Flyweight Design Pattern

Read Discuss

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.

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.

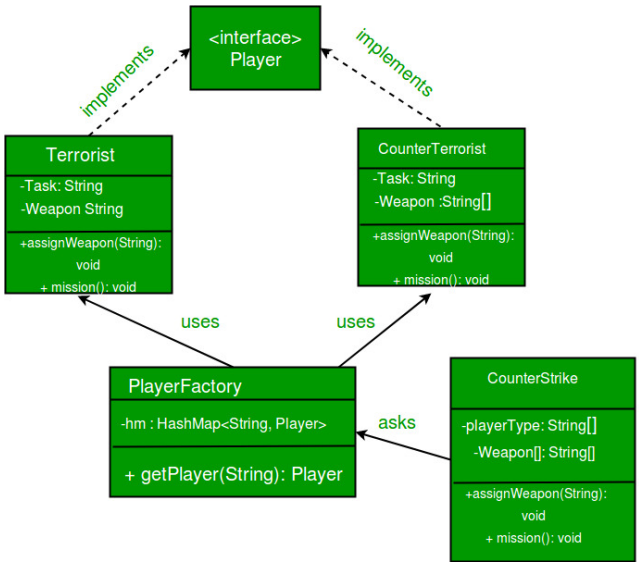
Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of Counter Strike. So we have 2 classes one for **T**errorist(**T**) and other for **C**ounter **T**errorist(**CT**). Whenever a player asks for a weapon we assign him the asked weapon. In the mission, terrorist's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

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
{
    // Intrinsic Attribute
    private final String TASK;

    // Extrinsic Attribute
    private 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);
    }
}

// Class used to get a player 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(IS) or CounterTerrorist(CT)   */
```

```
    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 weapon (used by getRandPlayerType()
    // and getRandWeapon()
    private static String[] playerType =
                    {"Terrorist", "CounterTerrorist"};
    private static String[] weapons =
      {"AK-47", "Maverick", "Gut Knife", "Desert Eagle"};
```

```
            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];
    }
}
```

```
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 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
```

Further Read – Flyweight Method in Python

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 review-team@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

Last Updated : 01 Sep, 2021

25

Similar Reads

3. Decorator Pattern | Set 3 (Coding the Design)
4. Singleton Design Pattern | Introduction
5. Java Singleton Design Pattern Practices with Examples
6. Proxy Design Pattern
7. Composite Design Pattern
8. Prototype Design Pattern
9. Mediator design pattern
10. Template Method Design Pattern

Next

Facade Design Pattern | Introduction

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Improve Article

Report Issue

GeeksforGeeks
A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Algorithms

Sorting

Searching

Greedy

Dynamic Programming

Pattern Searching

Recursion

Backtracking

Web Development

HTML

CSS

JavaScript

Bootstrap

ReactJS

AngularJS

NodeJS

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Python

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

Python Interview Question

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Deep Learning Tutorial

DevOps

Git

AWS

Docker

Kubernetes

Azure

GCP