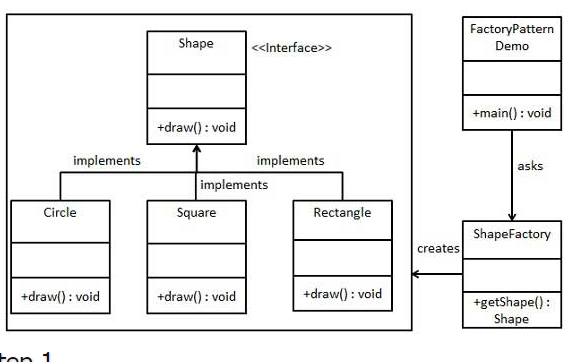
[Creational Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial" \l "creational-patterns)

1. [Singleton Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#singleton-pattern)
2. [Factory Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#factory-pattern)
3. [Abstract Factory Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#abstract-factory-pattern)
4. [Builder Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#builder-pattern)
5. [Prototype Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#prototype-pattern)

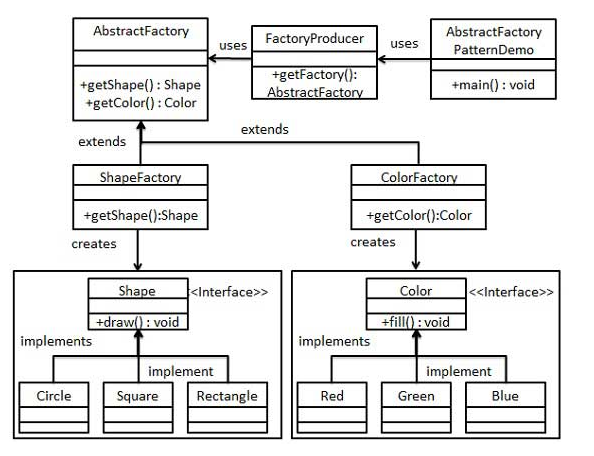
Factory pattern

we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



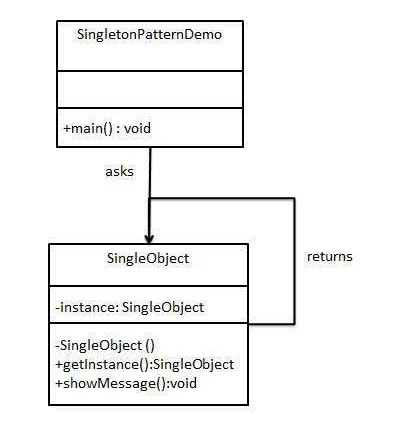
Abstract factory pattern:

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.



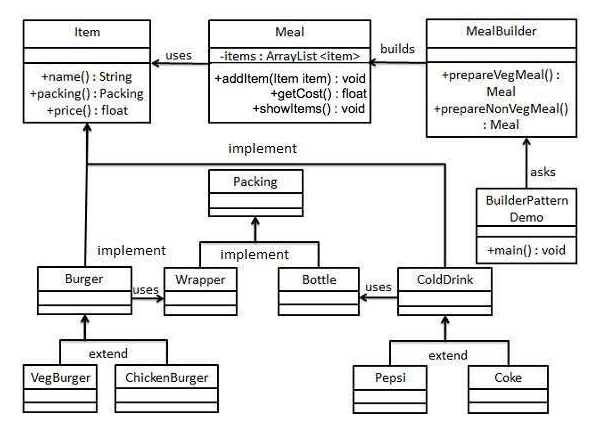
Single ton:

s pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.



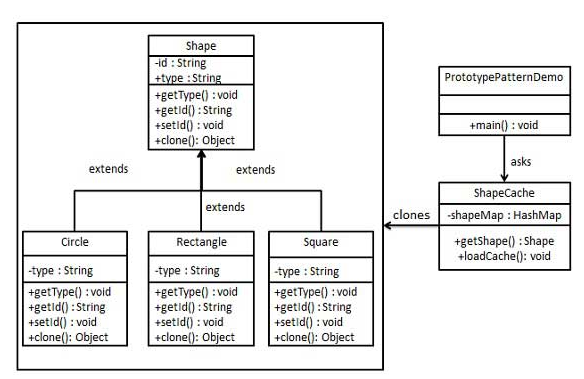
Builder pattern:

A Builder class builds the final object step by step. This builder is independent of other objects.



Prototype pattern:

 pattern involves implementing a prototype interface which tells to create a clone of the current object.

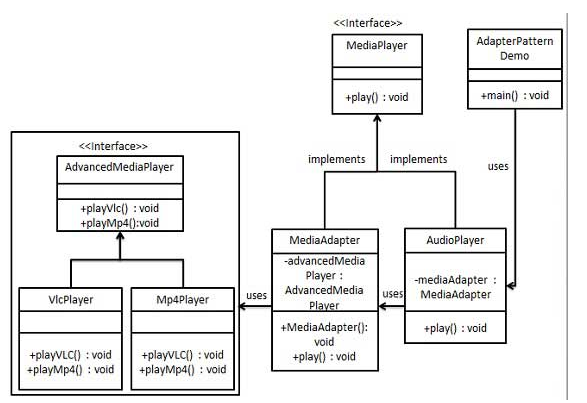


[Structural Design Patterns](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#structural-patterns)

1. [Adapter Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#adapter-pattern)
2. [Composite Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#composite-pattern)
3. [Proxy Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#proxy-pattern)
4. [Flyweight Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#flyweight-pattern)
5. [Facade Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#facade-pattern)
6. [Bridge Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#bridge-pattern)
7. [Decorator Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#decorator-pattern)

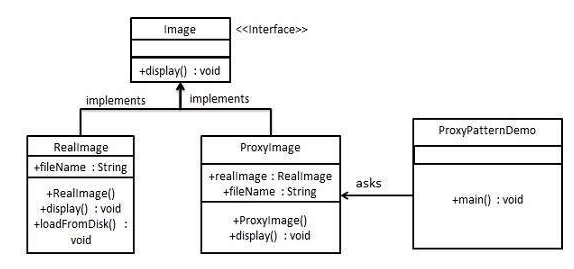
Adapter pattern:

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.



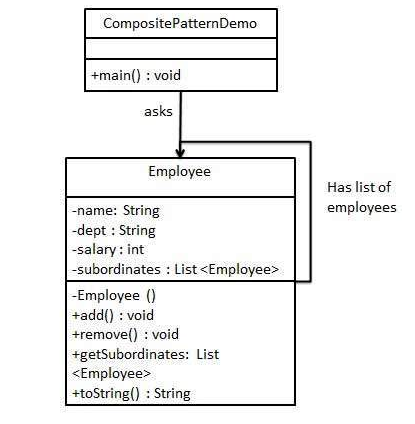
[Proxy Pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial#proxy-pattern)

we create object having original object to interface its functionality to outer world



Composite pattern:

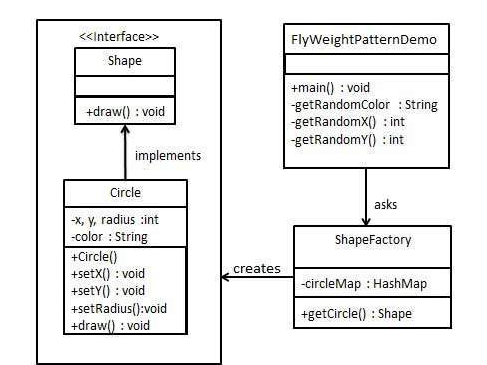
This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects. composes objects in term of a tree structure to represent part as well as whole hierarchy(employees hierarchy)



Flyweight design pattern:

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

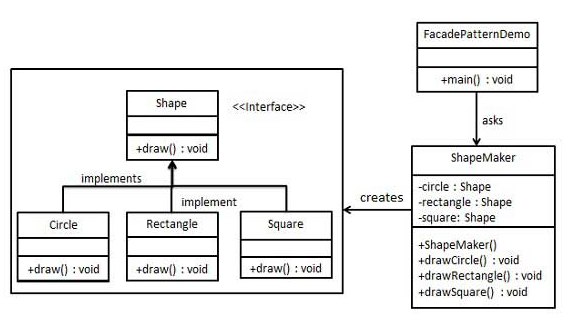
Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found. We will demonstrate this pattern by drawing 20 circles of different locations but we will create only 5 objects. Only 5 colors are available so color property is used to check already existing *Circle* objects.



. Facade pattern:

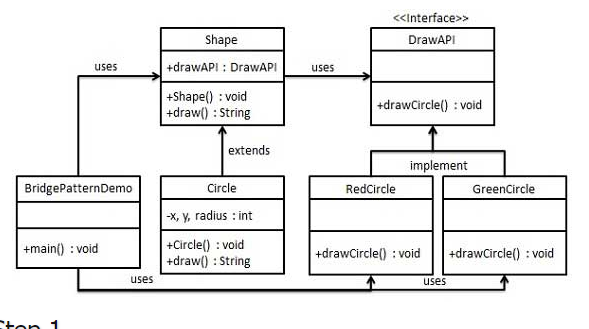
Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.



Bridge pattern:

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other



Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

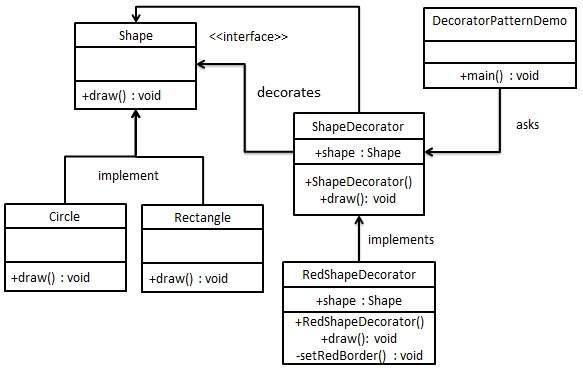
We are demonstrating the use of decorator pattern via following example in which we will decorate a shape with some color without alter shape class.

Implementation

We're going to create 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.



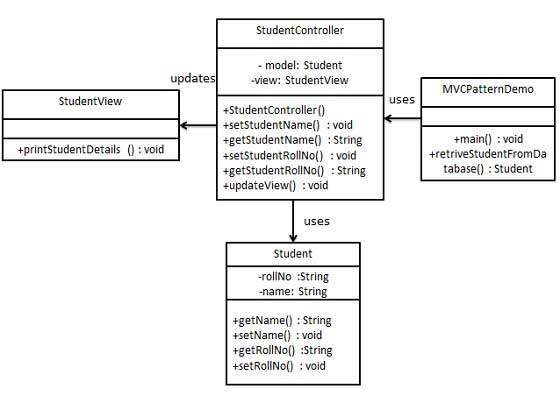
MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

* **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
* **View** - View represents the visualization of the data that model contains.
* **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

## Implementation

We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController*is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.

*MVCPatternDemo*, our demo class, will use *StudentController* to demonstrate use of MVC pattern.



Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

* **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
* **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
* **Model Object or Value Object** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

## Implementation

We are going to create a *Student* object acting as a Model or Value Object.*StudentDao* is Data Access Object Interface.*StudentDaoImpl* is concrete class implementing Data Access Object Interface. *DaoPatternDemo*, our demo class, will use *StudentDao* to demonstrate the use of Data Access Object pattern.



front controller design pattern :

The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

* **Front Controller** - Single handler for all kinds of requests coming to the application (either web based/ desktop based).
* **Dispatcher** - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.
* **View** - Views are the object for which the requests are made.

## Implementation

We are going to create a *FrontController* and *Dispatcher* to act as Front Controller and Dispatcher correspondingly. *HomeView* and *StudentView*represent various views for which requests can come to front controller.

*FrontControllerPatternDemo*, our demo class, will use *FrontController* to demonstrate Front Controller Design Pattern.

