com/watch?v=bxOY0uBvz38>

***https://howtodoinjava.com/gang-of-four-java-design-patterns/***

*JournelDev, Dzone*