Design Archive

Flyweight Design Pattern

Flyweight pattern is one of the structural design patterns as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 10^5). One important feature of flyweight objects is that they are **immutable**. This means that they cannot be modified once they have been constructed.

Why do we care for number of objects in our program?

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.

Intrinsic and Extrinsic States

To understand Intrinsic and Extrinsic state, let us consider an example.

Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter 'B' is no different from another 'B' . If client again types a 'B' we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.

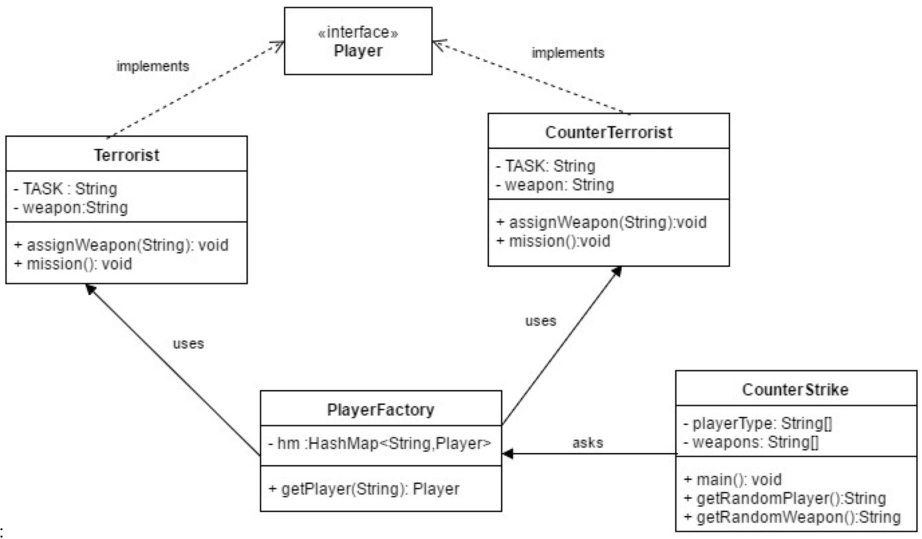
Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can't be shared among objects.

Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of Counter Strike. So we have 2 classes one for **T**errorist(**T**) and other for **C**ounter **T**errorist(**CT**). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist's task is to plant a bomb while the counter terrorists have to diffuse the bomb.

Why to use Flyweight Design Pattern in this example? Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.

Intrinsic State : Here 'task' is an intrinsic state for both types of players, since this is always same for T's/CT's. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.

Extrinsic State : Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.



Class Diagram :

```
// A Java program to demonstrate working of
// FlyWeight Pattern with example of Counter
// Strike Game
import java.util.Random;
import java.util.HashMap;

// A common interface for all players
interface Player
{
   public void assignWeapon(String weapon);
   public void mission();
}

// Terrorist must have weapon and mission
class Terrorist implements Player
```

```java
{
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private String weapon;

    public Terrorist()
    {
        TASK = "PLANT A BOMB";
    }
    public void assignWeapon(String weapon)
    {
        // Assign a weapon
        this.weapon = weapon;
    }
    public void mission()
    {
        //Work on the Mission
        System.out.println("Terrorist with weapon "
                           + weapon + "|" + " Task is " + TASK);
    }
}

// CounterTerrorist must have weapon and mission
class CounterTerrorist implements Player
{
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private String weapon;

    public CounterTerrorist()
    {
        TASK = "DIFFUSE BOMB";
    }
    public void assignWeapon(String weapon)
    {
        this.weapon = weapon;
    }
    public void mission()
    {
        System.out.println("Counter Terrorist with weapon "
                           + weapon + "|" + " Task is " + TASK);
    }
}

// Claass used to get a playeer using HashMap (Returns
// an existing player if a player of given type exists.
// Else creates a new player and returns it.
class PlayerFactory
{
    /* HashMap stores the reference to the object
       of Terrorist(TS) or CounterTerrorist(CT).  */
    private static HashMap <String, Player> hm =
                         new HashMap<String, Player>();

    // Method to get a player
    public static Player getPlayer(String type)
    {
        Player p = null;

        /* If an object for TS or CT has already been
           created simply return its reference */
        if (hm.containsKey(type))
                p = hm.get(type);
        else
        {
            /* create an object of TS/CT  */
            switch(type)
            {
            case "Terrorist":
                System.out.println("Terrorist Created");
                p = new Terrorist();
                break;
            case "CounterTerrorist":
                System.out.println("Counter Terrorist Created");
                p = new CounterTerrorist();
                break;
            default :
                System.out.println("Unreachable code!");
            }

            // Once created insert it into the HashMap
            hm.put(type, p);
        }
        return p;
    }
}

// Driver class
public class CounterStrike
{
    // All player types and weapons (used by getRandPlayerType()
    // and getRandWeapon()
    private static String[] playerType =
                {"Terrorist", "CounterTerrorist"};
    private static String[] weapons =
      {"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};


    // Driver code
    public static void main(String args[])
    {
        /* Assume that we have a total of 10 players
           in the game. */
        for (int i = 0; i < 10; i++)
        {
            /* getPlayer() is called simply using the class
               name since the method is a static one */
            Player p = PlayerFactory.getPlayer(getRandPlayerType());

            /* Assign a weapon chosen randomly uniformly
               from the weapon array  */
            p.assignWeapon(getRandWeapon());

            // Send this player on a mission
            p.mission();
        }
    }

    // Utility methods to get a random player type and
    // weapon
    public static String getRandPlayerType()
    {
        Random r = new Random();

        // Will return an integer between [0,2)
        int randInt = r.nextInt(playerType.length);

        // return the player stored at index 'randInt'
        return playerType[randInt];
    }
    public static String getRandWeapon()
    {
        Random r = new Random();

        // Will return an integer between [0,5)
        int randInt = r.nextInt(weapons.length);

        // Return the weapon stored at index 'randInt'
        return weapons[randInt];
    }
}
```

Output:

```
Counter Terrorist Created
Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
Terrorist Created
Terrorist with weapon AK-47| Task is PLANT A BOMB
Terrorist with weapon Gut Knife| Task is PLANT A BOMB
Terrorist with weapon Gut Knife| Task is PLANT A BOMB
Terrorist with weapon Desert Eagle| Task is PLANT A BOMB
Terrorist with weapon AK-47| Task is PLANT A BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
Counter Terrorist with weapon Gut Knife| Task is DIFFUSE BOMB
Counter Terrorist with weapon Desert Eagle| Task is DIFFUSE BOMB
```

References:

- Elements of Reusable Object-Oriented Software(By Gang Of Four)
- https://en.wikipedia.org/wiki/Flyweight_pattern

This article is contributed by **Chirag Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Curiously recurring template pattern (CRTP)

Background:

It is recommended to refer Virtual Functions and Runtime Polymorphism as a prerequisite of this. Below is an example program to demonstrate run time polymorphism.

```
// A simple C++ program to demonstrate run-time
// polymorphism
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
   Dimension(int _X, int _Y) {mX = _X;  mY = _Y; }
private:
   int mX, mY;
};

// Base class for all image types
class Image
{
public:
   virtual void Draw() = 0;
   virtual Dimension GetDimensionInPixels() = 0;
protected:
   int dimensionX;
   int dimensionY;
};

// For Tiff Images
class TiffImage : public Image
{
public:
   void Draw() { }
   Dimension GetDimensionInPixels() {
      return Dimension(dimensionX, dimensionY);
   }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc

// Driver code that calls virtual function
int main()
{
   // An image type
   Image* pImage = new TiffImage;

   // Store time before virtual function calls
   auto then = Clock::now();

   // Call Draw 1000 times to make sure performance
   // is visible
   for (int i = 0; i < 1000; ++i)
      pImage->Draw();

   // Store time after virtual function calls
   auto now = Clock::now();

   cout << "Time taken: "
        << std::chrono::duration_cast
          <std::chrono::nanoseconds>(now - then).count()
        << " nanoseconds" << endl;

   return 0;
}
```

Output :

```
Time taken: 2613 nanoseconds
```

See this for above result.

When a method is declared virtual, compiler secretly does two things for us:

1. Defines a VPtr in first 4 bytes of the class object
2. Inserts code in constructor to initialize VPtr to point to the VTable

What are VTable and VPtr?

When a method is declared virtual in a class, compiler creates a virtual table (aka VTable) and stores addresses of virtual methods in that table. A virtual pointer (aka VPtr) is then created and initialized to point to that VTable. A VTable is shared across all the instances of the class, i.e. compiler creates only one instance of VTable to be shared across all the objects of a class. Each instance of the class has its own version of VPtr. If we print the size of a class object containing at least one virtual method, the output will be sizeof(class data) + sizeof(VPtr).

Since address of virtual method is stored in VTable, VPtr can be manipulated to make calls to those virtual methods thereby violating principles of encapsulation. See below example:

```
// A C++ program to demonstrate that we can directly
// manipulate VPtr. Note that this program is based
// on the assumption that compiler store vPtr in a
// specific way to achieve run-time polymorphism.
#include <iostream>
using namespace std;

#pragma pack(1)

// A base class with virtual function foo()
class CBase
{
public:
    virtual void foo() noexcept {
        cout << "CBase::Foo() called" << endl;
    }
protected:
    int mData;
};

// A derived class with its own implementation
// of foo()
class CDerived : public CBase
{
public:
    void foo() noexcept {
        cout << "CDerived::Foo() called" << endl;
    }
private:
    char cChar;
};

// Driver code
int main()
{
    // A base type pointer pointing to derived
    CBase *pBase = new CDerived;

    // Accessing vPtr
    int* pVPtr = *(int**)pBase;

    // Calling virtual method
    ((void(*)())pVPtr[0])();

    // Changing vPtr
    delete pBase;
    pBase = new CBase;
    pVPtr = *(int**)pBase;

    // Calls method for new base object
    ((void(*)())pVPtr[0])();

    return 0;
}
```

Output :

```
CDerived::Foo() called
CBase::Foo() called
```

We are able to access vPtr and able to make calls to virtual methods through it. The memory representation of objects is explained here.

Is it wise to use virtual method?

As it can be seen, through base class pointer, call to derived class method is being dispatched. Everything seems to be working fine. Then what is the problem?

If a virtual routine is called many times (order of hundreds of thousands), it drops the performance of system, reason being each time the routine is called, its address needs to be resolved by looking through VTable using VPtr. Extra indirection (pointer dereference) for each call to a virtual method makes accessing VTable a costly operation and it is better to avoid it as much as we can.

Curiously Recurring Template Pattern (CRTP)

Usage of VPtr and VTable can be avoided altogether through Curiously Recurring Template Pattern (CRTP). CRTP is a design pattern in C++ in which a class X derives from a class template instantiation using X itself as template argument. More generally it is known as F-bound polymorphism.

```
// Image program (similar to above) to demonstrate
// working of CRTP
#include <iostream>
#include <chrono>
using namespace std;

typedef std::chrono::high_resolution_clock Clock;

// To store dimensions of an image
class Dimension
{
public:
    Dimension(int _X, int _Y)
    {
        mX = _X;
        mY = _Y;
    }
private:
    int mX, mY;
};

// Base class for all image types. The template
// parameter T is used to know type of derived
// class pointed by pointer.
template <class T>
class Image
{
public:
    void Draw()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->Draw();
    }
    Dimension GetDimensionInPixels()
    {
        // Dispatch call to exact type
        static_cast<T*> (this)->GetDimensionInPixels();
    }

protected:
    int dimensionX, dimensionY;
};


// For Tiff Images
class TiffImage : public Image<TiffImage>
{
public:
    void Draw()
    {
        // Uncomment this to check method dispatch
        // cout << "TiffImage::Draw() called" << endl;
    }
    Dimension GetDimensionInPixels()
    {
        return Dimension(dimensionX, dimensionY);
    }
};

// There can be more derived classes like PngImage,
// BitmapImage, etc

// Driver code
int main()
{
    // An Image type pointer pointing to Tiffimage
    Image<TiffImage>* pImage = new TiffImage;

    // Store time before virtual function calls
    auto then = Clock::now();

    // Call Draw 1000 times to make sure performance
    // is visible
    for (int i = 0; i < 1000; ++i)
        pImage->Draw();

    // Store time after virtual function calls
    auto now = Clock::now();

    cout << "Time taken: "
         << std::chrono::duration_cast
         <std::chrono::nanoseconds>(now - then).count()
         << " nanoseconds" << endl;

    return 0;
}
```

Output :

```
Time taken: 732 nanoseconds
```

See this for above result.

Virtual method vs CRTP benchmark

The time taken while using virtual method was 2613 nanoseconds. This (small) performance gain from CRTP is because the use of a VTable dispatch has been circumvented. Please note that the performance depends on a lot of factors like compiler used, operations performed by virtual methods. Performance numbers might differ in different runs, but (small) performance gain is expected from CRTP.
Note: If we print size of class in CRTP, it can bee seen that VPtr no longer reserves 4 bytes of memory.

```
cout )
Questions? Keep them coming. We would love to answer.


Reference(s)

https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern


This article is contributed by Aashish Barnwal. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforg


Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above
```

GATE CS Corner
Company Wise Coding Practice

Design

Iterator Pattern

Iterator Pattern is a relatively simple and frequently used design pattern. There are a lot of data structures/collections available in every language. Each collection must provide an iterator that lets it iterate through its objects. However while doing so it should make sure that it does not expose its implementation.
Suppose we are building an application that requires us to maintain a list of notifications. Eventually some part of your code will require to iterate over all notifications. If we implemented your collection of notifications as array you would iterate over them as:

```
// If a simple array is used to store notifications
for (int i = 0; i < notificationList.length; i++)
    Notification notification = notificationList[i]);
```

```
// If ArrayList is Java is used, then we would iterate
// over them as:
for (int i = 0; i < notificationList.size(); i++)
    Notification notification = (Notification)notificationList.get(i);
```

And if it were some other collection like set, tree etc. way of iterating would change slightly. Now what if we build an iterator that provides a generic way of iterating over a collection independent of its type.

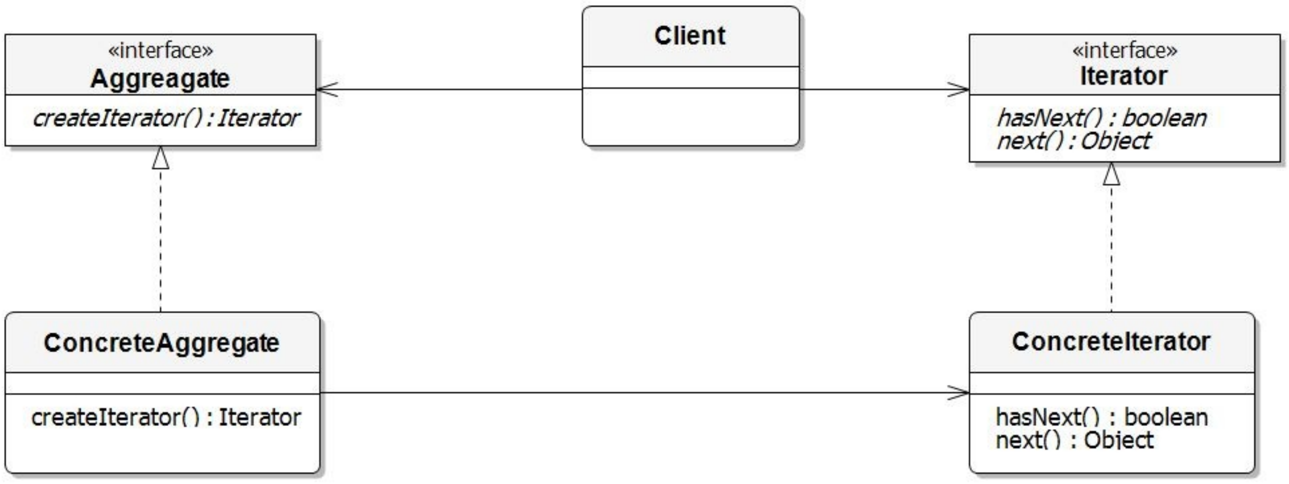
```
// Create an iterator
Iterator iterator = notificationList.createIterator();

// It wouldn't matter if list is Array or ArrayList or
// anything else.
while (iterator.hasNext())
{
    Notification notification = iterator.next());
```

```
}
```

Iterator pattern lets us do just that. Formally it is defined as below:

The iterator pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.

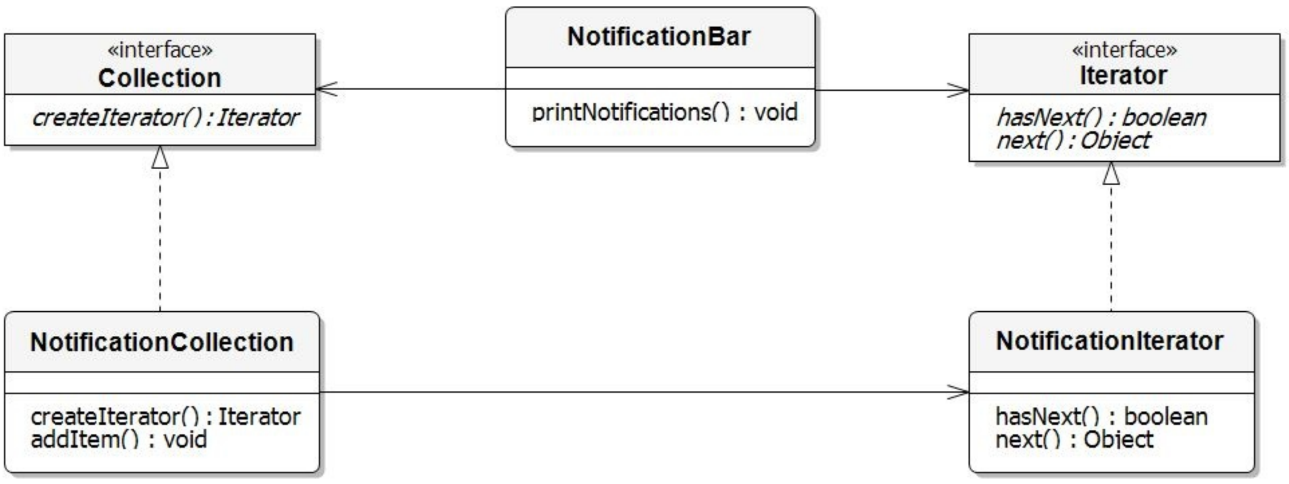
Class Diagram:



Here we have a common interface Aggregate for client as it decouples it from the implementation of your collection of objects. The ConcreteIterator implements createIterator() that returns iterator for its collection. Each ConcreteAggregate's responsibility is to instantiate a ConcreteIterator that can iterate over its collection of objects. The iterator interface provides a set of methods for traversing or modifying the collection that is in addition to next()/hasNext() it can also provide functions for search, remove etc.

Let's understand this through an example. Suppose we are creating a notification bar in our application that displays all the notifications which are held in a notification collection. NotificationCollection provides an iterator to iterate over its elements without exposing how it has implemented the collection (array in this case) to the Client (NotificationBar).

The class diagram would be:



Below is the Java implementation of the same:

```
// A Java program to demonstrate implementation
// of iterator pattern with the example of
// notifications

// A simple Notification class
class Notification
{
   // To store notification message
   String notification;

   public Notification(String notification)
   {
      this.notification = notification;
   }
   public String getNotification()
   {
      return notification;
```

```
    }
}

// Collection interface
interface Collection
{
    public Iterator createIterator();
}

// Collection of notifications
class NotificationCollection implements Collection
{
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    Notification[] notificationList;

    public NotificationCollection()
    {
        notificationList = new Notification[MAX_ITEMS];

        // Let us add some dummy notifications
        addItem("Notification 1");
        addItem("Notification 2");
        addItem("Notification 3");
    }

    public void addItem(String str)
    {
        Notification notification = new Notification(str);
        if (numberOfItems >= MAX_ITEMS)
            System.err.println("Full");
        else
        {
            notificationList[numberOfItems] = notification;
            numberOfItems = numberOfItems + 1;
        }
    }

    public Iterator createIterator()
    {
        return new NotificationIterator(notificationList);
    }
}

// We could also use Java.Util.Iterator
interface Iterator
{
    // indicates whether there are more elements to
    // iterate over
    boolean hasNext();

    // returns the next element
    Object next();
}

// Notification iterator
class NotificationIterator implements Iterator
{
    Notification[] notificationList;

    // maintains curr pos of iterator over the array
    int pos = 0;

    // Constructor takes the array of notifiactionList are
    // going to iterate over.
    public  NotificationIterator (Notification[] notificationList)
    {
        this.notificationList = notificationList;
    }

    public Object next()
    {
        // return next element in the array and increment pos
        Notification notification =  notificationList[pos];
        pos += 1;
        return notification;
    }

    public boolean hasNext()
    {
        if (pos >= notificationList.length ||
            notificationList[pos] == null)
            return false;
        else
            return true;
    }
```

```
}

// Contains collection of notifications as an object of
// NotificationCollection
class NotificationBar
{
   NotificationCollection notifications;

   public NotificationBar(NotificationCollection notifications)
   {
      this.notifications = notifications;
   }

   public void printNotifications()
   {
      Iterator iterator = notifications.createIterator();
      System.out.println("-------NOTIFICATION BAR------------");
      while (iterator.hasNext())
      {
         Notification n = (Notification)iterator.next();
         System.out.println(n.getNotification());
      }
   }
}

// Driver class
class Main
{
   public static void main(String args[])
   {
      NotificationCollection nc = new NotificationCollection();
      NotificationBar nb = new NotificationBar(nc);
      nb.printNotifications();
   }
}
```

Output:

```
-------NOTIFICATION BAR------------
Notification 1
Notification 2
Notification 3
```

Notice that if we would have used ArrayList instead of Array there will not be any change in the client (notification bar) code due to the decoupling achieved by the use of iterator interface.

References:
Head First Design Patterns

This article is contributed by **Sulabh Kumar.**If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design

Opaque Pointer

What is an opaque pointer?
Opaque as the name suggests is something we can't see through. e.g. wood is opaque. Opaque pointer is a pointer which points to a DS whose contents are not exposed at the time of its definition.

Following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

Why Opaque pointer?
There are places where we just want to hint the compiler that "Hey! This is some DS which will be used by our clients. Don't worry, clients will provide its implementation while preparing compilation unit". Such type of design is robust when we deal with shared code. Please see below example:

Let's say we are working on an app to deal with images. Since we are living in a world where everything is moving to cloud and devices are very affordable to buy, we want to develop apps for windows, android and apple platforms. So, it would be nice to have a good design which is robust, scalable and flexible as per our requirements. We can have shared code which would be used by all platforms and then different end-point can have platform specific code.
To deal with images, we have a CImage class exposing APIs to deal with various image operations (scale, rotate, move, save etc).
Since all the platforms will be providing same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have different mechanism to access pixels of an image than Windows does. This means that APIs might demand different set of info to perform operations. So to work on shared code, this is what we would like to do:

Image.h : A header file to store class declaration.

```
// This class provides API to deal with various
// image operations. Different platforms can
// implement these operations in different ways.
class CImage
{
public:
   CImage();
   ~CImage();
   struct SImageInfo* pImageInfo;
   void Rotate(double angle);
   void Scale(double scaleFactorX,
           double scaleFactorY);
   void Move(int toX, int toY);
private:
   void InitImageInfo();
};
```

Image.cpp : Code that will be shared across different end-points

```
// Constructor and destructor for CImage
CImage::CImage()
{
   InitImageInfo();
}

CImage::~CImage()
{
   // Destroy stuffs here
}
```

Image_windows.cpp : Code specific to Windows will reside here

```
struct SImageInfo
{
  // Windows specific DataSet
};

void CImage::InitImageInfo()
{
   pImageInfo = new SImageInfo;
   // Initialize windows specific info here
}

void CImage::Rotate()
{
   // Make use of windows specific SImageInfo
}
```

Image_apple.cpp : Code specific to Apple will reside here

```
struct SImageInfo
{
   // Apple specific DataSet
};
void CImage::InitImageInfo()
{
   pImageInfo = new SImageInfo;

   // Initialize apple specific info here
}
void CImage::Rotate()
{
   // Make use of apple specific SImageInfo
}
```

As it can be seen from the above example, **while defining blueprint of the CImage class we are only mentioning that there is a SImageInfo DS. The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that DS and use it as per their requirement**. If in future we want to develop app for a new end-point 'X', the design is already there. We only need to define SImageInfo for end-point 'X' and use it accordingly.

Please note that the above explained example is one way of doing this. Design is all about discussion and requirement. A good design is decided taking many factors into account. We can also have platform specific classes like CImageWindows, CImageApple and put all platform specific code there.

Questions? Keep them coming. We would love to answer.

This article is contributed by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

C/C++ Puzzles
Design
cpp-pointer

Command Pattern

Like previous articles, let us take up a design problem to understand command pattern. Suppose you are building a home automation system. There is a programmable remote which can be used to turn on and off various items in your home like lights, stereo, AC etc. It looks something like this.

You can do it with simple if-else statements like

```
if (buttonPressed == button1)
   lights.on()
```

But we need to keep in mind that turning on some devices like stereo comprises of many steps like setting cd, volume etc. Also we can reassign a button to do something else. By using simple if-else we are coding to implementation rather than interface. Also there is tight coupling.

So what we want to achieve is a design that provides loose coupling and remote control should not have much information about a particular device. The command pattern helps us do that.

Definition: The **command pattern** encapsulates a request as an object, thereby letting us parameterize other objects with different requests, queue or log requests, and support undoable operations.

The definition is a bit confusing at first but let’s step through it. In analogy to our problem above remote control is the client and stereo, lights etc. are the receivers. In command pattern there is a Command object that *encapsulates a request* by binding together a set of actions on a specific receiver. It does so by exposing just one method execute() that causes some actions to be invoked on the receiver.

Parameterizing other objects with different requests in our analogy means that the button used to turn on the lights can later be used to turn on stereo or maybe open the garage door.

queue or log requests, and support undoable operations means that Command’s Execute operation can store state for reversing its effects in the Command itself. The Command may have an added unExecute operation that reverses the effects of a previous call to execute.It may also support logging changes so that they can be reapplied in case of a system crash.

Below is the Java implementation of above mentioned remote control example:

```
// A simple Java program to demonstrate
// implementation of Command Pattern using
// a remote control example.

// An interface for command
interface Command
{
   public void execute();
}

// Light class and its corresponding command
// classes
class Light
{
   public void on()
   {
      System.out.println("Light is on");
   }
   public void off()
   {
      System.out.println("Light is off");
   }
}
class LightOnCommand implements Command
{
   Light light;

   // The constructor is passed the light it
   // is going to control.
   public LightOnCommand(Light light)
   {
      this.light = light;
   }
   public void execute()
   {
      light.on();
   }
}
class LightOffCommand implements Command
{
   Light light;
   public LightOffCommand(Light light)
   {
      this.light = light;
   }
   public void execute()
   {
        light.off();
   }
}

// Stereo and its command classes
class Stereo
{
   public void on()
   {
      System.out.println("Stereo is on");
   }
   public void off()
   {
      System.out.println("Stereo is off");
   }
   public void setCD()
   {
      System.out.println("Stereo is set " +
                         "for CD input");
   }
   public void setDVD()
   {
      System.out.println("Stereo is set"+
                       " for DVD input");
   }
   public void setRadio()
   {
      System.out.println("Stereo is set" +
                         " for Radio");
   }
   public void setVolume(int volume)
   {
      // code to set the volume
      System.out.println("Stereo volume set"
                         + " to " + volume);
   }
}
class StereoOffCommand implements Command
{
   Stereo stereo;
   public StereoOffCommand(Stereo stereo)
   {
      this.stereo = stereo;
   }
   public void execute()
   {
      stereo.off();
   }
}
class StereoOnWithCDCommand implements Command
{
     Stereo stereo;
     public StereoOnWithCDCommand(Stereo stereo)
     {
        this.stereo = stereo;
     }
     public void execute()
     {
        stereo.on();
        stereo.setCD();
        stereo.setVolume(11);
     }
}

// A Simple remote control with one button
class SimpleRemoteControl
{
   Command slot;  // only one button

   public SimpleRemoteControl()
   {
   }

   public void setCommand(Command command)
   {
      // set the command the remote will
      // execute
      slot = command;
   }

   public void buttonWasPressed()
   {
      slot.execute();
   }
}

// Driver class
class RemoteControlTest
{
   public static void main(String[] args)
   {
      SimpleRemoteControl remote =
                new SimpleRemoteControl();
      Light light = new Light();
      Stereo stereo = new Stereo();

      // we can change command dynamically
      remote.setCommand(new
                  LightOnCommand(light));
      remote.buttonWasPressed();
      remote.setCommand(new
              StereoOnWithCDCommand(stereo));
      remote.buttonWasPressed();
      remote.setCommand(new
                 StereoOffCommand(stereo));
      remote.buttonWasPressed();
     }
  }
```

Output:

```
Light is on
Stereo is on
Stereo is set for CD input
Stereo volume set to 11
Stereo is off
```

Notice that the remote control doesn't know anything about turning on the stereo. That information is contained in a separate command object. This reduces the coupling between them.

Advantages:

- Makes our code extensible as we can add new commands without changing existing code.
- Reduces coupling the invoker and receiver of a command.

Disadvantages:

- Increase in the number of classes for each individual command

References:

- Head First Design Patterns (book)
- https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/command

If This article is contributed by **Sulabh Kumar**. you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

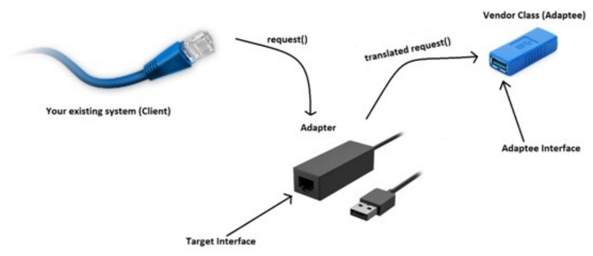
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design
design

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:

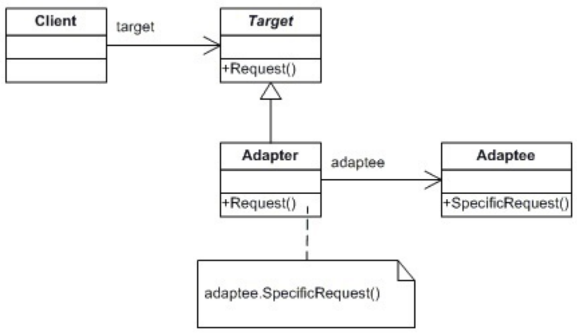


Image source = http://www.dofactory.com/net/adapter-design-pattern

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.

```
// 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();
        PlasticToyDuck 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();

        // bird behaving like a toy duck
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```

Output:

```
Sparrow...
Flying
Chirp Chirp
ToyDuck...
Squeak
BirdAdapter...
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:

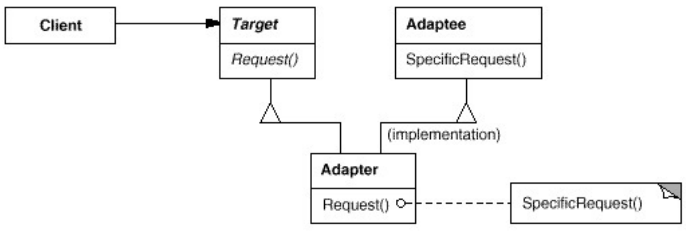


Image Source = http://www.bogotobogo.com/DesignPatterns/images/adapter/class_adapter.gif

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.

References:
Head First Design Patterns (Book)

This article is contributed by **Sulabh Kumar.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design
design

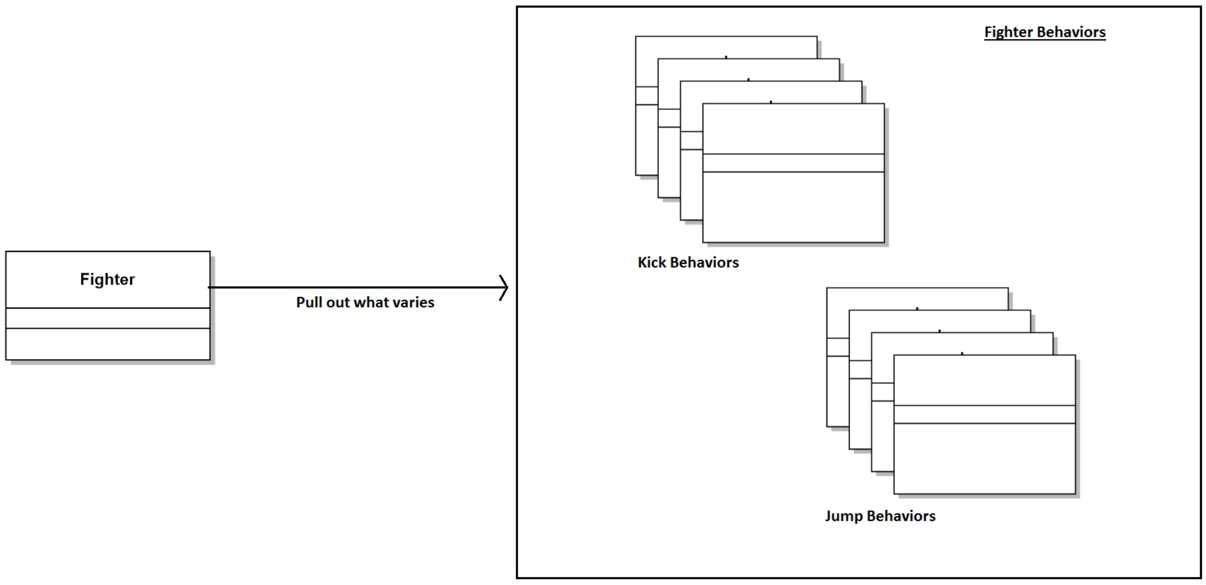
Strategy Pattern | Set 2 (Implementation)

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

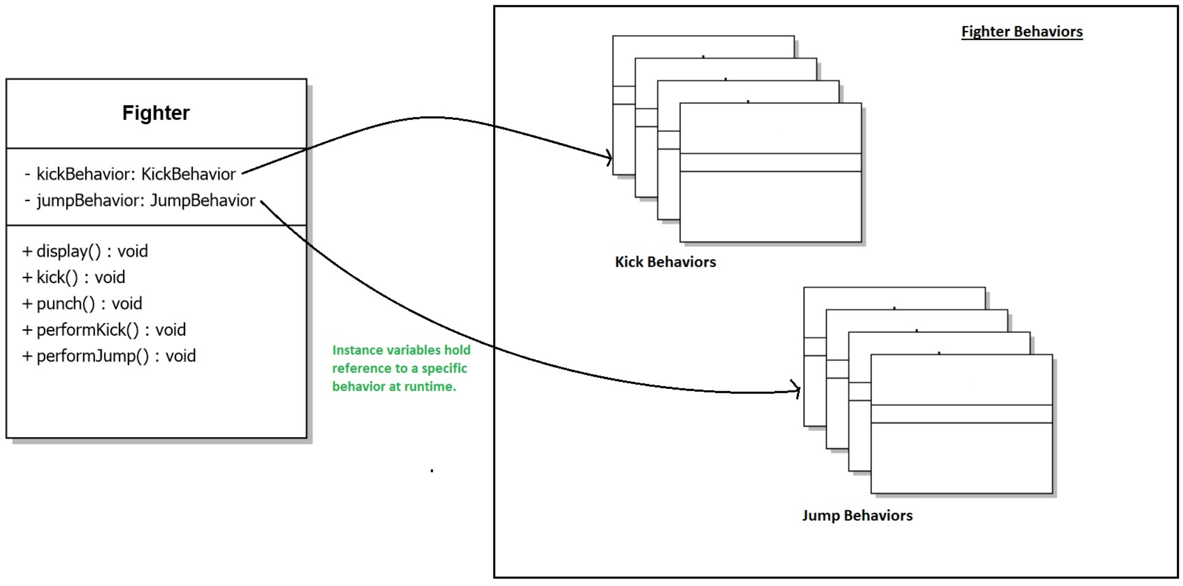
Strategy Pattern | Set 1 (Introduction)

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

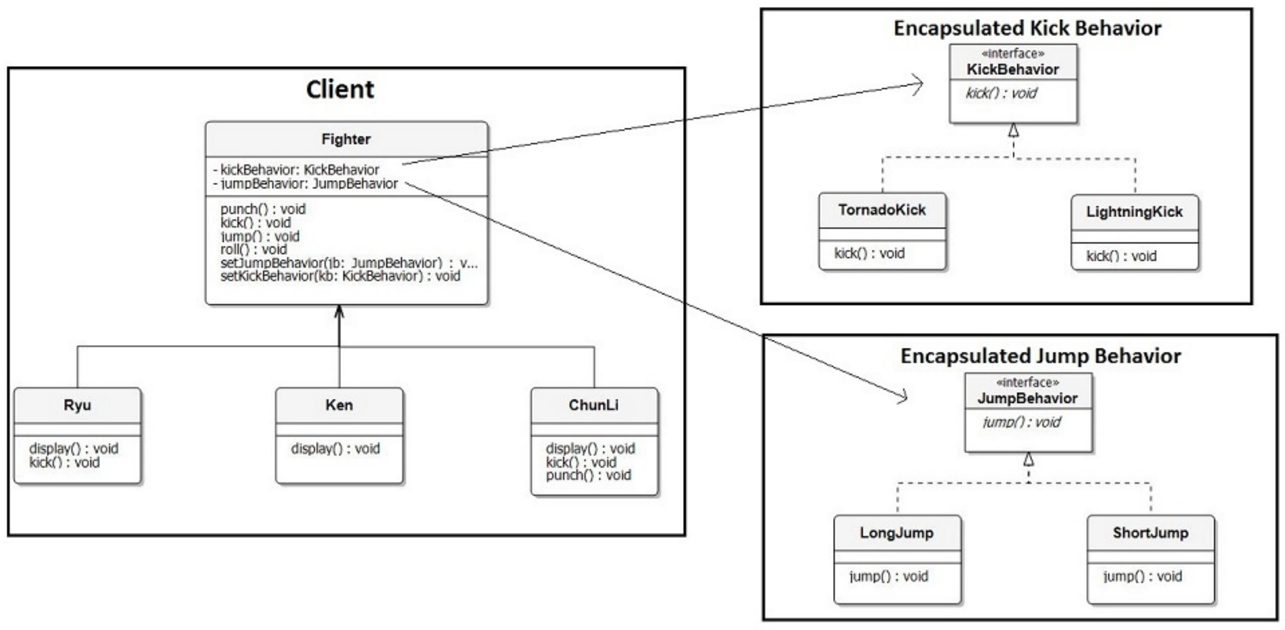
The first step is to identify the behaviors that may vary across different classes in future and separate them from the rest. For our example let them be kick and jump behaviors. To separate these behaviors we will pull both methods out of **Fighter** class and create a new set of classes to represent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

```
// 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();
}
```

```java
// 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
```

References:

Head First Design Patterns

This article is contributed by **Sulabh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

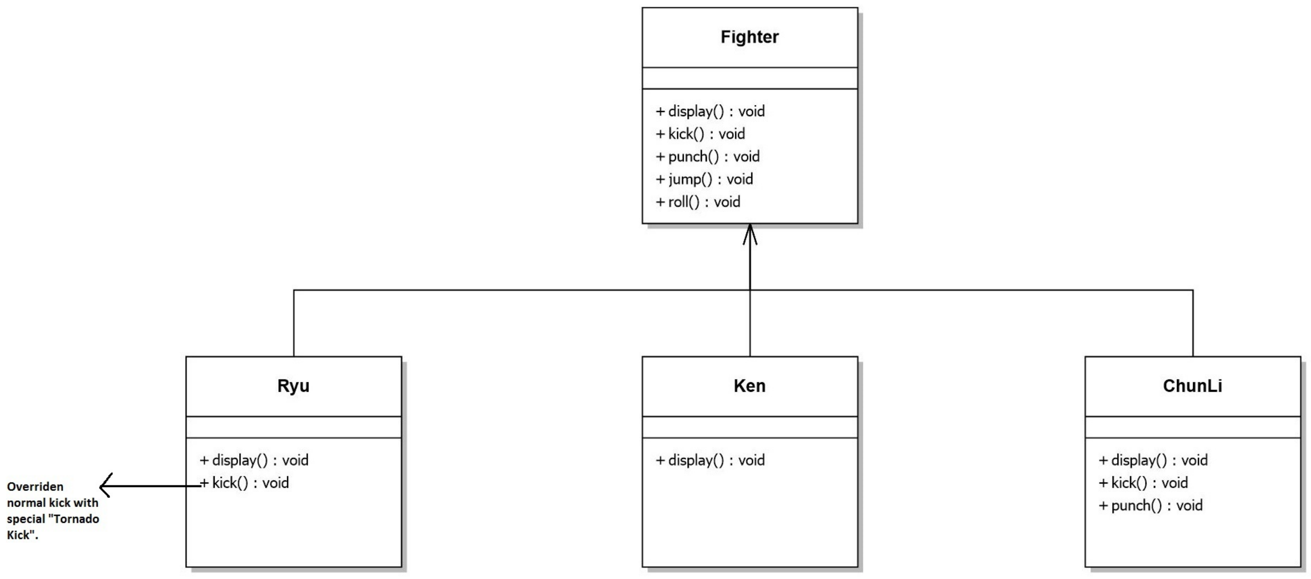
GATE CS Corner   Company Wise Coding Practice

Design
design

Strategy Pattern | 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:

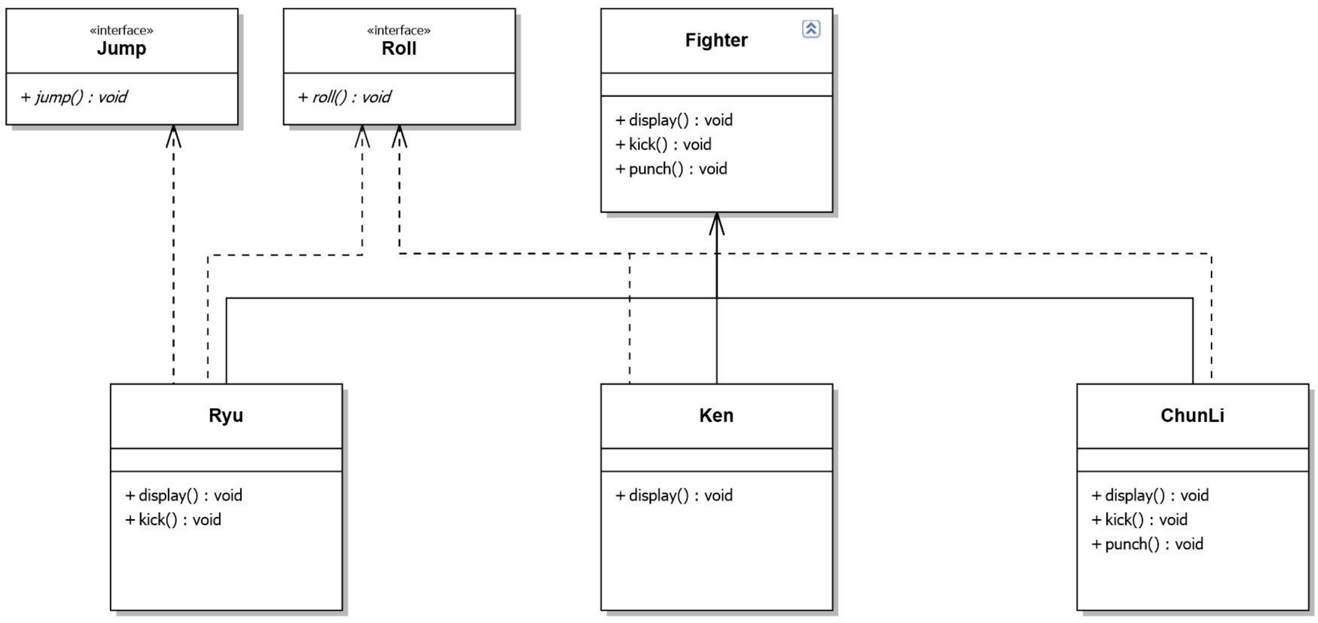


What are the problems with above design?

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.

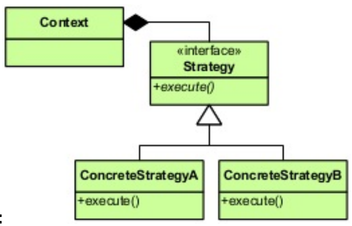
Definition:

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:**



Imgsrc = https://upload.wikimedia.org/wikipedia/commons/3/39/Strategy_Pattern_in_UML.png

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.

References:

- Head First Design Patterns
- http://wiki.expertiza.ncsu.edu/index.php/CSC/ECE_517_Fall_2007/wiki1b_8_sa
- https://en.wikipedia.org/wiki/Strategy_pattern

This article is contributed by **Sulabh Kumar.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

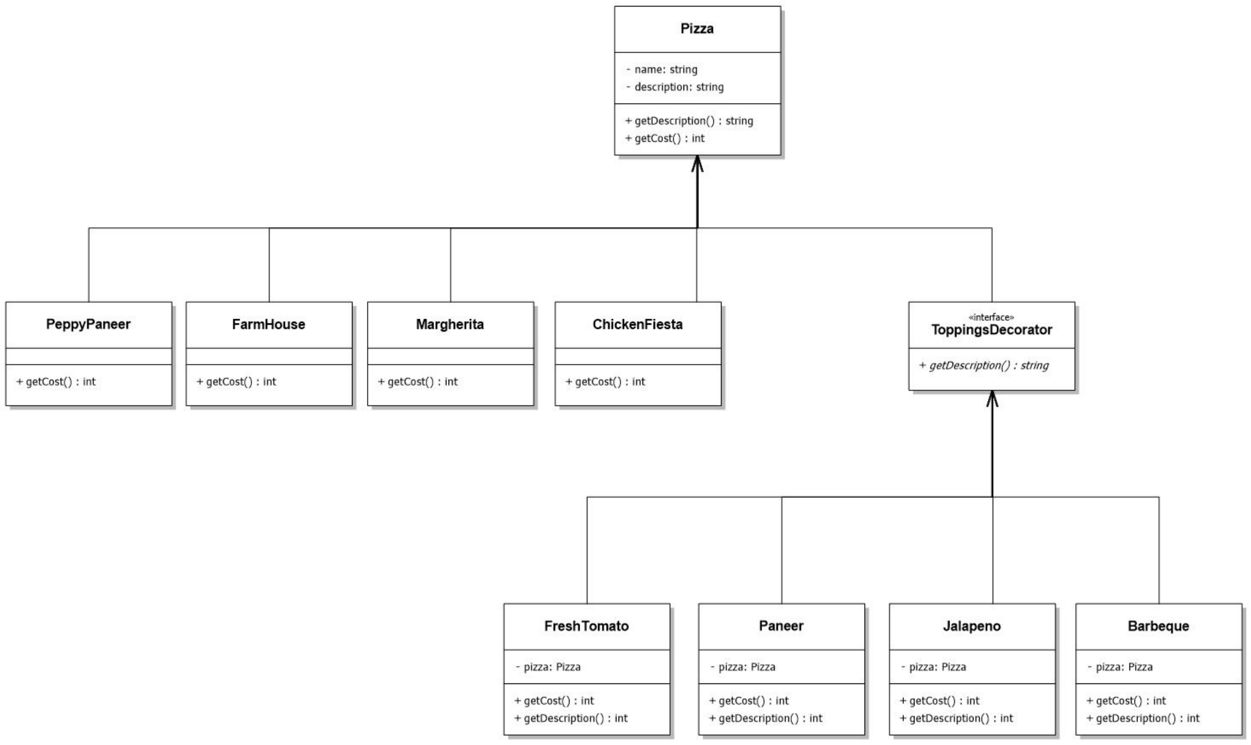
Design
design

Decorator Pattern | Set 3 (Coding the Design)

We have discussed Pizza design problem and different naive approaches to solve it in set 1. We have also introduced Decorator pattern in Set 2.

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.

```
// 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.

This article is contributed by **Sulabh Kumar.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

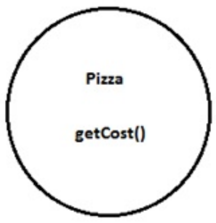
Design

The Decorator Pattern | Set 2 (Introduction and Design)

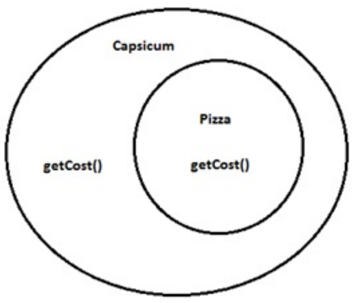
As we saw our previous designs using inheritance didn't work out that well. In this article, decorator pattern is discussed for the design problem in previous set.

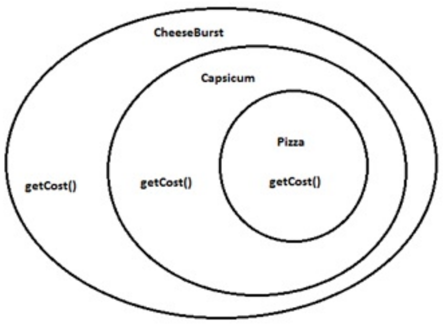
So what we do now is take a pizza and "decorate" it with toppings at runtime:

1. Take a pizza object.

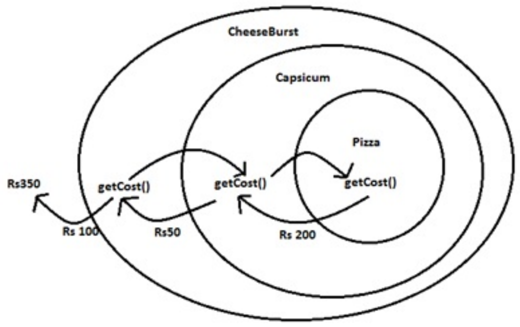


2. "Decorate" it with a Capsicum object.





3. "Decorate" it with a CheeseBurst object.



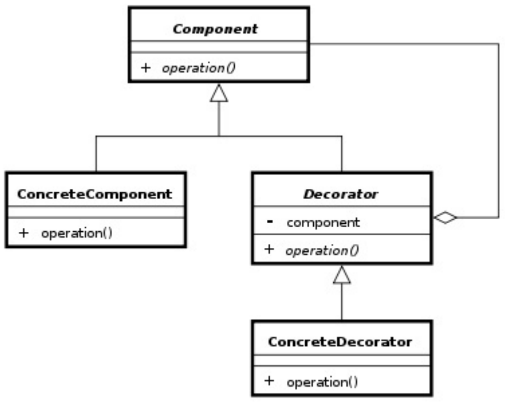
4. Call getCost() and use delegation instead of inheritance to calculate the toppings cost.

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 the 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

- 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.

References:

- Head First Design Patterns(Book)
- https://neillmorgan.wordpress.com/2010/02/07/decorator-pattern-pros-and-cons/
- https://en.wikipedia.org/wiki/Decorator_pattern
- http://stackoverflow.com/questions/4842978/decorator-pattern-versus-sub-classing

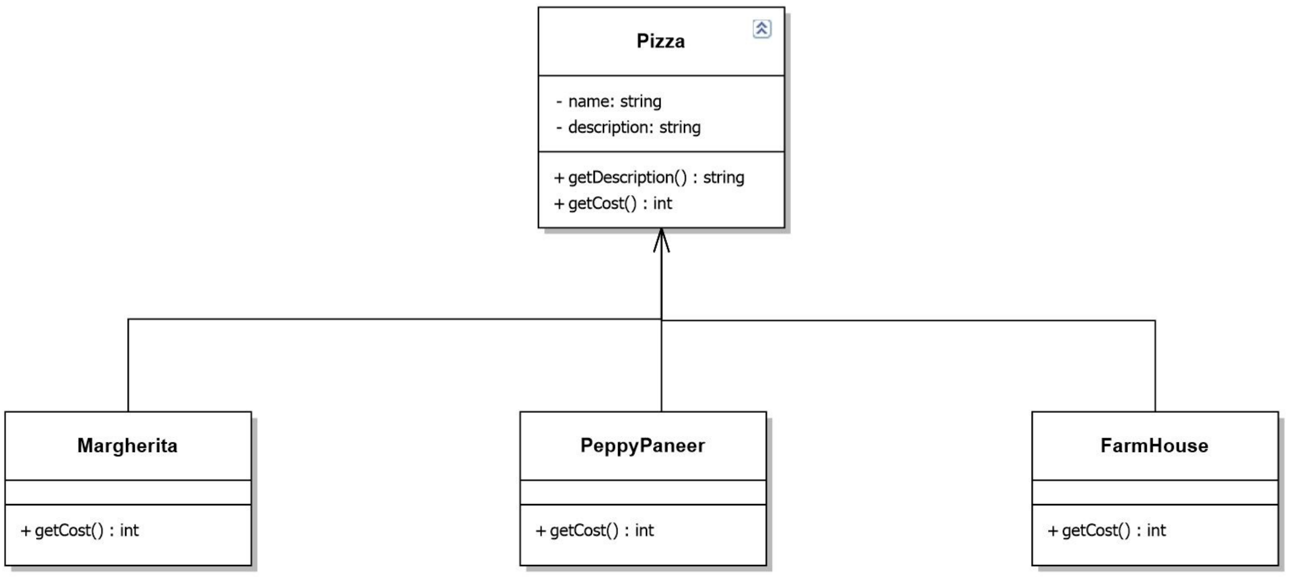
In the next post, we will be discussing implementation of decorator pattern.

This article is contributed by **Sulabh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Decorator Pattern | Set 1 (Background)

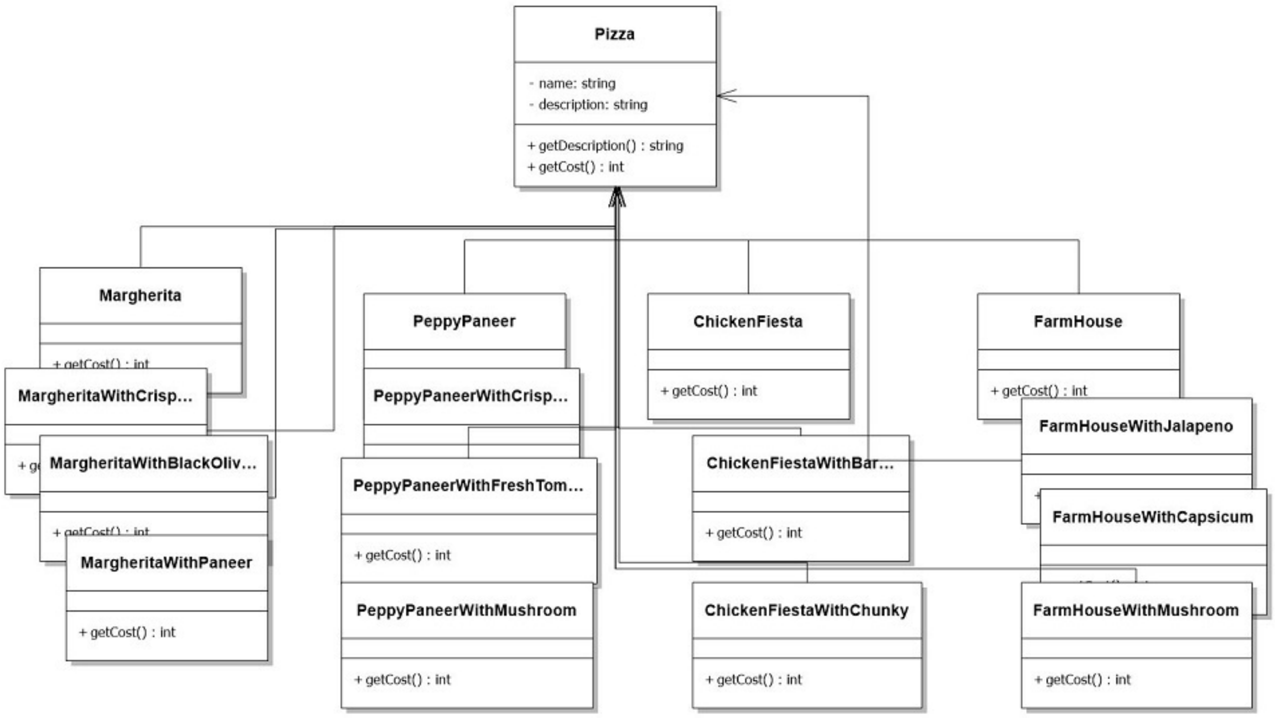
To understand decorator pattern let us consider a scenario inspired from the book "Head First Design Pattern". Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken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



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.

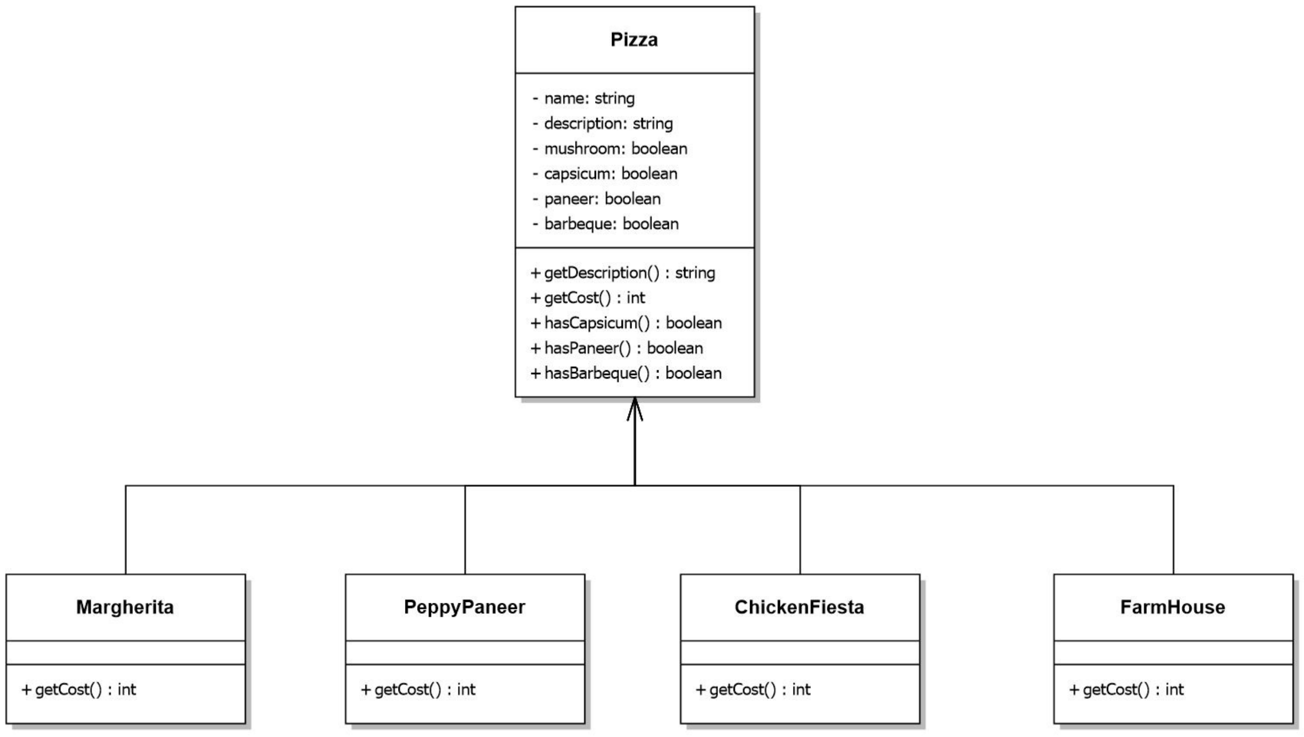
Let us look at various options.

Option 1
Create a new subclass for every topping with a pizza. The class diagram would look like:



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** which states that classes should be open for extension and closed for modification.

In the next set, we will be introducing Decorator Pattern and apply it to above above problem.

References: Head First Design Patterns(Book).

This article is contributed by **Sulabh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design
design

Singleton Design Pattern

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.

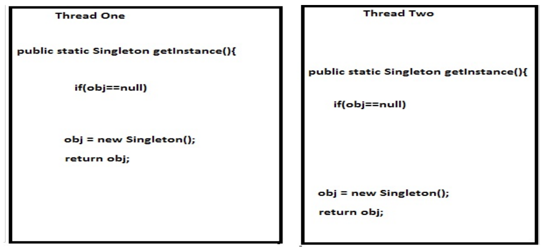
Let’s see various design options for implementing such a class. If you have a good handle on static class variables and access modifiers this should not be a difficult task.

Method 1: Classic Implementation

```
// 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.



```
Thread One

public static Singleton getInstance(){

          if(obj==null)


        obj = new Singleton();
        return obj;
```

```
Thread Two


public static Singleton getInstance(){

     if(obj==null)




   obj = new Singleton();
   return obj;
```

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

Method 2: make getInstance() synchronized

```
// 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

```
// Static initializer based Java implementation of
// singleton design pattern
class Singleton
{
```

```java
    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"

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.

```java
// 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 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.

References:

Head First Design Patterns book (Highly recommended)
https://en.wikipedia.org/wiki/Singleton_pattern

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design
Java
design

Observer Pattern | Set 2 (Implementation)

We strongly recommend to refer below Set 1 before moving on to this post.

Observer Pattern -Introduction

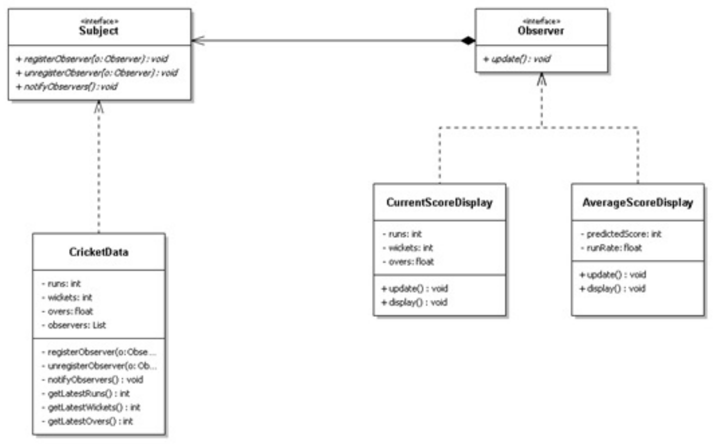
In Set 1, we discussed below problem, a solution for the problem without Observer pattern and problems with the solution.

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

Applying Observer pattern to above problem:

Let us see how we can improve the design of our application using observer pattern. If we observe the flow of data we can easily see that the CricketData and display elements follow subject-observers relationship.

New Class Diagram:



Java Implementation:

```java
// Java program to demonstrate working of
// onserver pattern
import java.util.ArrayList;
import java.util.Iterator;

// Implemented by Cricket data to communicate
// with observers
interface Subject
{
   public void registerObserver(Observer o);
   public void unregisterObserver(Observer o);
   public void notifyObservers();
}

class CricketData implements Subject
{
   int runs;
   int wickets;
   float overs;
   ArrayList<Observer> observerList;

   public CricketData() {
      observerList = new ArrayList<Observer>();
   }

   @Override
   public void registerObserver(Observer o) {
      observerList.add(o);
   }

   @Override
   public void unregisterObserver(Observer o) {
      observerList.remove(observerList.indexOf(o));
   }

   @Override
   public void notifyObservers()
   {
      for (Iterator<Observer> it =
           observerList.iterator(); it.hasNext();)
      {
         Observer o = it.next();
         o.update(runs,wickets,overs);
      }
   }

   // 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 90 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();

        notifyObservers();
    }
}

// This interface is implemented by all those
// classes that are to be updated whenever there
// is an update from CricketData
interface Observer
{
    public void update(int runs, int wickets,
                      float overs);
}

class AverageScoreDisplay implements Observer
{
    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);
    }
}

class CurrentScoreDisplay implements Observer
{
    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();

        // register display elements
        cricketData.registerObserver(averageScoreDisplay);
        cricketData.registerObserver(currentScoreDisplay);

        // in real app you would have some logic to
        // call this function when data changes
        cricketData.dataChanged();

        //remove an observer
        cricketData.unregisterObserver(averageScoreDisplay);

        // now only currentScoreDisplay gets the
        // notification
        cricketData.dataChanged();
    }
}
```

Output:

```
Average Score Display:
Run Rate: 8.823529
PredictedScore: 441

Current Score Display:
Runs: 90
Wickets:2
Overs: 10.2

Current Score Display:
Runs: 90
Wickets:2
Overs: 10.2
```

Note: Now we can add/delete as many observers without changing the subject.

References:

1. https://en.wikipedia.org/wiki/Observer_pattern
2. Head First Design Patterns book (highly recommended)

This article is contributed by **Sulabh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

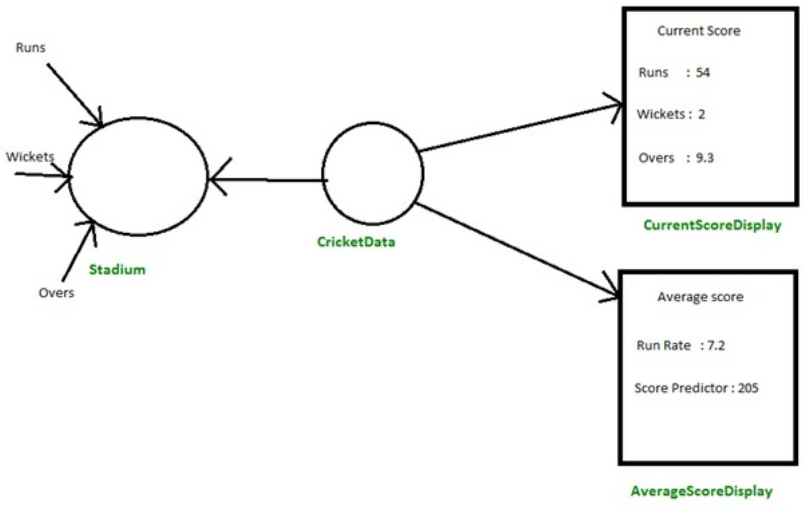
Design
design

Observer Pattern | Set 1 (Introduction)

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.

Below is the java implementation of this design.

```
// 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
```

```java
// 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 it 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:

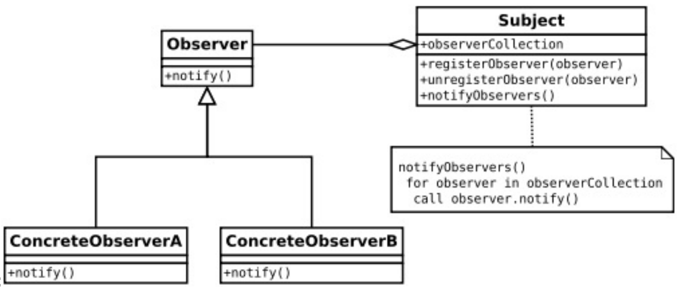


Image Source : Wikipedia

- 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 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.

Observer Pattern | Set 2 (Implementation)

This article is contributed by **Sulabh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

GATE CS Corner   Company Wise Coding Practice

Design
design

Design Patterns | Set 2 (Factory Method)

Factory method is a creational design pattern, 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:

```
// 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.

```
// 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 here.

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.
3. Also see this interview question.

Thanks to Abhijit Saha providing the first 2 examples.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice

Design Patterns | Set 1 (Introduction)

Design pattern is a general reusable solution or template to a commonly occurring problem in software design. The patterns typically show relationships and interactions between classes or objects. The idea is to speed up the development process by providing tested, proven development paradigm.

Goal:

• Understand the problem and matching it with some pattern.

• Reusage of old interface or making the present design reusable for the future usage.

Example:

For example, in many real world situations we want to create only one instance of a class. For example, there can be only one active president of country at a time regardless of personal identity. This pattern is called Singleton pattern. Other software examples could be 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.

Types of Design Patterns

There are mainly three types of design patterns:

1. Creational

These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

Creational design patterns are Factory Method, Abstract Factory, Builder, Singleton, Object Pool, Prototype and Singleton.

2. Structural

These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.

Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data and Proxy.

3. Behavioral

Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.

Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

References:

https://sourcemaking.com/design_patterns

https://sourcemaking.com/design_patterns/singleton

This article is contributed by **Abhijit Saha**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

GATE CS Corner   Company Wise Coding Practice