1. **Adapter Pattern**

This pattern is easy to understand as the real world is full of adapters.   For example consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other. we use an adapter that converts one to other. This example is pretty analogous to Object Oriented Adapters. In design, adapters are used when we have a class (Client) expecting some type of object and we have an object (Adaptee) offering the same features but exposing a different interface.

To use an adapter:

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates that request on the adaptee using the adaptee interface.
3. Client receive the results of the call and is unaware of adapter’s presence.

***Definition:***

The adapter pattern convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

**Class Diagram:**  


The client sees only the target interface and not the adapter. The adapter implements the target interface. Adapter delegates all requests to Adaptee.

**Example:**

Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with squeak() method. Let’s assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can’t use them directly. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.

Below is Java implementation of it.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java implementation of Adapter pattern    interface Bird  {      // birds implement Bird interface that allows      // them to fly and make sounds adaptee interface      public void fly();      public void makeSound();  }    class Sparrow implements Bird  {      // a concrete implementation of bird      public void fly()      {          System.out.println("Flying");      }      public void makeSound()      {          System.out.println("Chirp Chirp");      }  }    interface ToyDuck  {      // target interface      // toyducks dont fly they just make      // squeaking sound      public void squeak();  }    class PlasticToyDuck implements ToyDuck  {      public void squeak()      {          System.out.println("Squeak");      }  }    class BirdAdapter implements ToyDuck  {      // You need to implement the interface your      // client expects to use.      Bird bird;      public BirdAdapter(Bird bird)      {          // we need reference to the object we          // are adapting          this.bird = bird;      }        public void squeak()      {          // translate the methods appropriately          bird.makeSound();      }  }    class Main  {      public static void main(String args[])      {          Sparrow sparrow = new Sparrow();          ToyDuck toyDuck = new PlasticToyDuck();            // Wrap a bird in a birdAdapter so that it          // behaves like toy duck          ToyDuck birdAdapter = new BirdAdapter(sparrow);            System.out.println("Sparrow...");          sparrow.fly();          sparrow.makeSound();            System.out.println("ToyDuck...");          toyDuck.squeak();            // toy duck behaving like a bird          System.out.println("BirdAdapter...");          birdAdapter.squeak();      }  } |

Output:

Sparrow...

Flying

Chirp Chirp

ToyDuck...

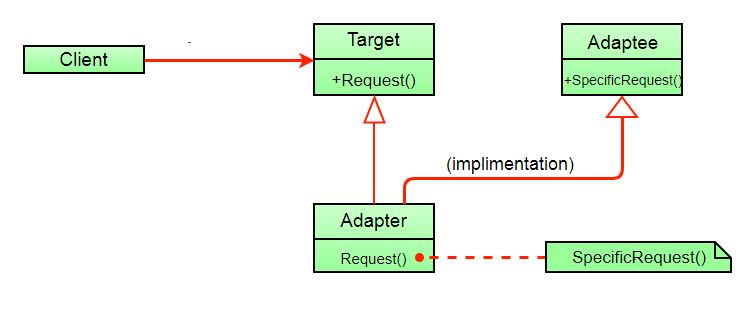
Squeak

BirdAdapter...

Chirp Chirp

**Explanation :**   
Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound than ?  
Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.  
**Before :** ToyDuck toyDuck = new PlasticToyDuck();  
**After :** ToyDuck toyDuck = new BirdAdapter(sparrow);  
You can see that by changing just one line the toyDuck can now do Chirp Chirp !!

**Object Adapter Vs Class Adapter**  
The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee. There is also another type called Class Adapter Pattern which use inheritance instead of composition but you require multiple inheritance to implement it.  
Class diagram of Class Adapter Pattern:



Here instead of having an adaptee object inside adapter (composition) to make use of its functionality adapter inherits the adaptee.

Since multiple inheritance is not supported by many languages including java and is associated with many problems we have not shown implementation using class adapter pattern.

**Advantages:**

* Helps achieve reusability and flexibility.
* Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

**Disadvantages:**

* All requests are forwarded, so there is a slight increase in the overhead.
* Sometimes many adaptations are required along an adapter chain to reach the type which is required.

1. **Facade Design Pattern | Introduction**

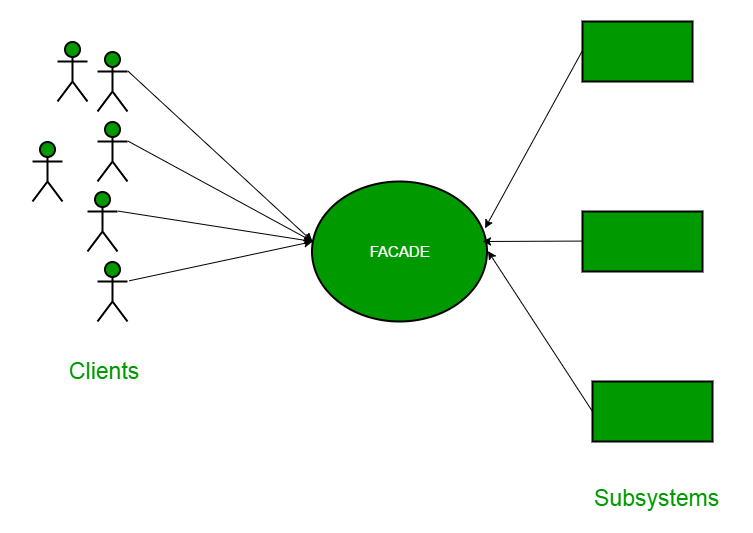
Facade is a part of Gang of Four design pattern and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern.

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

**More examples**

In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.  
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.

[](https://media.geeksforgeeks.org/wp-content/uploads/facadeA.png)

**Facade Design Pattern Diagram**

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants.  
You, as client want access to different menus of different restaurants . You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.  
Let’s see how it works :

**Interface of Hotel**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;  public interface Hotel  {      public Menus getMenus();  } |

The hotel interface only returns Menus.  
Similarly, the Restaurant are of three types and can implement the hotel interface. Let’s have a look at the code for one of the Restaurants.

**NonVegRestaurant.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;    public class NonVegRestaurant implements Hotel  {      public Menus getMenus()      {          NonVegMenu nv = new NonVegMenu();          return nv;      }  } |

**VegRestaurant.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;    public class VegRestaurant implements Hotel  {      public Menus getMenus()      {          VegMenu v = new VegMenu();          return v;      }  } |

**VegNonBothRestaurant.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;    public class VegNonBothRestaurant implements Hotel  {      public Menus getMenus()      {          Both b = new Both();          return b;      }  } |

Now let’s consider the facade,

**HotelKeeper.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;    public class HotelKeeper  {      public VegMenu getVegMenu()      {          VegRestaurant v = new VegRestaurant();          VegMenu vegMenu = (VegMenu)v.getMenus();          return vegMenu;      }        public NonVegMenu getNonVegMenu()      {          NonVegRestaurant v = new NonVegRestaurant();          NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();          return NonvegMenu;      }        public Both getVegNonMenu()      {          VegNonBothRestaurant v = new VegNonBothRestaurant();          Both bothMenu = (Both)v.getMenus();          return bothMenu;      }  } |

From this, It is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

**How will the client program access this façade?**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package structural.facade;    public class Client  {      public static void main (String[] args)      {          HotelKeeper keeper = new HotelKeeper();            VegMenu v = keeper.getVegMenu();          NonVegMenu nv = keeper.getNonVegMenu();          Both = keeper.getVegNonMenu();        }  } |

In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

**When Should this pattern be used?**

The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

# Design Patterns | Set 2 (Factory Method)

Factory method is a [creational design pattern](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/), i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.  
The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn’t have tight coupling with the class hierarchy of the library.

***What is meant when we talk about library and clients?***  
A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.

***Why factory pattern?***  
Let us understand it with an example:

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // A design without factory pattern  #include <iostream>  using namespace std;    // Library classes  class Vehicle {  public:      virtual void printVehicle() = 0;  };  class TwoWheeler : public Vehicle {  public:      void printVehicle()  {          cout << "I am two wheeler" << endl;      }  };  class FourWheeler : public Vehicle {      public:      void printVehicle()  {          cout << "I am four wheeler" << endl;      }  };    // Client (or user) class  class Client {  public:      Client(int type)  {            // Client explicitly creates classes according to type          if (type == 1)              pVehicle = new TwoWheeler();          else if (type == 2)              pVehicle = new FourWheeler();          else              pVehicle = NULL;      }        ~Client()   {          if (pVehicle)          {              delete[] pVehicle;              pVehicle = NULL;          }      }        Vehicle\* getVehicle() {          return pVehicle;      }  private:      Vehicle \*pVehicle;  };    // Driver program  int main() {      Client \*pClient = new Client(1);      Vehicle \* pVehicle = pClient->getVehicle();      pVehicle->printVehicle();      return 0;  } |

Output:

I am two wheeler

***What is the problems with above design?***  
As you must have observed in the above example, Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing its object.  
Say, library introduces a new class ThreeWheeler to incorporate three wheeler vehicles also. What would happen? Client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design.

**How to avoid the problem?**  
The answer is, create a static (or factory) method. Let us see below code.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // C++ program to demonstrate factory method design pattern  #include <iostream>  using namespace std;    enum VehicleType {      VT\_TwoWheeler,    VT\_ThreeWheeler,    VT\_FourWheeler  };    // Library classes  class Vehicle {  public:      virtual void printVehicle() = 0;      static Vehicle\* Create(VehicleType type);  };  class TwoWheeler : public Vehicle {  public:      void printVehicle() {          cout << "I am two wheeler" << endl;      }  };  class ThreeWheeler : public Vehicle {  public:      void printVehicle() {          cout << "I am three wheeler" << endl;      }  };  class FourWheeler : public Vehicle {      public:      void printVehicle() {          cout << "I am four wheeler" << endl;      }  };    // Factory method to create objects of different types.  // Change is required only in this function to create a new object type  Vehicle\* Vehicle::Create(VehicleType type) {      if (type == VT\_TwoWheeler)          return new TwoWheeler();      else if (type == VT\_ThreeWheeler)          return new ThreeWheeler();      else if (type == VT\_FourWheeler)          return new FourWheeler();      else return NULL;  }    // Client class  class Client {  public:        // Client doesn't explicitly create objects      // but passes type to factory method "Create()"      Client()      {          VehicleType type = VT\_ThreeWheeler;          pVehicle = Vehicle::Create(type);      }      ~Client() {          if (pVehicle) {              delete[] pVehicle;              pVehicle = NULL;          }      }      Vehicle\* getVehicle()  {          return pVehicle;      }    private:      Vehicle \*pVehicle;  };    // Driver program  int main() {      Client \*pClient = new Client();      Vehicle \* pVehicle = pClient->getVehicle();      pVehicle->printVehicle();      return 0;  } |

Output:

I am three wheeler

In the above example, we have totally decoupled the selection of type for object creation from Client. The library is now responsible to decide which object type to create based on an input. Client just needs to make call to library’s factory Create method and pass the type it wants without worrying about the actual implementation of creation of objects.

Thanks to Rumplestiltskin for providing above explanation.

**Other examples of Factory Method:**

1. Say, in a ‘Drawing’ system, depending on user’s input, different pictures like square, rectangle, circle can be drawn. Here we can use factory method to create instances depending on user’s input. For adding new type of shape, no need to change client’s code.
2. Another example: In travel site, we can book train ticket as well bus tickets and flight ticket. In this case user can give his travel type as ‘bus’, ‘train’ or ‘flight’.  
   Here we have an abstract class ‘AnyTravel’ with a static member function ‘GetObject’ which depending on user’s travel type, will create & return object of ‘BusTravel’ or ‘ TrainTravel’. ‘BusTravel’ or ‘ TrainTravel’ have common functions like passenger name, Origin, destinationparameters.

# Abstract Factory Pattern

**Introduction**

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type.

**Let see the GOFs representation of Abstract Factory Pattern :**  
  
UML class diagram example for the Abstract Factory Design Pattern.

* **AbstractFactory** : Declares an interface for operations that create abstract product objects.
* **ConcreteFactory** : Implements the operations declared in the AbstractFactory to create concrete product objects.
* **Product** : Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
* **Client** : Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects.  
The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.

So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects.

**Implementation**

Let’s take an example, Suppose we want to build a global car factory. If it was [factory design pattern](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/), then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.

We need car factories in each location like IndiaCarFactory, USACarFactory and DefaultCarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling wrong factory for a particular location.

Here we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to user. This is exactly the problem, which abstract factory pattern is used to solve.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java Program to demonstrate the  // working of Abstract Factory Pattern    enum CarType  {      MICRO, MINI, LUXURY  }    abstract class Car  {      Car(CarType model, Location location)      {          this.model = model;          this.location = location;      }        abstract void construct();        CarType model = null;      Location location = null;        CarType getModel()      {          return model;      }        void setModel(CarType model)      {          this.model = model;      }        Location getLocation()      {          return location;      }        void setLocation(Location location)      {          this.location = location;      }        @Override      public String toString()      {          return "CarModel - "+model + " located in "+location;      }  }    class LuxuryCar extends Car  {      LuxuryCar(Location location)      {          super(CarType.LUXURY, location);          construct();      }      @Override      protected void construct()      {          System.out.println("Connecting to luxury car");      }  }    class MicroCar extends Car  {      MicroCar(Location location)      {          super(CarType.MICRO, location);          construct();      }      @Override      protected void construct()      {          System.out.println("Connecting to Micro Car ");      }  }    class MiniCar extends Car  {      MiniCar(Location location)      {          super(CarType.MINI,location );          construct();      }        @Override      void construct()      {          System.out.println("Connecting to Mini car");      }  }    enum Location  {    DEFAULT, USA, INDIA  }    class INDIACarFactory  {      static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.INDIA);                  break;                case MINI:                  car = new MiniCar(Location.INDIA);                  break;                case LUXURY:                  car = new LuxuryCar(Location.INDIA);                  break;                    default:                  break;            }          return car;      }  }    class DefaultCarFactory  {      public static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.DEFAULT);                  break;                case MINI:                  car = new MiniCar(Location.DEFAULT);                  break;                case LUXURY:                  car = new LuxuryCar(Location.DEFAULT);                  break;                    default:                  break;            }          return car;      }  }      class USACarFactory  {      public static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                  car = new MicroCar(Location.USA);                  break;                case MINI:                  car = new MiniCar(Location.USA);                  break;                case LUXURY:                  car = new LuxuryCar(Location.USA);                  break;                    default:                  break;            }          return car;      }  }    class CarFactory  {      private CarFactory()      {        }      public static Car buildCar(CarType type)      {          Car car = null;          // We can add any GPS Function here which          // read location property somewhere from configuration          // and use location specific car factory          // Currently I'm just using INDIA as Location          Location location = Location.INDIA;            switch(location)          {              case USA:                  car = USACarFactory.buildCar(type);                  break;                case INDIA:                  car = INDIACarFactory.buildCar(type);                  break;                default:                  car = DefaultCarFactory.buildCar(type);            }            return car;        }  }    class AbstractDesign  {      public static void main(String[] args)      {          System.out.println(CarFactory.buildCar(CarType.MICRO));          System.out.println(CarFactory.buildCar(CarType.MINI));          System.out.println(CarFactory.buildCar(CarType.LUXURY));      }  } |

Output :

Connecting to Micro Car

CarModel - MICRO located in INDIA

Connecting to Mini car

CarModel - MINI located in INDIA

Connecting to luxury car

CarModel - LUXURY located in INDIA

**Difference**

* The main difference between a “factory method” and an “abstract factory” is that the factory method is a single method, and an abstract factory is an object.
* The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.
* The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

**Advantages**

This pattern is particularly useful when the client doesn’t know exactly what type to create.

* **Isolation of concrete classes:** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
* **Exchanging Product Families easily:** The class of a concrete factory appears only once in an application, that is where it’s instantiated. This makes it easy to change the concrete factory an application uses. It can use various product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.
* **Promoting consistency among products:** When product objects in a family are designed to work together, it’s important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.n.

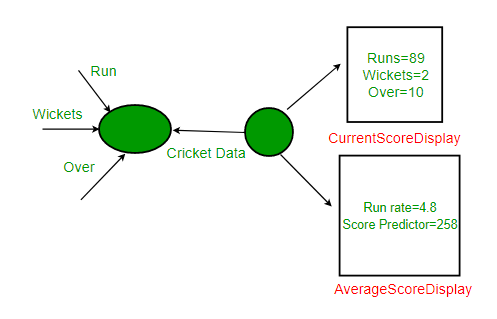
**Disadvantages**

* **Difficult to support new kind of products:** Extending abstract factories to produce new kinds of Products isn’t easy. That’s because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.

# Observer Pattern

Let us first consider the following scenario to understand observer pattern.

**Scenario**:

Suppose we are building a cricket app that notifies viewers about the information such as current score, run rate etc. Suppose we have made two display elements CurrentScoreDisplay and AverageScoreDisplay. CricketData has all the data (runs, bowls etc.) and whenever data changes the display elements are notified with new data and they display the latest data accordingly.[](https://media.geeksforgeeks.org/wp-content/uploads/ObserverPatternSet-1.png)

Below is the java implementation of this design.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java implementation of above design for Cricket App. The  // problems with this design are discussed below.    // A class that gets information from stadium and notifies  // display elements, CurrentScoreDisplay & AverageScoreDisplay  class CricketData  {      int runs, wickets;      float overs;      CurrentScoreDisplay currentScoreDisplay;      AverageScoreDisplay averageScoreDisplay;        // Constructor      public CricketData(CurrentScoreDisplay currentScoreDisplay,                         AverageScoreDisplay averageScoreDisplay)      {          this.currentScoreDisplay = currentScoreDisplay;          this.averageScoreDisplay = averageScoreDisplay;      }        // Get latest runs from stadium      private int getLatestRuns()      {          // return 90 for simplicity          return 90;      }        // Get latest wickets from stadium      private int getLatestWickets()      {          // return 2 for simplicity          return 2;      }        // Get latest overs from stadium      private float getLatestOvers()      {          // return 10.2 for simplicity          return (float)10.2;      }        // This method is used update displays when data changes      public void dataChanged()      {          // get latest data          runs = getLatestRuns();          wickets = getLatestWickets();          overs = getLatestOvers();            currentScoreDisplay.update(runs,wickets,overs);          averageScoreDisplay.update(runs,wickets,overs);      }  }    // A class to display average score. Data of this class is  // updated by CricketData  class AverageScoreDisplay  {      private float runRate;      private int predictedScore;        public void update(int runs, int wickets, float overs)      {          this.runRate = (float)runs/overs;          this.predictedScore = (int) (this.runRate \* 50);          display();      }        public void display()      {          System.out.println("\nAverage Score Display:\n" +                             "Run Rate: " + runRate +                             "\nPredictedScore: " + predictedScore);      }  }    // A class to display score. Data of this class is  // updated by CricketData  class CurrentScoreDisplay  {      private int runs, wickets;      private float overs;        public void update(int runs,int wickets,float overs)      {          this.runs = runs;          this.wickets = wickets;          this.overs = overs;          display();      }        public void display()      {          System.out.println("\nCurrent Score Display: \n" +                             "Runs: " + runs +"\nWickets:"                             + wickets + "\nOvers: " + overs );      }  }    // Driver class  class Main  {      public static void main(String args[])      {          // Create objects for testing          AverageScoreDisplay averageScoreDisplay =                                         new AverageScoreDisplay();          CurrentScoreDisplay currentScoreDisplay =                                         new CurrentScoreDisplay();            // Pass the displays to Cricket data          CricketData cricketData = new CricketData(currentScoreDisplay,                                                    averageScoreDisplay);            // In real app you would have some logic to call this          // function when data changes          cricketData.dataChanged();      }  } |

**Output:**

Current Score Display:

Runs: 90

Wickets:2

Overs: 10.2

Average Score Display:

Run Rate: 8.823529

PredictedScore: 441

**Problems with above design:**

* CricketData holds references to concrete display element objects even though it needs to call only the update method of these objects. It has access to too much additional information than it requires.
* This statement “currentScoreDisplay.update(runs,wickets,overs);” violates one of the most important design principle “Program to interfaces, not implementations.” as we are using concrete objects to share data rather than abstract interfaces.
* CricketData and display elements are tightly coupled.
* If in future another requirement comes in and we need another display element to be added we need to make changes to the non-varying part of our code(CricketData). This is definitely not a good design practice and application might not be able to handle changes and not easy to maintain.

**How to avoid these problems?**  
Use Observer Pattern

**Observer pattern**

To understand observer pattern, first you need to understand the subject and observer objects.

The relation between subject and observer can easily be understood as an analogy to magazine subscription.

* A magazine publisher(subject) is in the business and publishes magazines (data).
* If you(user of data/observer) are interested in the magazine you subscribe(register), and if a new edition is published it gets delivered to you.
* If you unsubscribe(unregister) you stop getting new editions.
* Publisher doesn’t know who you are and how you use the magazine, it just delivers it to you because you are a subscriber(loose coupling).

**Definition:**

The Observer Pattern defines a one to many dependency between objects so that one object changes state, all of its dependents are notified and updated automatically.

**Explanation:**

* One to many dependency is between Subject(One) and Observer(Many).
* There is dependency as Observers themselves don’t have access to data. They are dependent on Subject to provide them data.

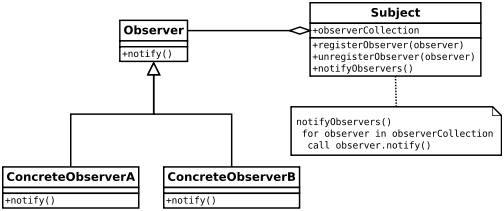
**Class diagram:**

Image Source : [Wikipedia](https://en.wikipedia.org/wiki/Observer_pattern)

* Here Observer and Subject are interfaces(can be any abstract super type not necessarily java interface).
* All observers who need the data need to implement observer interface.
* notify() method in observer interface defines the action to be taken when the subject provides it data.
* The subject maintains an observerCollection which is simply the list of currently registered(subscribed) observers.
* registerObserver(observer) and unregisterObserver(observer) are methods to add and remove observers respectively.
* notifyObservers() is called when the data is changed and the observers need to be supplied with new data.

**Advantages:**  
Provides a loosely coupled design between objects that interact. Loosely coupled objects are flexible with changing requirements. Here loose coupling means that the interacting objects should have less information about each other.

Observer pattern provides this loose coupling as:

* Subject only knows that observer implement Observer interface.Nothing more.
* There is no need to modify Subject to add or remove observers.
* We can reuse subject and observer classes independently of each other.

**Disadvantages:**

* Memory leaks caused by [Lapsed listener problem](https://en.wikipedia.org/wiki/Lapsed_listener_problem) because of explicit register and unregistering of observers.

**When to use this pattern?**  
You should consider using this pattern in your application when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.

**Real Life Uses:**

* It is heavily used in GUI toolkits and event listener. In java the button(subject) and onClickListener(observer) are modelled with observer pattern.
* Social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
* All users of an app on play store gets notified if there is an update.

1. **Singleton Design Pattern**

[Singleton Design Pattern | Introduction](https://www.geeksforgeeks.org/singleton-design-pattern-introduction/)

The singleton pattern is one of the simplest design patterns. Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

**Definition:**  
*The singleton pattern is a design pattern that restricts the instantiation of a class to one object.*  
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

**Method 1: Classic Implementation**

*filter\_none*

*edit*

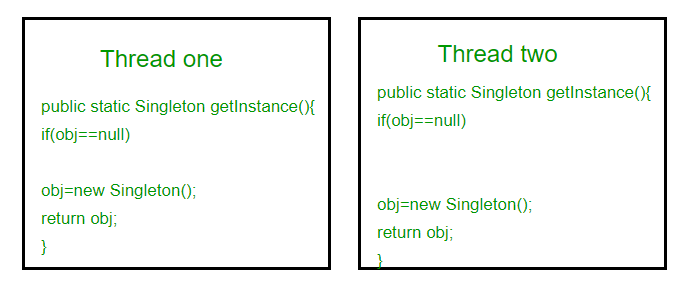
*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Classical Java implementation of singleton  // design pattern  class Singleton  {      private static Singleton obj;        // private constructor to force use of      // getInstance() to create Singleton object      private Singleton() {}        public static Singleton getInstance()      {          if (obj==null)              obj = new Singleton();          return obj;      }  } |

Here we have declared getInstance() static so that we can call it without instantiating the class. The first time getInstance() is called it creates a new singleton object and after that it just returns the same object. Note that Singleton obj is not created until we need it and call getInstance() method. This is called lazy instantiation.

The main problem with above method is that it is not thread safe. Consider the following execution sequence.

[](https://media.geeksforgeeks.org/wp-content/uploads/singleton-1.png)

This execution sequence creates two objects for singleton. Therefore this classic implementation is not thread safe.

**Method 2: make getInstance() synchronized**

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Thread Synchronized Java implementation of  // singleton design pattern  class Singleton  {      private static Singleton obj;        private Singleton() {}        // Only one thread can execute this at a time      public static synchronized Singleton getInstance()      {          if (obj==null)              obj = new Singleton();          return obj;      }  } |

Here using synchronized makes sure that only one thread at a time can execute getInstance().  
The main disadvantage of this is method is that using synchronized every time while creating the singleton object is expensive and may decrease the performance of your program. However if performance of getInstance() is not critical for your application this method provides a clean and simple solution.

**Method 3: Eager Instantiation**

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Static initializer based Java implementation of  // singleton design pattern  class Singleton  {      private static Singleton obj = new Singleton();        private Singleton() {}        public static Singleton getInstance()      {          return obj;      }  } |

Here we have created instance of singleton in static initializer. JVM executes static initializer when the class is loaded and hence this is guaranteed to be thread safe. Use this method only when your singleton class is light and is used throughout the execution of your program.

**Method 4 (Best): Use “**[**Double Checked Locking**](https://en.wikipedia.org/wiki/Double-checked_locking)**”**   
If you notice carefully once an object is created synchronization is no longer useful because now obj will not be null and any sequence of operations will lead to consistent results.  
So we will only acquire lock on the getInstance() once, when the obj is null. This way we only synchronize the first way through, just what we want.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Double Checked Locking based Java implementation of  // singleton design pattern  class Singleton  {      private volatile static Singleton obj;        private Singleton() {}        public static Singleton getInstance()      {          if (obj == null)          {              // To make thread safe              synchronized (Singleton.class)              {                  // check again as multiple threads                  // can reach above step                  if (obj==null)                      obj = new Singleton();              }          }          return obj;      }  } |

We have declared the obj [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) which ensures that multiple threads offer the obj variable correctly when it is being initialized to Singleton instance. This method drastically reduces the overhead of calling the synchronized method every time.

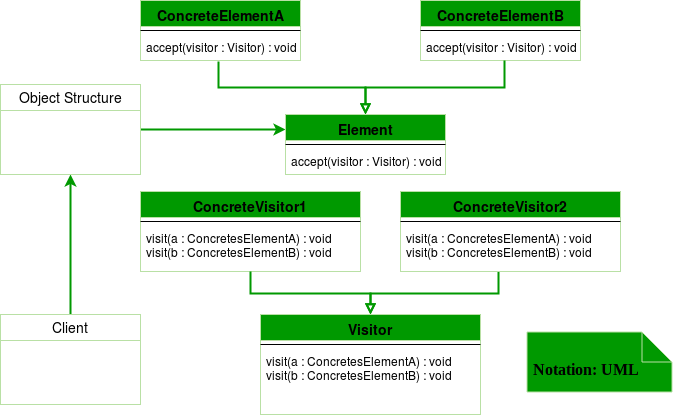
# Visitor design pattern

Visitor design pattern is one of the behavioral design patterns. It 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.

The visitor pattern consists of two parts:

* a method called **Visit()** which is implemented by the visitor and is called for every element in the data structure
* visitable classes providing **Accept()** methods that accept a visitor

**UML Diagram Visitor design pattern**



**Design components**

* **Client :** The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
* **Visitor :** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
* **ConcreteVisitor :** For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.
* **Visitable :** This is an interface which declares the accept operation. This is the entry point which enables an object to be “visited” by the visitor object.
* **ConcreteVisitable :** These classes implement the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.

**Let’s see an example of Visitor design pattern in Java.**

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| interface ItemElement  {      public int accept(ShoppingCartVisitor visitor);  }    class Book implements ItemElement  {      private int price;      private String isbnNumber;        public Book(int cost, String isbn)      {          this.price=cost;          this.isbnNumber=isbn;      }        public int getPrice()      {          return price;      }        public String getIsbnNumber()      {          return isbnNumber;      }        @Override      public int accept(ShoppingCartVisitor visitor)      {          return visitor.visit(this);      }    }    class Fruit implements ItemElement  {      private int pricePerKg;      private int weight;      private String name;        public Fruit(int priceKg, int wt, String nm)      {          this.pricePerKg=priceKg;          this.weight=wt;          this.name = nm;      }        public int getPricePerKg()      {          return pricePerKg;      }        public int getWeight()      {          return weight;      }        public String getName()      {          return this.name;      }        @Override      public int accept(ShoppingCartVisitor visitor)      {          return visitor.visit(this);      }    }    interface ShoppingCartVisitor  {        int visit(Book book);      int visit(Fruit fruit);  }    class ShoppingCartVisitorImpl implements ShoppingCartVisitor  {        @Override      public int visit(Book book)      {          int cost=0;          //apply 5$ discount if book price is greater than 50          if(book.getPrice() > 50)          {              cost = book.getPrice()-5;          }          else              cost = book.getPrice();            System.out.println("Book ISBN::"+book.getIsbnNumber() + " cost ="+cost);          return cost;      }        @Override      public int visit(Fruit fruit)      {          int cost = fruit.getPricePerKg()\*fruit.getWeight();          System.out.println(fruit.getName() + " cost = "+cost);          return cost;      }    }    class ShoppingCartClient  {        public static void main(String[] args)      {          ItemElement[] items = new ItemElement[]{new Book(20, "1234"),                                new Book(100, "5678"), new Fruit(10, 2, "Banana"),                                new Fruit(5, 5, "Apple")};            int total = calculatePrice(items);          System.out.println("Total Cost = "+total);      }        private static int calculatePrice(ItemElement[] items)      {          ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();          int sum=0;          for(ItemElement item : items)          {              sum = sum + item.accept(visitor);          }          return sum;      }    } |

**Output:**

Book ISBN::1234 cost =20

Book ISBN::5678 cost =95

Banana cost = 20

Apple cost = 25

Total Cost = 160

Here, in the implementation if accept() method in all the items are same but it can be different. For example there can be logic to check if item is free then don’t call the visit() method at all.

**Advantages :**

* If the logic of operation changes, then we need to make change only in the visitor implementation rather than doing it in all the item classes.
* Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

**Disadvantages :**

* We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
* If there are too many implementations of visitor interface, it makes it hard to extend.

**Another example of visitor pattern in C++**

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| //Write CPP code here    #include <iostream>  using namespace std;    class Stock  {    public:      virtual void accept(class Visitor \*) = 0;    };    class Apple : public Stock  {    public:      /\*virtual\*/ void accept(Visitor \*);      void buy()      {          cout << "Apple::buy\n";      }      void sell()      {          cout << "Apple::sell\n";      }    };  class Google : public Stock  {    public:      /\*virtual\*/ void accept(Visitor \*);      void buy()      {          cout << "Google::buy\n";      }        void sell()      {          cout << "Google::sell\n";      }  };    class Visitor  {    public:      virtual void visit(Apple \*) = 0;      virtual void visit(Google \*) = 0;      //private:      static int m\_num\_apple, m\_num\_google;      void total\_stocks()      {          cout << "m\_num\_apple " << m\_num\_apple               << ", m\_num\_google " << m\_num\_google << '\n';      }  };  int Visitor::m\_num\_apple = 0;  int Visitor::m\_num\_google = 0;  class BuyVisitor : public Visitor  {    public:      BuyVisitor()      {          m\_num\_apple = m\_num\_google = 0;      }      /\*virtual\*/ void visit(Apple \*r)      {          ++m\_num\_apple;          r->buy();          cout << "m\_num\_apple " << m\_num\_apple << endl;      }      /\*virtual\*/ void visit(Google \*b)      {          ++m\_num\_google;          b->buy();          cout << " m\_num\_google " << m\_num\_google << '\n';      }  };    class SellVisitor : public Visitor  {    public:      /\*virtual\*/ void visit(Apple \*a)      {            --m\_num\_apple;          a->sell();          cout << "m\_num\_apple " << m\_num\_apple << endl;      }      /\*virtual\*/ void visit(Google \*g)      {          --m\_num\_google;          g->sell();          cout << "m\_num\_google " << m\_num\_google << endl;      }  };    void Apple::accept(Visitor \*v)  {      v->visit(this);  }    void Google::accept(Visitor \*v)  {      v->visit(this);  }    int main()  {      Stock \*set[] = { new Apple, new Google, new Google,                       new Apple, new Apple, 0 };        BuyVisitor buy\_operation;      SellVisitor sell\_operation;      for (int i = 0; set[i]; i++)      {          set[i]->accept(&buy\_operation);      }      buy\_operation.total\_stocks();        for (int i = 0; set[i]; i++)      {            set[i]->accept(&sell\_operation);      }      sell\_operation.total\_stocks();  } |

**Output:**

Apple::buy

m\_num\_apple 1

Google::buy

m\_num\_google 1

Google::buy

m\_num\_google 2

Apple::buy

m\_num\_apple 2

Apple::buy

m\_num\_apple 3

m\_num\_apple 3, m\_num\_google 2

Apple::sell

m\_num\_apple 2

Google::sell

m\_num\_google 1

Google::sell

m\_num\_google 0

Apple::sell

m\_num\_apple 1

Apple::sell

m\_num\_apple 0

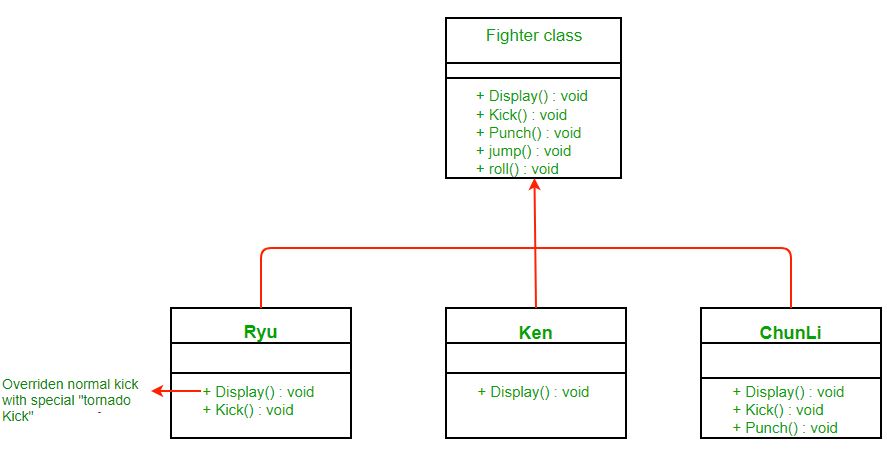
m\_num\_apple 0, m\_num\_google 0

# Strategy Pattern

## 8.1 Set 1 (Introduction)

As always we will learn this pattern by defining a problem and using strategy pattern to solve it. Suppose we are building a game “Street Fighter”. For simplicity assume that a character may have four moves that is kick, punch, roll and jump. Every character has kick and punch moves, but roll and jump are optional. How would you model your classes? Suppose initially you use inheritance and abstract out the common features in a **Fighter** class and let other characters subclass **Fighter** class.

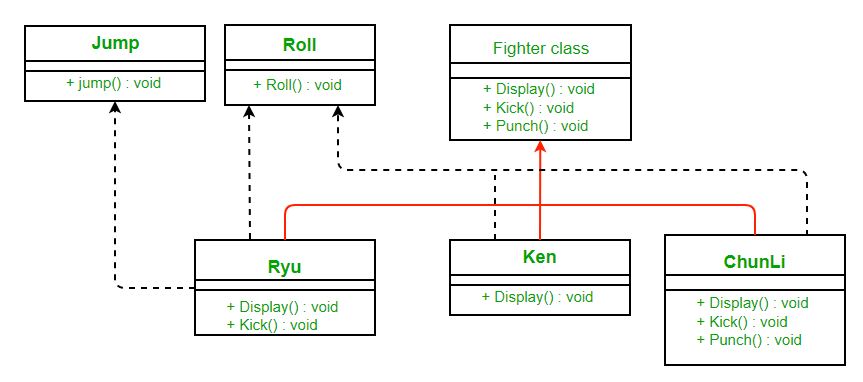
**Fighter** class will we have default implementation of normal actions. Any character with specialized move can override that action in its subclass. Class diagram would be as follows:



**What are the problems with above design?**

What if a character doesn’t perform jump move? It still inherits the jump behavior from superclass. Although you can override jump to do nothing in that case but you may have to do so for many existing classes and take care of that for future classes too. This would also make maintenance difficult. So we can’t use inheritance here.

**What about an Interface?**

Take a look at the following design:

It’s much cleaner. We took out some actions (which some characters might not perform) out of **Fighter**class and made interfaces for them. That way only characters that are supposed to jump will implement the **JumpBehavior.**

**What are the problems with above design?**

The main problem with the above design is code reuse. Since there is no default implementation of jump and roll behavior we may have code duplicity. You may have to rewrite the same jump behavior over and over in many subclasses.

**How can we avoid this?**

What if we made **JumpBehavior** and **RollBehavior**classes instead of interface? Well then we would have to use multiple inheritance that is not supported in many languages due to many problems associated with it.

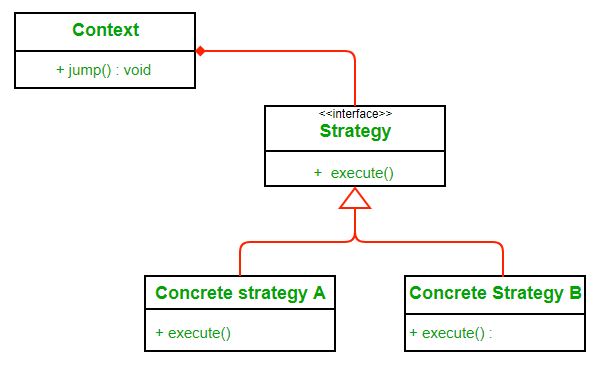
*Here strategy pattern comes to our rescue. We will learn what the strategy pattern is and then apply it to solve our problem.*

**Definition:**

Wikipedia defines strategy pattern as:

*“In computer programming, the****strategy pattern****(also known as the****policy pattern****) is a software design pattern that enables an algorithm’s behavior to be selected at runtime. The strategy pattern*

* *defines a family of algorithms,*
* *encapsulates each algorithm, and*
* *makes the algorithms interchangeable within that family.”*

***Class* Diagram:**

Here we rely on composition instead of inheritance for reuse. **Context**is composed of a **Strategy**. Instead of implementing a behavior the **Context** delegates it to **Strategy**. The context would be the class that would require changing behaviors. We can change behavior dynamically. **Strategy** is implemented as interface so that we can change behavior without affecting our context.

We will have a clearer understanding  of strategy pattern when we will use it to solve our problem.

**Advantages:**

1. A family of algorithms can be defined as a class hierarchy and can be used interchangeably to alter application behavior without changing its architecture.
2. By encapsulating the algorithm separately, new algorithms complying with the same interface can be easily introduced.
3. The application can switch strategies at run-time.
4. Strategy enables the clients to choose the required algorithm, without using a “switch” statement or a series of “if-else” statements.
5. Data structures used for implementing the algorithm are completely encapsulated in Strategy classes. Therefore, the implementation of an algorithm can be changed without affecting the Context class.

**Disadvantages:**

1. The application must be aware of all the strategies to select the right one for the right situation.
2. Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviours, which some concrete Strategy classes might not implement.
3. In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one.

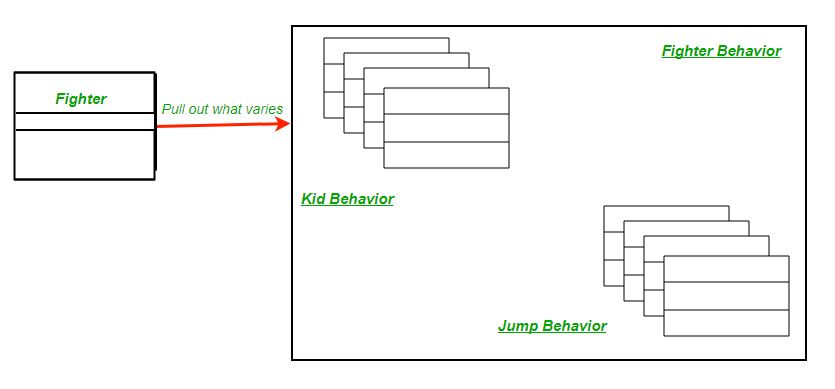
## 8.2 Strategy Pattern | Set 2 (Implementation)

We have discussed a fighter example and introduced Strategy Pattern in set 1.

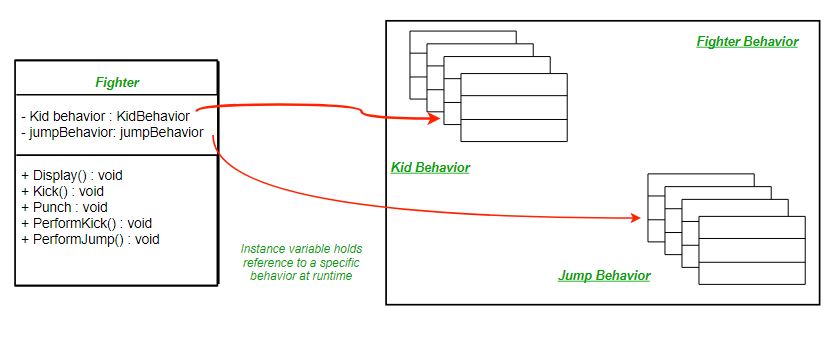
[Strategy Pattern | Set 1 (Introduction)](https://www.geeksforgeeks.org/strategy-pattern-set-1/)

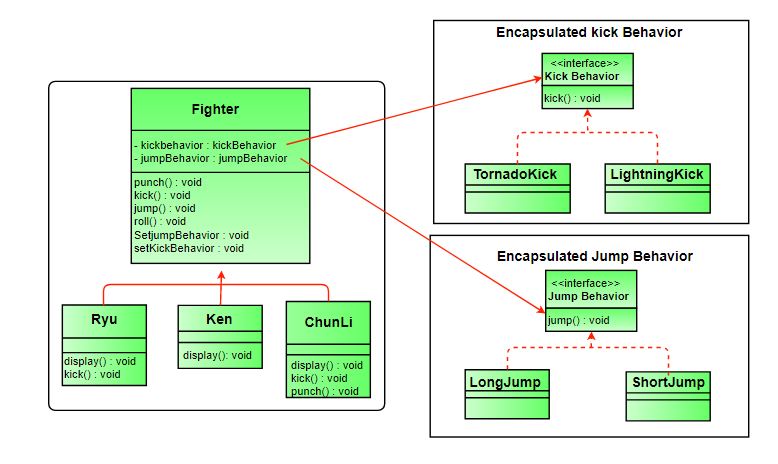
In this post, we apply Strategy Pattern to the Fighter Problem and discuss implementation.

The first step is to identify the behaviors that may vary across different classes in future and separate them from the rest. For our example let them be kick and jump behaviors. To separate these behaviors we will pull both methods out of **Fighter** class and create a new set of classes to represent each behavior.



The Fighter class will now delegate its kick and jump behavior instead of using kick and jump methods defined in the Fighter class or its subclass.



After reworking the final class diagram would be (Click on image for better view):  


Comparing our design to the definition of strategy pattern encapsulated kick and jump behaviors are two families of algorithms. And these algorithms are interchangeable as evident in implementation.

Below is the Java implementation of the same.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java program to demonstrate implementation of  // Strategy Pattern    // Abstract as you must have a specific fighter  abstract class Fighter  {      KickBehavior kickBehavior;      JumpBehavior jumpBehavior;        public Fighter(KickBehavior kickBehavior,                     JumpBehavior jumpBehavior)      {          this.jumpBehavior = jumpBehavior;          this.kickBehavior = kickBehavior;      }      public void punch()      {          System.out.println("Default Punch");      }      public void kick()      {          // delegate to kick behavior          kickBehavior.kick();      }      public void jump()      {            // delegate to jump behavior          jumpBehavior.jump();      }      public void roll()      {          System.out.println("Default Roll");      }      public void setKickBehavior(KickBehavior kickBehavior)      {          this.kickBehavior = kickBehavior;      }      public void setJumpBehavior(JumpBehavior jumpBehavior)      {          this.jumpBehavior = jumpBehavior;      }      public abstract void display();  }    // Encapsulated kick behaviors  interface KickBehavior  {      public void kick();  }  class LightningKick implements KickBehavior  {      public void kick()      {          System.out.println("Lightning Kick");      }  }  class TornadoKick implements KickBehavior  {      public void kick()      {          System.out.println("Tornado Kick");      }  }    // Encapsulated jump behaviors  interface JumpBehavior  {      public void jump();  }  class ShortJump implements JumpBehavior  {      public void jump()      {          System.out.println("Short Jump");      }  }  class LongJump implements JumpBehavior  {      public void jump()      {          System.out.println("Long Jump");      }  }    // Characters  class Ryu extends Fighter  {      public Ryu(KickBehavior kickBehavior,                 JumpBehavior jumpBehavior)      {          super(kickBehavior,jumpBehavior);      }      public void display()      {          System.out.println("Ryu");      }  }  class Ken extends Fighter  {      public Ken(KickBehavior kickBehavior,                 JumpBehavior jumpBehavior)      {          super(kickBehavior,jumpBehavior);      }      public void display()      {          System.out.println("Ken");      }  }  class ChunLi extends Fighter  {      public ChunLi(KickBehavior kickBehavior,                    JumpBehavior jumpBehavior)      {          super(kickBehavior,jumpBehavior);      }      public void display()      {          System.out.println("ChunLi");      }  }    // Driver class  class StreetFighter  {      public static void main(String args[])      {          // let us make some behaviors first          JumpBehavior shortJump = new ShortJump();          JumpBehavior LongJump = new LongJump();          KickBehavior tornadoKick = new TornadoKick();            // Make a fighter with desired behaviors          Fighter ken = new Ken(tornadoKick,shortJump);          ken.display();            // Test behaviors          ken.punch();          ken.kick();          ken.jump();            // Change behavior dynamically (algorithms are          // interchangeable)          ken.setJumpBehavior(LongJump);          ken.jump();      }  } |

Output :

Ken

Default Punch

Tornado Kick

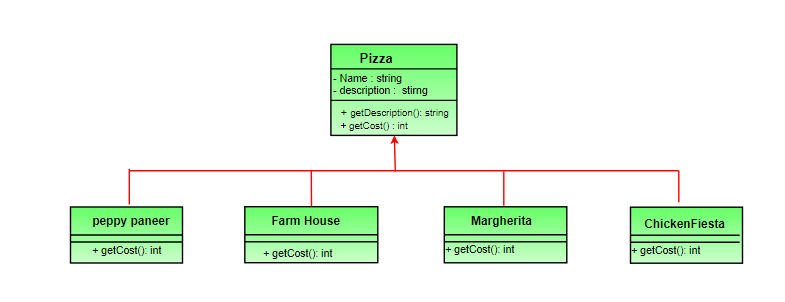
Short Jump

Long Jump

# Decorator Pattern

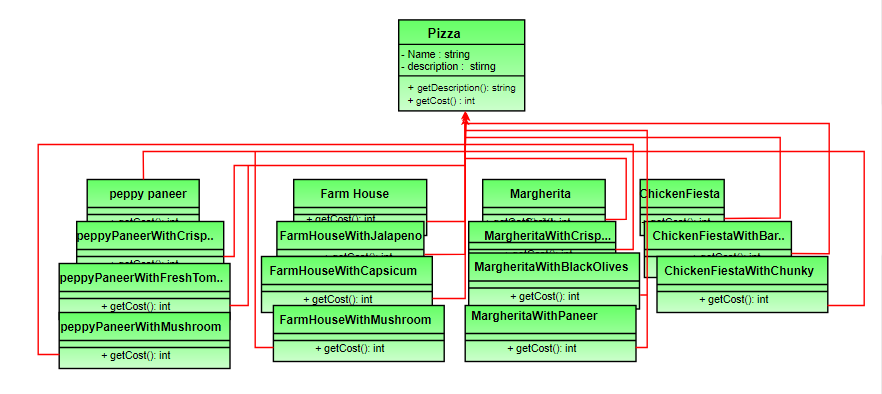
## 9.1 Set 1 (Background)

To understand decorator pattern let us consider a scenario inspired from the book “Head First Design Pattern”.  Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita  and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza**class.

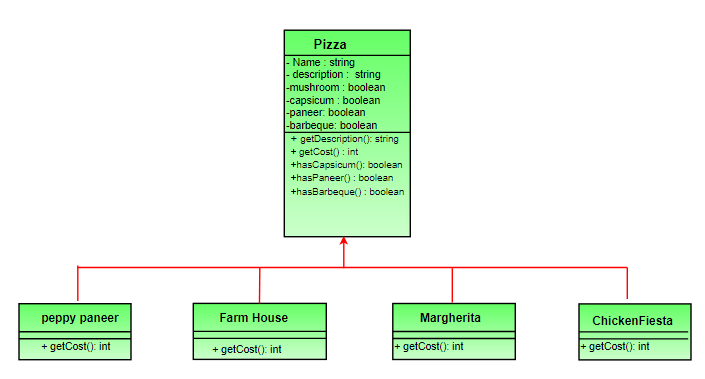
[](https://media.geeksforgeeks.org/wp-content/uploads/decorePattern-1.png)

Each pizza has different cost. We have overridden the getCost() in the subclasses to find the appropriate cost. Now suppose a new requirement, in addition to a pizza, customer can also ask for several toppings such as Fresh Tomato, Paneer, Jalapeno, Capsicum, Barbeque, etc. Let us think about for sometime that how do we accommodate changes in the above classes so that customer can choose pizza with toppings and we get the total cost of pizza and toppings the customer chooses.

Let us look at various options.

**Option 1**  
Create a new subclass for every topping with a pizza. The class diagram would look like:[](https://media.geeksforgeeks.org/wp-content/uploads/decorePattern.png)

This looks very complex. There are way too many classes and is a maintenance nightmare. Also if we want to add a new topping or pizza we have to add so many classes. This is obviously very bad design.

**Option 2:**  
Let’s add instance variables to pizza base class to represent whether or not each pizza has a topping. The class diagram would look like:

The getCost() of superclass calculates the costs for all the toppings while the one in the subclass adds the cost of that specific pizza.

// Sample getCost() in super class

public int getCost()

{

int totalToppingsCost = 0;

if (hasJalapeno() )

totalToppingsCost += jalapenoCost;

if (hasCapsicum() )

totalToppingsCost += capsicumCost;

// similarly for other toppings

return totalToppingsCost;

}

// Sample getCost() in subclass

public int getCost()

{

// 100 for Margherita and super.getCost()

// for toppings.

return super.getCost() + 100;

}

This design looks good at first but lets take a look at the problems associated with it.

* Price changes in toppings will lead to alteration in the existing code.
* New toppings will force us to add new methods and alter getCost() method in superclass.
* For some pizzas, some toppings may not be appropriate yet the subclass inherits them.
* What if customer wants double capsicum or double cheeseburst?

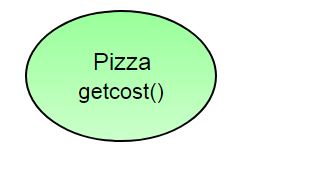
In short our design violates one of the most popular design principle – [**The Open-Closed Principle**](https://en.wikipedia.org/wiki/Open/closed_principle) which states that classes should be open for extension and closed for modification.

## 9.2 Set 2 (Introduction and Design)

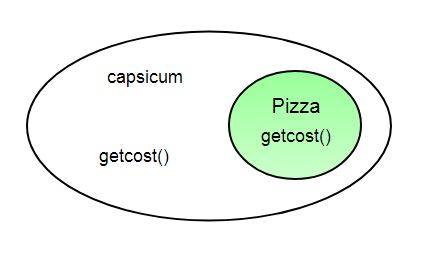
As we saw our [previous designs](https://www.geeksforgeeks.org/decorator-pattern/) using inheritance didn’t work out that well. In this article, decorator pattern is discussed for the design problem in previous set.

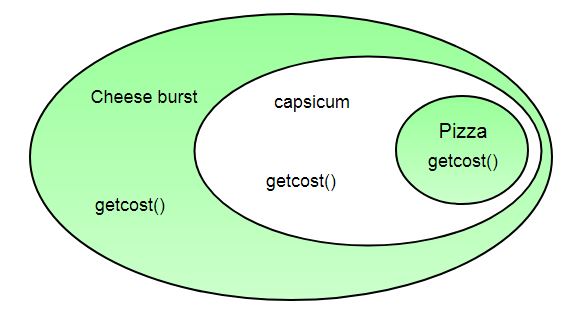
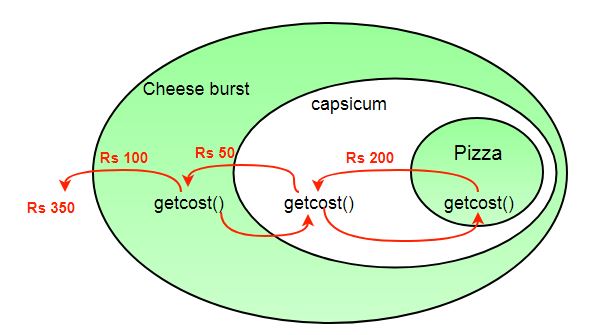
So what we do now is take a pizza and “decorate” it with toppings at runtime:

1. Take a pizza object.



1. “Decorate” it with a Capsicum object.



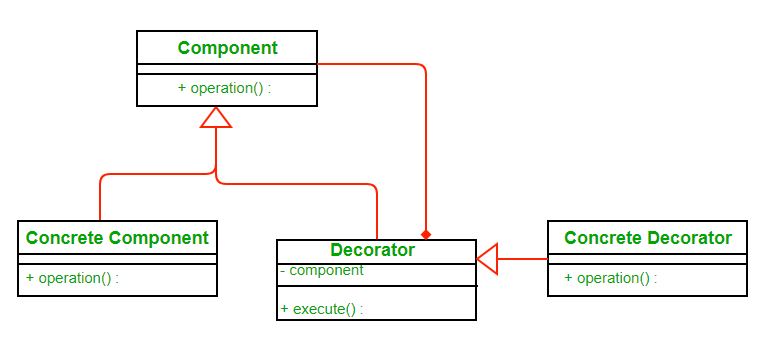
1. “Decorate” it with a CheeseBurst object.
2. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.  
   

What we get in the end is a pizza with cheeseburst and capsicum toppings. Visualize the “decorator” objects  like wrappers. Here are some of the properties of decorators:

* Decorators have the same super type as the object they decorate.
* You can use multiple decorators to wrap an object.
* Since decorators have same type as object, we can pass around decorated object instead of original.
* We can decorate objects at runtime.

**Definition:**

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

**Class Diagram:**Image src: [Wikipedia](https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Decorator_UML_class_diagram.svg/600px-Decorator_UML_class_diagram.svg.png)

* Each component can be used on its own or may be wrapped by a decorator.
* Each decorator has an instance variable that holds the reference to component it decorates(HAS-A relationship).
* The ConcreteComponent is the object we are going to dynamically decorate.

**Advantages:**

* The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime**.**
* The decorator pattern is an alternative to subclassing. Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects.
* Decorator offers a pay-as-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.

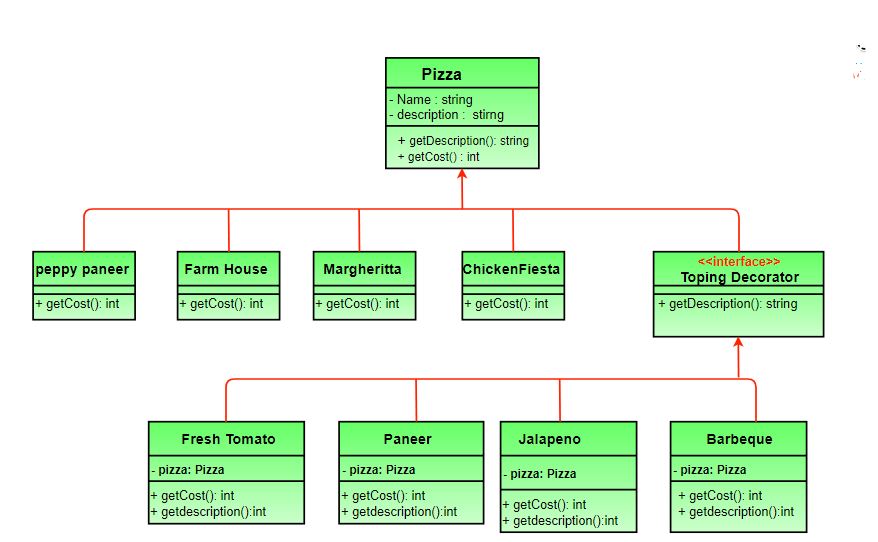
**Disadvantages:**

* Decorators can complicate the process of instantiating the component because you not only have to instantiate the component, but wrap it in a number of decorators.
* It can be complicated to have decorators keep track of other decorators, because to look back into multiple layers of the decorator chain starts to push the decorator pattern beyond its true intent.

## 9.3 Set 3 (Coding the Design)

We have discussed [Pizza design problem and different naive approaches to solve it](https://www.geeksforgeeks.org/decorator-pattern/) in set 1. We have also introduced [Decorator pattern in Set 2](https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/).

In this article, design and implementation of decorator pattern for Pizza problem is discussed. It is highly recommended that you try it yourself first.

The new class diagram (Click on the picture to see it clearly)  


* **Pizza**acts as our abstract component class.
* There are four concrete components namely **PeppyPaneer** , **FarmHouse**, **Margherita**, **ChickenFiesta**.
* **ToppingsDecorator**is our abstract decorator and **FreshTomato** , **Paneer**, **Jalapeno**, **Barbeque** are concrete decorators.

Below is the java implementation of above design.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java program to demonstrate Decorator  // pattern    // Abstract Pizza class (All classes extend  // from this)  abstract class Pizza  {      // it is an abstract pizza      String description = "Unkknown Pizza";        public String getDescription()      {          return description;      }        public abstract int getCost();  }    // The decorator class :  It extends Pizza to be  // interchangable with it topings decorator can  // also be implemented as an interface  abstract class ToppingsDecorator extends Pizza  {      public abstract String getDescription();  }    // Concrete pizza classes  class PeppyPaneer extends Pizza  {      public PeppyPaneer() { description = "PeppyPaneer"; }      public int getCost() {  return 100; }  }  class FarmHouse extends Pizza  {      public FarmHouse() {  description = "FarmHouse"; }      public int getCost() { return 200; }  }  class Margherita extends Pizza  {      public Margherita()  { description = "Margherita"; }      public int getCost() { return 100;  }  }  class ChickenFiesta extends Pizza  {      public ChickenFiesta() { description = "ChickenFiesta";}      public int getCost() { return 200; }  }  class SimplePizza extends Pizza  {  public SimplePizza() { description = "SimplePizza"; }  public int getCost() {  return 50;  }  }    // Concrete toppings classes  class FreshTomato extends ToppingsDecorator  {      // we need a reference to obj we are decorating      Pizza pizza;        public FreshTomato(Pizza pizza) { this.pizza = pizza; }      public String getDescription() {          return pizza.getDescription() + ", Fresh Tomato ";      }      public int getCost() { return 40 + pizza.getCost(); }  }  class Barbeque extends ToppingsDecorator  {      Pizza pizza;      public Barbeque(Pizza pizza) {  this.pizza = pizza;  }      public String getDescription() {          return pizza.getDescription() + ", Barbeque ";      }      public int getCost() { return 90 + pizza.getCost(); }  }  class Paneer extends ToppingsDecorator  {      Pizza pizza;      public Paneer(Pizza pizza)  {  this.pizza = pizza; }      public String getDescription() {          return pizza.getDescription() + ", Paneer ";      }      public int getCost()  {  return 70 + pizza.getCost(); }  }    // Other toppings can be coded in a similar way    // Driver class and method  class PizzaStore  {      public static void main(String args[])      {          // create new margherita pizza          Pizza pizza = new Margherita();          System.out.println( pizza.getDescription() +                           " Cost :" + pizza.getCost());            // create new FarmHouse pizza          Pizza pizza2 = new FarmHouse();            // decorate it with freshtomato topping          pizza2 = new FreshTomato(pizza2);            //decorate it with paneer topping          pizza2 = new Paneer(pizza2);            System.out.println( pizza2.getDescription() +                           " Cost :" + pizza2.getCost());          Pizza pizza3 = new Barbeque(null);    //no specific pizza          System.out.println( pizza3.getDescription() + "  Cost :" + pizza3.getCost());     }  } |

Output:

Margherita Cost :100

FarmHouse, Fresh Tomato , Paneer Cost :310

Notice how we can add/remove new pizzas and toppings with no alteration in previously tested code and toppings and pizzas are decoupled.

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // CPP program to demonstrate  // Decorator pattern  #include <iostream>  #include <string>  using namespace std;    // Component  class MilkShake  {  public:      virtual string Serve() = 0;      virtual float price() = 0;  };      // Concrete Component  class BaseMilkShake : public MilkShake  {  public:      string Serve()      {          return "MilkShake";      }        float price()      {          return 30;      }  };    // Decorator  class MilkShakeDecorator: public MilkShake  {  protected:      MilkShake \*m\_MilkShake;  public:        MilkShakeDecorator(MilkShake \*baseMilkShake): m\_MilkShake(baseMilkShake){}        string Serve()      {          return m\_MilkShake->Serve();      }        float price()      {          return m\_MilkShake->price();      }  };      // Concrete Decorator  class MangoMilkShake: public MilkShakeDecorator  {  public:      MangoMilkShake(MilkShake \*baseMilkShake): MilkShakeDecorator(baseMilkShake){}        string Serve()      {          return m\_MilkShake->Serve() + " decorated with Mango ";      }      float price()      {          return m\_MilkShake->price() + 40;      }  };      class VanillaMilkShake: public MilkShakeDecorator  {  public:      VanillaMilkShake(MilkShake \*baseMilkShake): MilkShakeDecorator(baseMilkShake){}        string Serve()      {          return m\_MilkShake->Serve() + " decorated with Vanilla ";      }      float price()      {          return m\_MilkShake->price() + 80;      }  };    int main()  {    MilkShake \*baseMilkShake = new BaseMilkShake();    cout << "Basic Milk shake \n";    cout << baseMilkShake -> Serve() << endl;    cout << baseMilkShake -> price() << endl;      MilkShake \*decoratedMilkShake = new MangoMilkShake(baseMilkShake);    cout << "Mango decorated Milk shake \n";    cout << decoratedMilkShake -> Serve() << endl;    cout << decoratedMilkShake -> price() << endl;      delete decoratedMilkShake;      decoratedMilkShake = new VanillaMilkShake(baseMilkShake);    cout << "Vanilla decorated Milk shake \n";    cout << decoratedMilkShake -> Serve() << endl;    cout << decoratedMilkShake -> price() << endl;     delete decoratedMilkShake;   delete baseMilkShake;    return 0;  } |

Output:

Basic Milk shake

MilkShake

Price of MilkShake : 30

Mango decorated Milk shake

MilkShake decorated with Mango

Price of Mango MilkShake : 70

Vanilla decorated Milk shake

MilkShake decorated with Vanilla

Price of Vanilla MilkShake : 110

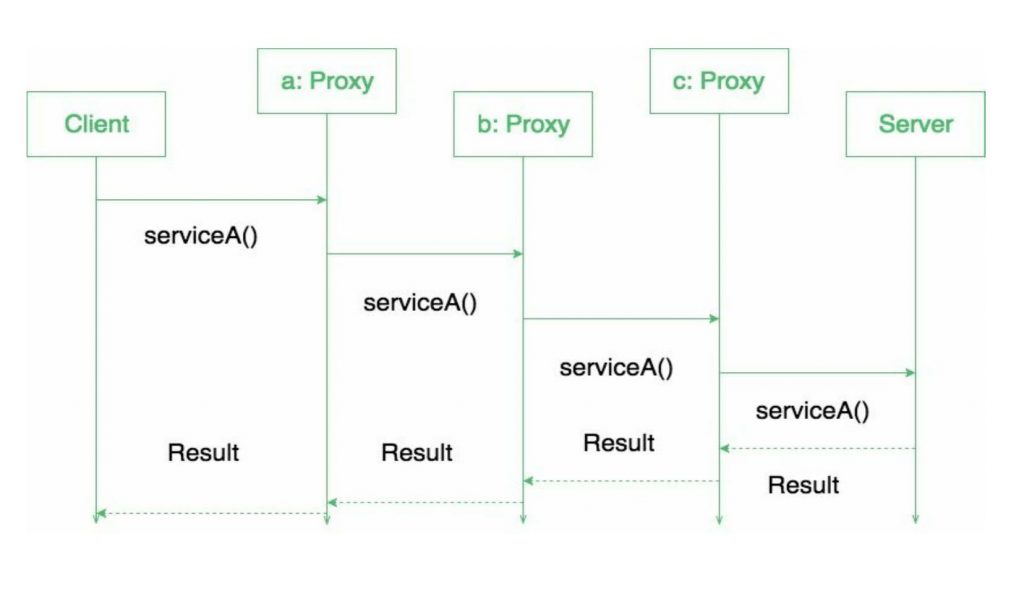
# Proxy Design Pattern

Proxy means ‘in place of’, representing’ or ‘in place of’ or ‘on behalf of’ are literal meanings of proxy and that directly explains **Proxy Design Pattern**.  
Proxies are also called surrogates, handles, and wrappers. They are closely related in structure, but not purpose, to [Adapters](https://www.geeksforgeeks.org/adapter-pattern/) and [Decorators](https://www.geeksforgeeks.org/the-decorator-pattern-set-2-introduction-and-design/).

A real world example can be a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash, and provides a means of accessing that cash when required. And that’s exactly what the Proxy pattern does – “**Controls and manage access to the object they are protecting**“.

**Behavior**

As in the decorator pattern, proxies can be chained together. The client, and each proxy, believes it is delegating messages to the real server:

[](https://media.geeksforgeeks.org/wp-content/uploads/19702635_1413820232028220_8628994446322905016_o.jpg)

**When to use this pattern?**

Proxy pattern is used when we need to create a wrapper to cover the main object’s complexity from the client.

**Types of proxies**

**Remote proxy:**  
They are responsible for representing the object located remotely. Talking to the real object might involve marshalling and unmarshalling of data and talking to the remote object. All that logic is encapsulated in these proxies and the client application need not worry about them.

**Virtual proxy:**

These proxies will provide some default and instant results if the real object is supposed to take some time to produce results. These proxies initiate the operation on real objects and provide a default result to the application. Once the real object is done, these proxies push the actual data to the client where it has provided dummy data earlier.

**Protection proxy:**

If an application does not have access to some resource then such proxies will talk to the objects in applications that have access to that resource and then get the result back.

**Smart Proxy:**

A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. An example can be to check if the real object is locked before it is accessed to ensure that no other object can change it.

**Some Examples**

A very simple real life scenario is our college internet, which restricts few site access. The proxy first checks the host you are connecting to, if it is not part of restricted site list, then it connects to the real internet. This example is based on Protection proxies.

Lets see how it works :

**Interface of Internet**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package com.saket.demo.proxy;    public interface Internet  {      public void connectTo(String serverhost) throws Exception;  } |

**RealInternet.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package com.saket.demo.proxy;    public class RealInternet implements Internet  {      @Override      public void connectTo(String serverhost)      {          System.out.println("Connecting to "+ serverhost);      }  } |

**ProxyInternet.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package com.saket.demo.proxy;    import java.util.ArrayList;  import java.util.List;      public class ProxyInternet implements Internet  {      private Internet internet = new RealInternet();      private static List<String> bannedSites;        static      {          bannedSites = new ArrayList<String>();          bannedSites.add("abc.com");          bannedSites.add("def.com");          bannedSites.add("ijk.com");          bannedSites.add("lnm.com");      }        @Override      public void connectTo(String serverhost) throws Exception      {          if(bannedSites.contains(serverhost.toLowerCase()))          {              throw new Exception("Access Denied");          }            internet.connectTo(serverhost);      }    } |

**Client.java**

*filter\_none*

*brightness\_4*

|  |
| --- |
| package com.saket.demo.proxy;    public class Client  {      public static void main (String[] args)      {          Internet internet = new ProxyInternet();          try          {              internet.connectTo("geeksforgeeks.org");              internet.connectTo("abc.com");          }          catch (Exception e)          {              System.out.println(e.getMessage());          }      }  } |

As one of the site is mentioned in the banned sites, So  
Running the program will give the output :

Connecting to geeksforgeeks.org

Access Denied

**Benefits:**

* One of the advantages of Proxy pattern is security.
* This pattern avoids duplication of objects which might be huge size and memory intensive. This in turn increases the performance of the application.
* The remote proxy also ensures about security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.

**Drawbacks/Consequences:**

This pattern introduces another layer of abstraction which sometimes may be an issue if the RealSubject code is accessed by some of the clients directly and some of them might access the Proxy classes. This might cause disparate behaviour.

**Interesting points:**

* There are few differences between the related patterns. Like Adapter pattern gives a different interface to its subject, while Proxy patterns provides the same interface from the original object but the decorator provides an enhanced interface. Decorator pattern adds additional behaviour at runtime.
* Proxy used in Java API: java.rmi.\*;

# Bridge Design Pattern

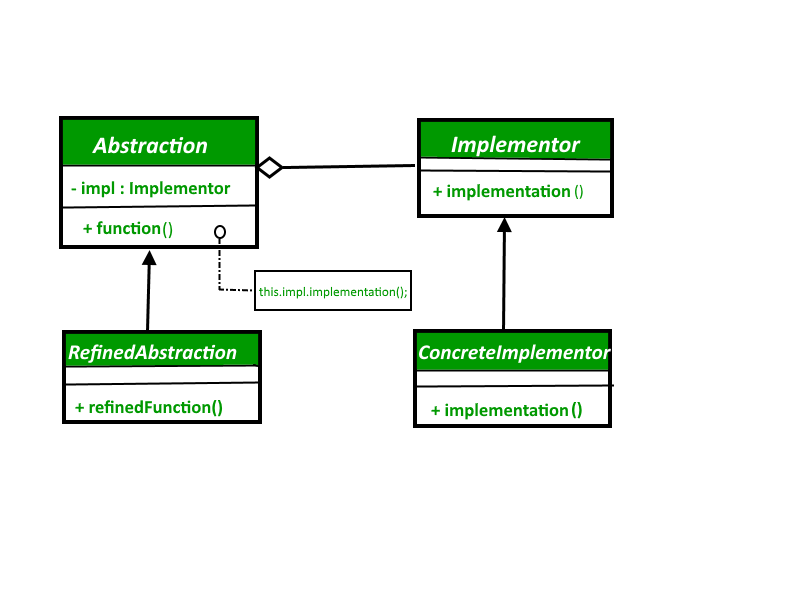
The Bridge design pattern allows you to separate the abstraction from the implementation.It is a structural design pattern.  
**There are 2 parts in Bridge design pattern :**

1. Abstraction
2. Implementation

This is a design mechanism that encapsulates an implementation class inside of an interface class.

* The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
* The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.
* The abstraction contains a reference to the implementor. Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors. Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction’s implementor at run-time. Changes to the implementor do not affect client code.
* It increases the loose coupling between class abstraction and it’s implementation.

**UML Diagram of Bridge Design Pattern**



**Elements of Bridge Design Pattern**

* **Abstraction** – core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
* **Refined Abstraction** – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implemetors.
* **Implementer** – It defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementer interface.
* **Concrete Implementation** – Implements the above implementer by providing concrete implementation.

**Lets see an Example of Bridge Design Pattern :**

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // Java code to demonstrate  // bridge design pattern    // abstraction in bridge pattern  abstract class Vehicle {      protected Workshop workShop1;      protected Workshop workShop2;        protected Vehicle(Workshop workShop1, Workshop workShop2)      {          this.workShop1 = workShop1;          this.workShop2 = workShop2;      }        abstract public void manufacture();  }    // Refine abstraction 1 in bridge pattern  class Car extends Vehicle {      public Car(Workshop workShop1, Workshop workShop2)      {          super(workShop1, workShop2);      }        @Override      public void manufacture()      {          System.out.print("Car ");          workShop1.work();          workShop2.work();      }  }    // Refine abstraction 2 in bridge pattern  class Bike extends Vehicle {      public Bike(Workshop workShop1, Workshop workShop2)      {          super(workShop1, workShop2);      }        @Override      public void manufacture()      {          System.out.print("Bike ");          workShop1.work();          workShop2.work();      }  }    // Implementor for bridge pattern  interface Workshop  {      abstract public void work();  }    // Concrete implementation 1 for bridge pattern  class Produce implements Workshop {      @Override      public void work()      {          System.out.print("Produced");      }  }    // Concrete implementation 2 for bridge pattern  class Assemble implements Workshop {      @Override      public void work()      {          System.out.print(" And");          System.out.println(" Assembled.");      }  }    // Demonstration of bridge design pattern  class BridgePattern {      public static void main(String[] args)      {          Vehicle vehicle1 = new Car(new Produce(), new Assemble());          vehicle1.manufacture();          Vehicle vehicle2 = new Bike(new Produce(), new Assemble());          vehicle2.manufacture();      }  } |

Output :

Car Produced And Assembled.

Bike Produced And Assembled.

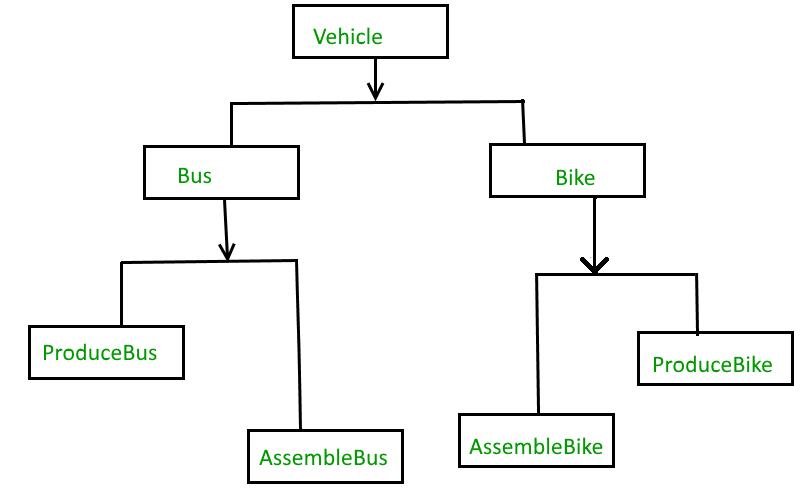
Here we’re producing and assembling the two different vehicles using Bridge design pattern.

**When we need bridge design pattern**

The Bridge pattern is an application of the old advice, “prefer composition over inheritance”. It becomes handy when you must subclass different times in ways that are orthogonal with one another.

For Example, the above example can also be done something like this :

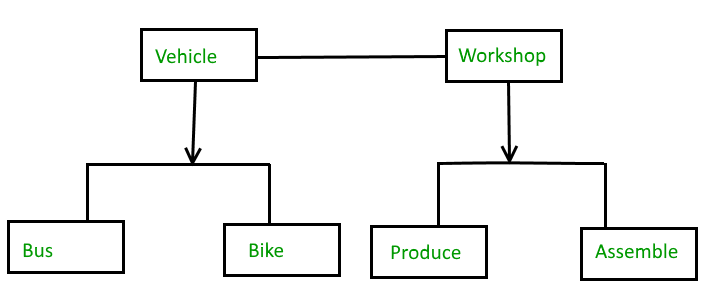
**Without Bridge Design Pattern**



But the above solution has a problem. If you want to change the Bus class, then you may end up changing ProduceBus and AssembleBus as well and if the change is workshop specific then you may need to change the Bike class as well.

**With Bridge Design Pattern**

You can solve the above problem by decoupling the Vehicle and Workshop interfaces in the below manner.



**Advantages**

1. Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
2. It is used mainly for implementing platform independence feature.
3. It adds one more method level redirection to achieve the objective.
4. Publish abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
5. Use bridge pattern to run-time binding of the implementation.
6. Use bridge pattern to map orthogonal class hierarchies
7. Bridge is designed up-front to let the abstraction and the implementation vary independently.

# Prototype Design Pattern

Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process.

The prototype pattern is a creational design pattern. Prototype patterns is required, when object creation is time consuming, and costly operation, so we create object with existing object itself. One of the best available way to create object from existing objects are **clone() method**. Clone is the simplest approach to implement prototype pattern. However, it is your call to decide how to copy existing object based on your business model.

**Prototype Design Participants**

1) **Prototype** : This is the prototype of actual object.

2) **Prototype registry** : This is used as registry service to have all prototypes accessible using simple string parameters.

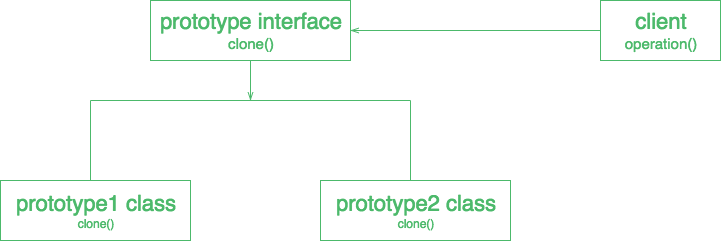
3) **Client** : Client will be responsible for using registry service to access prototype instances.

**When to use the Prototype Design Pattern**

When a system should be independent of how its products are created, composed, and represented and  
When the classes to instantiate are specified at run-time.  
For example,  
1) By dynamic loading or To avoid building a class hierarchy of factories that parallels the class hierarchy of products or

2) When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

**The UML Diagram of the Prototype Design Pattern**

[](https://media.geeksforgeeks.org/wp-content/uploads/download-1.png)

*filter\_none*

*edit*

*play\_arrow*

*brightness\_4*

|  |
| --- |
| // A Java program to demonstrate working of  // Prototype Design Pattern with example  // of a ColorStore class to store existing objects.    import java.util.HashMap;  import java.util.Map;      abstract class Color implements Cloneable  {        protected String colorName;        abstract void addColor();        public Object clone()      {          Object clone = null;          try          {              clone = super.clone();          }          catch (CloneNotSupportedException e)          {              e.printStackTrace();          }          return clone;      }  }    class blueColor extends Color  {      public blueColor()      {          this.colorName = "blue";      }        @Override      void addColor()      {          System.out.println("Blue color added");      }    }    class blackColor extends Color{        public blackColor()      {          this.colorName = "black";      }        @Override      void addColor()      {          System.out.println("Black color added");      }  }    class ColorStore {        private static Map<String, Color> colorMap = new HashMap<String, Color>();        static      {          colorMap.put("blue", new blueColor());          colorMap.put("black", new blackColor());      }        public static Color getColor(String colorName)      {          return (Color) colorMap.get(colorName).clone();      }  }      // Driver class  class Prototype  {      public static void main (String[] args)      {          ColorStore.getColor("blue").addColor();          ColorStore.getColor("black").addColor();          ColorStore.getColor("black").addColor();          ColorStore.getColor("blue").addColor();      }  } |

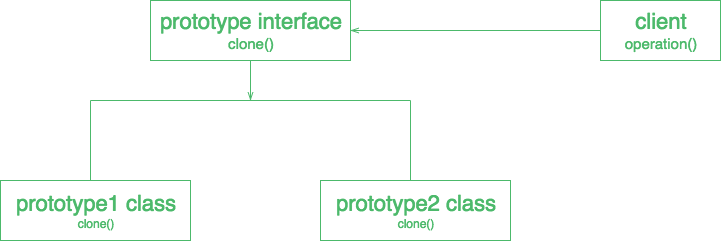
Output :

Blue color added

Black color added

Black color added

Blue color added

**UML diagram of example:**  
[](https://media.geeksforgeeks.org/wp-content/uploads/download-1.png)

**Advantages of Prototype Design Pattern**

* **Adding and removing products at run-time –** Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That’s a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
* **Specifying new objects by varying values –** Highly dynamic systems let you define new behavior through object composition by specifying values for an object’s variables and not by defining new classes.
* **Specifying new objects by varying structure –** Many applications build objects from parts and subparts. For convenience, such applications often let you instantiate complex, user-defined structures to use a specific subcircuit again and again.
* **Reduced subclassing –** Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don’t need a Creator class hierarchy at all.

**Disadvantages of Prototype Design Pattern**

* Overkill for a project that uses very few objects and/or does not have an underlying emphasis on the extension of prototype chains.
* It also hides concrete product classes from the client
* Each subclass of Prototype must implement the clone() operation which may be difficult, when the classes under consideration already exist. Also implementing clone() can be difficult when their internals include objects that don’t support copying or have circular references.