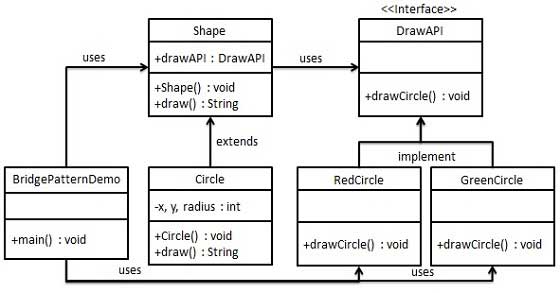
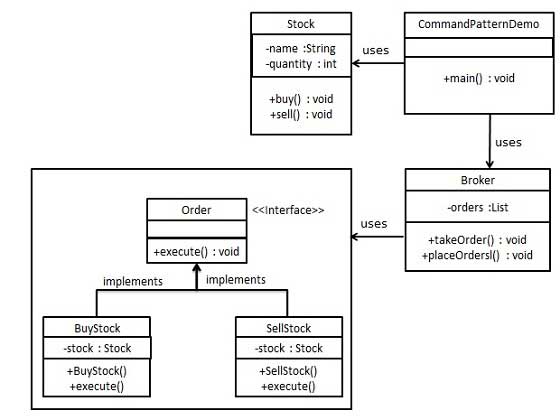
**Bridge and Command Pattern**

Similarities: Both Patterns decouple concrete implementation from client interface.

Differences: In Bridge Concrete Implementation differ in implementation. While in Command, concrete commands differ in functionality.

Example of Bridge Pattern: *DrawAPI* is the bridge. Concrete classes *RedCircle*, *GreenCircle* implement *DrawAPI* interface.*Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, our demo class will use *Shape* class to draw different colored circle. The implementations changes, but the functionality of drawCircle remains same.

Example of Command Pattern: We have created an interface *Order* which is acting as a command. We have created a *Stock* class which acts as a request. We have concrete command classes *BuyStock* and *SellStock* implementing *Order* interface which will do actual command processing. A class *Broker* is created which acts as an invoker object. It can take and place orders. *Broker* object uses command pattern to identify which object will execute which command based on the type of command. *CommandPatternDemo*, our demo class, will use *Broker* class to demonstrate command pattern. The functionality of BuyStock and SellStock are completely different, hence Bridge Pattern cannot be used here.



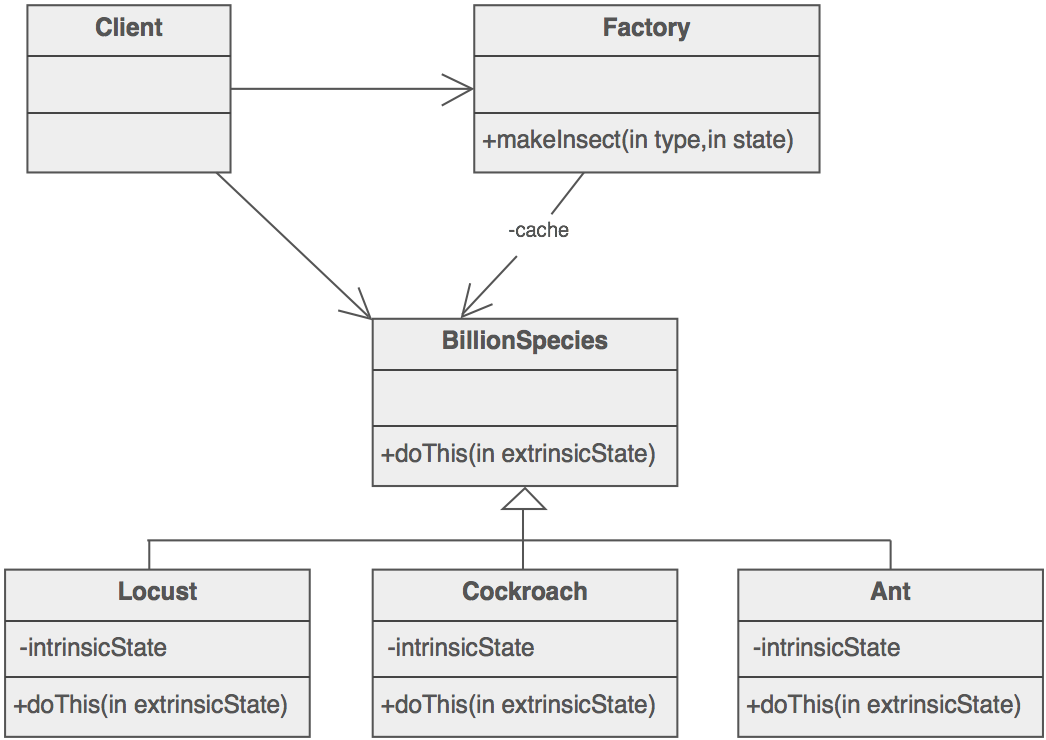
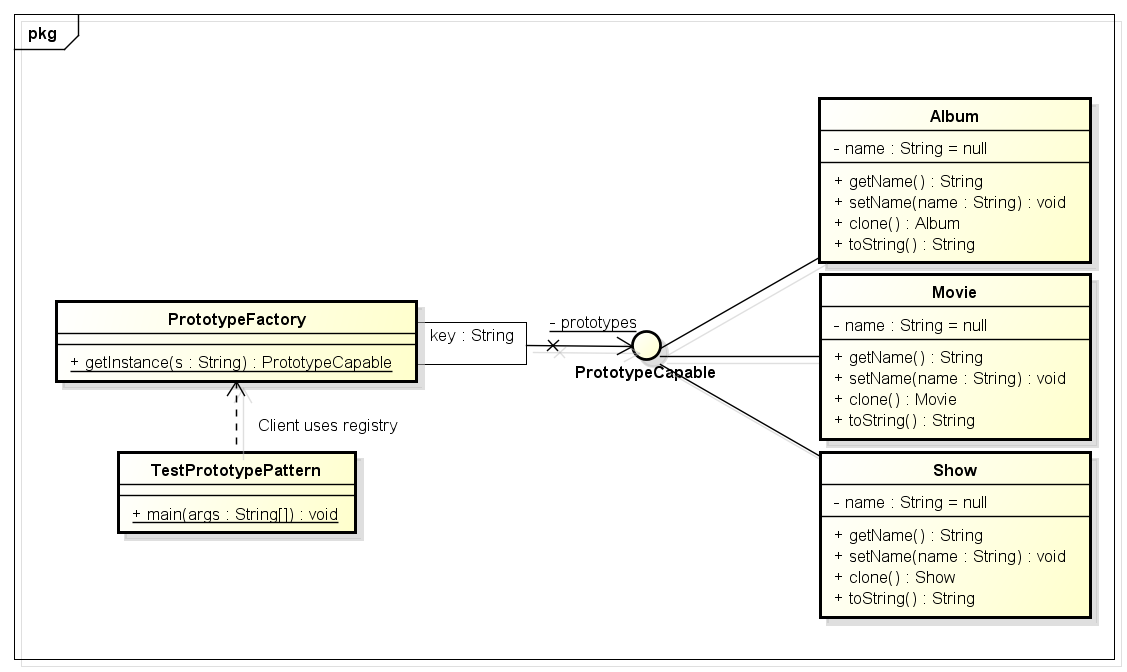
**Flyweight and Prototype Pattern**

Similarities: Both are used to create duplicate objects.

Differences: Flyweight uses minimal memory to create objects, by sharing as much data as possible with similar objects. While, Prototype we can clone the object and change the data as and when required. Also, the concrete commands differ in functionality.

Example of Flyweight Pattern: The Ant, Locust, and Cockroach classes can be "light-weight" because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.

Example of Prototype pattern: Here the user can create a prototype instance of Album, Movie and Show. If the user wants more object, the user can simply clone each of the classes and change the attributes as required.



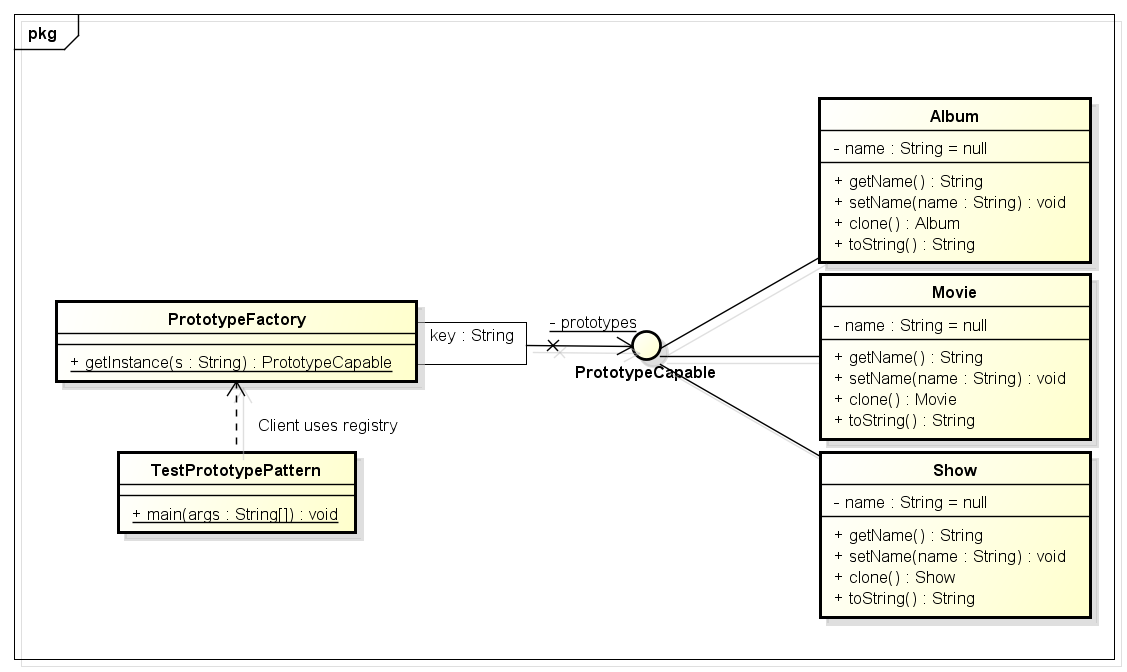
**Singleton and prototype pattern**

Similarities: Singleton is used to create a single object and reuse it as and when required. Prototype is used to duplicate an existing object.

Differences: Singleton limits the number of instances that can be created. Prototype, reduces the number of classes, by having similar behavior.

*Example of Singleton Pattern: SingleObject* class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

Example of Prototype pattern: Here the user can create a prototype instance of Album, Movie and Show. If the user wants more object, the user can simply clone each of the classes and change the attributes as required.



**Abstract factory and factory Method**

Similarities: Both the Abstract Factory Pattern and the Factory Method Pattern decouples the client system from the actual implementation classes through the abstract types and factories. The Factory Method creates objects through inheritance where the Abstract Factory creates objects through composition.

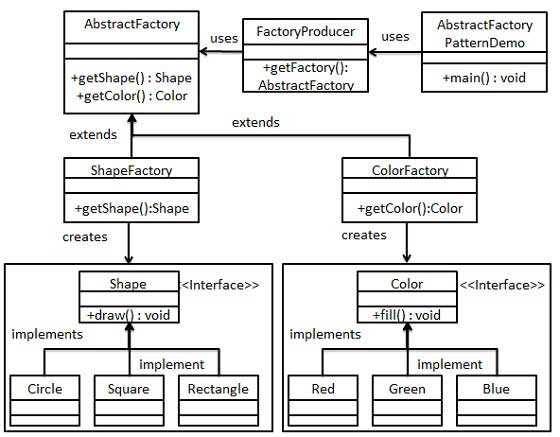
Differences: Factory Method uses interfaces and abstract classes to decouple the client from the generator class and the resulting products. Abstract Factory has a generator that is a container for several factory methods, along with interfaces decoupling the client from the generator and the products.

Example of Factory Method: *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

*FactoryPatternDemo*, will use *ShapeFactory* to get a *Shape*object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to*ShapeFactory* to get the type of object it needs.

Example of Abstract Factory Method: *Shape* and *Color* interfaces and concrete classes implementing these interfaces. Next create an abstract factory class *AbstractFactory*. Factory classes *ShapeFactory* and *ColorFactory*are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

*AbstractFactoryPatternDemo*, uses *FactoryProducer* to get an *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*) to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN / BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.

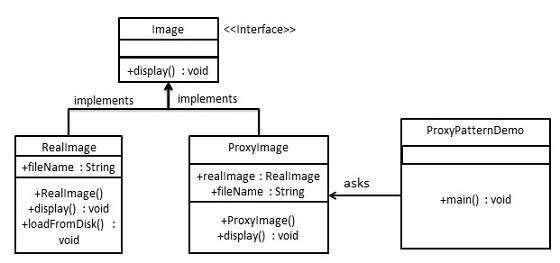


**Adapter and proxy patterns**

Similarities: Both of them use composition and delegation to fulfill their intent. Adapter uses composition to forward calls from target interface to adaptee interface, and Proxy also use composition and delegation to forward requests.

Differences: Involves a single class which is responsible to join functionalities of independent or incompatible objects. In proxy, we create object having original object to interface its functionality to outer world.

*Example of Proxy Pattern: Image interface and concrete classes implementing the Image interface. ProxyImage is a proxy class to reduce memory footprint of RealImage object loading. ProxyPatternDemo*, our demo class, will use *ProxyImage* to get an *Image*object to load and display as it needs.

Example of Adapter Pattern: *MediaPlayer* interface and a concrete class *AudioPlayer*implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default. Another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files. *To make AudioPlayer* to play other formats as well, it implements  *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format. *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format.*AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.



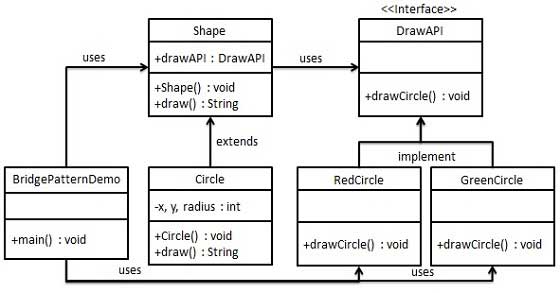
**Bridge and Strategy Pattern:**

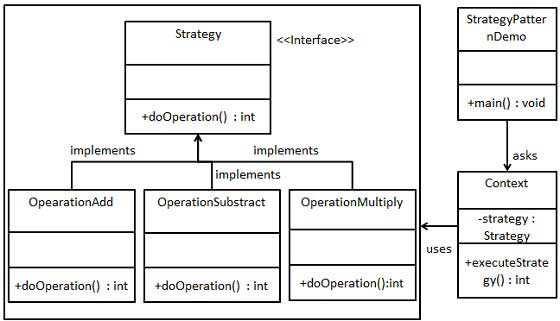
Similarities: both patterns are uses of inheritance to add flexibility. In both patterns, one class (the Container) has a reference to an Interface or base class, which is obviously intended to be subclassed.

Differences: The main difference is that in Bridge, the abstraction is part of the object. In Strategy the abstraction is performed by the object. The strategy is concerned with making algorithms interchangeable while the Bridge is concerned with decoupling the abstraction from the inplementation so that you can provide multiple implementations for the same abstraction. That is, the bridge is concerned with entire structures.

*Example of Bridge Pattern: DrawAPI* interface which is a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implement *DrawAPI* interface.*Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, will use *Shape* class to draw different colored circle.

Example of Strategy Pattern: *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a Strategy. *StrategyPatternDemo*, will use *Context* and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.





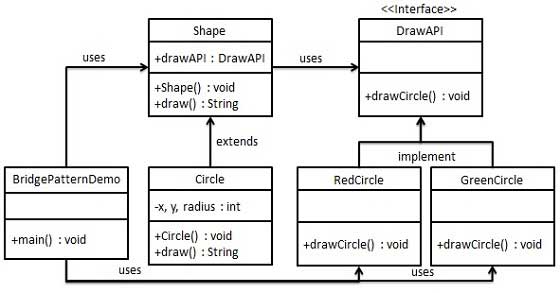
**Adapter and Bridge Pattern:**

Similarities: They both mask the underlying  
implementation.In bridge the clients of the class see the same interface, where as in adaptor the adaptor interface is used to mask another one.

Differences: The main difference between an adaptor and a bridge pattern, is that a bridge pattern serves to decouple an abstraction class from its implementation, and an adaptor pattern converts the interface between classes with less inheritance.

*Example of Bridge Pattern: DrawAPI* interface which is acting as a bridge implementer and concrete classes *RedCircle*, *GreenCircle* implementing the *DrawAPI* interface.*Shape* is an abstract class and will use object of *DrawAPI*. *BridgePatternDemo*, our demo class will use *Shape* class to draw different colored circle.

Example of Adapter Pattern: *MediaPlayer* interface and a concrete class *AudioPlayer*implementing the *MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default. Another interface *AdvancedMediaPlayer* and concrete classes implementing the *AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files. *To make AudioPlayer* to play other formats as well, it implements  *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format. *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format.*AdapterPatternDemo*, our demo class will use *AudioPlayer* class to play various formats.



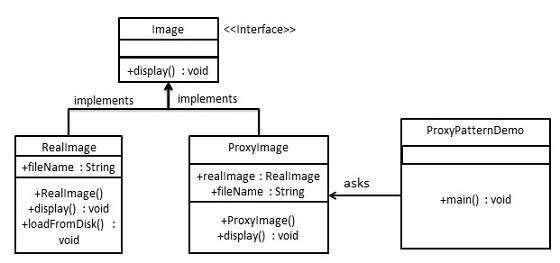
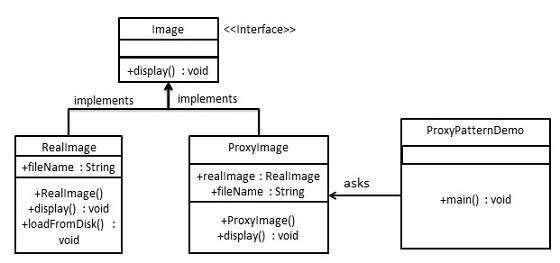
**Decorator and Proxy Pattern:**

Similarities: Both implement interfaces of a real object. Because of this, a decorator and proxy can be passed to a method, which accepts original or real object.

Differences: Decorator pattern allows a user to add new functionality to an existing object without altering its structure. In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern

*Example of Proxy Pattern: Image interface and concrete classes implementing the Image interface. ProxyImage is a proxy class to reduce memory footprint of RealImage object loading. ProxyPatternDemo*, our demo class, will use *ProxyImage* to get an *Image*object to load and display as it needs.

Example of Decorator Pattern: a *Shape* interface and concrete classes implementing the *Shape* interface. We will then create an abstract decorator class*ShapeDecorator* implementing the *Shape* interface and having *Shape* object as its instance variable. *RedShapeDecorator* is concrete class implementing *ShapeDecorator*. *DecoratorPatternDemo*, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.



**Code:**

**CarAudi.java**

public class CarAudi implements Prototype{

private String make;

private String model;

public CarAudi() {

make = "Audi";

model = "2015";

}

@Override

public Prototype clone() {

return new CarAudi();

}

public String getMake() {

return make;

}

public void setMake(String make) {

this.make = make;

}

public String getModel() {

return model;

}

public void setModel(String model) {

this.model = model;

}

public void display() {

System.out.println("I am " + this.make + " " + this.model);

}

}

**CarBMW.java**

public class CarBMW implements Prototype {

private String make;

private String model;

public CarBMW() {

make = "BMW";

model = "2015";

}

@Override

public Prototype clone() {

return new CarBMW();

}

public String getMake() {

return make;

}

public void setMake(String make) {

this.make = make;

}

public String getModel() {

return model;

}

public void setModel(String model) {

this.model = model;

}

public void display(){

System.out.println("I am " + this.make + " " + this.model);

}

}

**CarVW.java**

public class CarVW implements Prototype {

private String make;

private String model;

public CarVW() {

make = "VW";

model = "2015";

}

@Override

public Prototype clone() {

return new CarVW();

}

public String getMake() {

return make;

}

public void setMake(String make) {

this.make = make;

}

public String getModel() {

return model;

}

public void setModel(String model) {

this.model = model;

}

public void display(){

System.out.println("I am " + this.make + " " + this.model);

}

}

**CarPrototypeMgr.java**

import java.util.HashMap;

public class CarPrototypeMgr {

static HashMap<String, Prototype> carMap = new HashMap<String, Prototype>();

public static CarPrototypeMgr getInstance(){

return new CarPrototypeMgr();

}

public void addCar(String key, Prototype p){

carMap.put(key, p);

}

public Prototype getCar(String key){

return carMap.get(key);

}

}

**DocBook .java**

public class DocBook implements Prototype{

private String name;

private String year;

public DocBook() {

name = "Book";

year = "2015";

}

@Override

public Prototype clone() {

return new CarBMW();

}

public String getname() {

return name;

}

public void setname(String name) {

this.name = name;

}

public String getyear() {

return year;

}

public void setyear(String year) {

this.year = year;

}

public void display() {

System.out.println("I am " + this.name + " " + this.year);

}

}

**DocConfProc .java**

public class DocConfProc implements Prototype {

private String name;

private String year;

public DocConfProc() {

name = "Conference Proceedings";

year = "2015";

}

@Override

public Prototype clone() {

return new CarBMW();

}

public String getname() {

return name;

}

public void setname(String name) {

this.name = name;

}

public String getyear() {

return year;

}

public void setyear(String year) {

this.year = year;

}

public void display() {

System.out.println("I am " + this.name + " " + this.year);

}

}

**DocPeriodicals.java**

public class DocPeriodicals implements Prototype{

private String name;

private String year;

public DocPeriodicals() {

name = "Periodicals";

year = "2015";

}

@Override

public Prototype clone() {

return new CarBMW();

}

public String getname() {

return name;

}

public void setname(String name) {

this.name = name;

}

public String getyear() {

return year;

}

public void setyear(String year) {

this.year = year;

}

public void display() {

System.out.println("I am " + this.name + " " + this.year);

}

}

**DocumentPrototypeMgr.java**

import java.util.HashMap;

public class DocumentPrototypeMgr {

static HashMap<String, Prototype> documentMap = new HashMap<String, Prototype>();

protected DocumentPrototypeMgr(){

}

public static DocumentPrototypeMgr getInstance() {

return new DocumentPrototypeMgr();

}

public void addDocument(String key, Prototype p) {

documentMap.put(key, p);

}

public Prototype getDocument(String key) {

return documentMap.get(key);

}

}

**Prototype.java**

public interface Prototype {

public Prototype clone();

}

**References**

Definitions, similarities and differences are referred from the textbook.

<https://en.wikipedia.org/wiki/Bridge_pattern>

<http://www.tutorialspoint.com/design_pattern/bridge_pattern.htm>

<http://www.tutorialspoint.com/design_pattern/command_pattern.htm>

<http://howtodoinjava.com/2013/01/04/prototype-design-pattern-in-java/>

<https://sourcemaking.com/design_patterns/flyweight>

<http://howtodoinjava.com/2013/01/04/prototype-design-pattern-in-java/>

<http://stackoverflow.com/questions/1001767/what-is-the-basic-difference-between-factory-and-abstract-factory-patterns>

<http://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

<http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm>

<http://javarevisited.blogspot.com/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html>

<http://stackoverflow.com/questions/5863530/strategy-vs-bridge-patterns>

<http://game-engineering.blogspot.com/2008/07/bridge-pattern-vs-strategy-pattern.html>

<http://www.tutorialspoint.com/design_pattern/strategy_pattern.htm>

http://www.coderanch.com/t/99919/patterns/Difference-Adapter-Pattern-Bridge-pattern

<http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm>