

The Flyweight Pattern



Mátyás Lancelot Bors · [Follow](#)

4 min read · Oct 29, 2017



Listen



Share

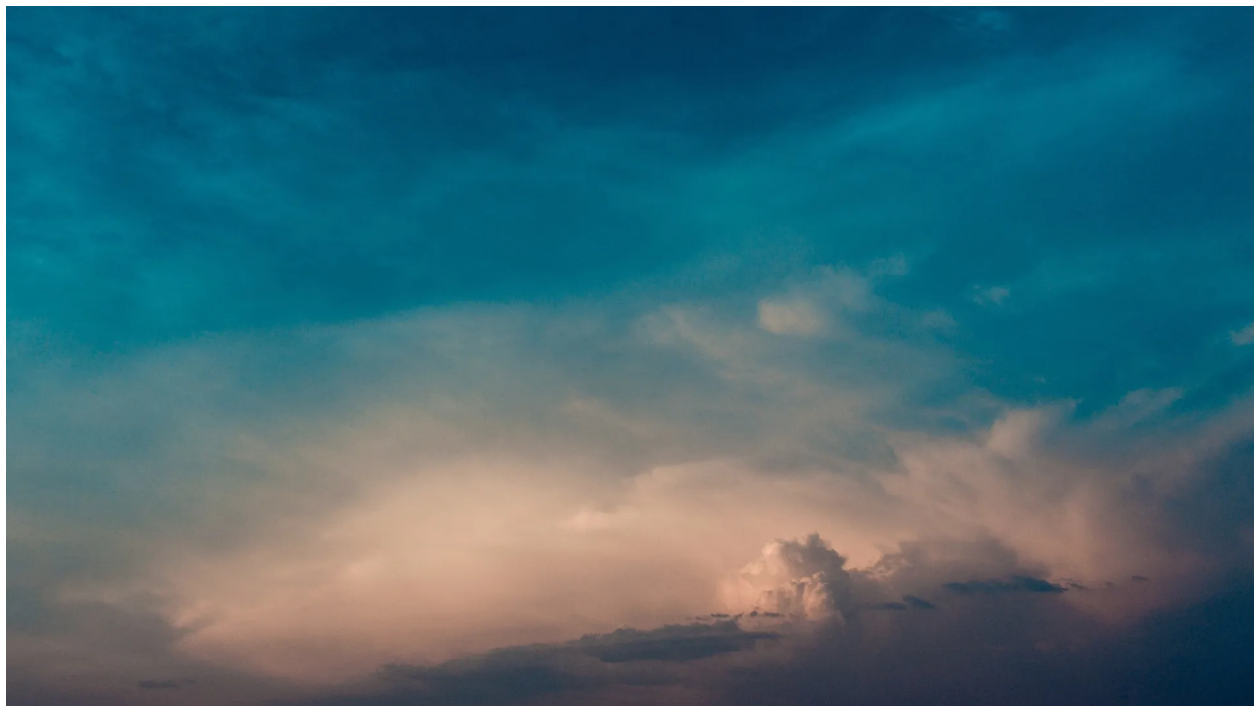


Image From [Pexels](#)

In this article we are going to take a look at the *Flyweight Pattern* to understand how it works and how we can use it.

Introduction

The *Flyweight Pattern* is a *Structural Pattern*. Its intent is to minimize memory usage. The famous *Gang of Four* defines it like so:

“Use sharing to support large numbers of fine-grained objects efficiently. A Flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context; it’s indistinguishable from an instance of the object that’s not shared.”

There is a lot of information in this definition. So, let’s illustrate it with a “real world” example: imagine that we are building a video game where a thousand characters have to fight on the battlefield. Each character is an object that contains, for example, a graphical representation, a behaviour, weapons and information about its location and its health. Creating such a number of objects will use a lot of memory. We can improve that by sharing the common information, such as graphical representation and behaviour. Health and location will, however, vary.

Now we have that in mind, let’s dig a little deeper.

The Pattern

The intent of the *Flyweight Pattern* is to use sharing to support a large number of objects that have part of their internal state in common while the other part of state varies.

A common practice is to keep state in external data structures and pass them to the *Flyweight Object* when needed. Each *Flyweight Object* is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part.

The intrinsic data is held in the properties of the *Flyweight Objects* that are shared. *Intrinsic State* is invariant, so it is *context* independent. In that sense, intrinsic properties make objects “constant”. The extrinsic data is context dependent. So it cannot be shared and must be passed in through arguments, but should never be stored within a shared *Flyweight Object*. In our video game example, the graphical representation would be the intrinsic data and the location and the health would be the extrinsic data.

This pattern is useful when we have a program using a large number of objects. This pattern is often combined with *Composite* to implement shared leaf nodes.

Structure

To apply this pattern, we need the following things:

Flyweight

It declares an interface through which *Flyweights* can receive and act on extrinsic state.

Concrete Flyweight

It implements the *Flyweight interface* and adds storage for intrinsic state

Unshared Concrete Flyweight

It provides a way to use this pattern without enforcing the shared concept.

Flyweight Factory

It creates and manages *Flyweight objects* and ensures that *Flyweights* are shared properly. It keeps track of the *Flyweights* that already exist and when it is asked to create a new *Flyweight*, it checks if a matching *Flyweight* already exists. If there is no object of the required type, it creates a *Flyweight* of the requested type, adds it to the pool, and returns a reference to the *Flyweight*.

Client

It maintains a reference to *Flyweights*, computes or stores the extrinsic state of those.

Example

Let's take back our example with our video game and let's make a really basic implementation of this pattern with *Java*.

First, we define the *Flyweight interface*:

```
public interface Character {  
    public void render();  
}
```

We can now create a concrete implementation:

```
public class CharacterA implements Character {  
    private String color;  
    private int x;  
    private int y;  
  
    public Circle(String color){  
        this.color = color;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
}
```

```

        public void setY(int y) {
            this.y = y;
        }

        public void render() {
            System.out.println("Character at position " + x + ", " + y + "
with color " + color);
        }
    }
}

```

Let's make the *Factory* that embeds a *Pool system*:

```

public class CharacterFactory {
    private static HashMap<String, Character> characterMap = new
HashMap();

    public static Character getCharacter(String color) {
        Character character = (Character)characterMap.get(color);

        if (character == null) {
            character = new CharacterA(color);
            characterMap.put(color, character);
        }

        return character;
    }
}

```

And we can make our *Client* like so:

```

public class CharacterClient {
    private static final String colors[] = { "Red", "Green", "Blue",
"White", "Black" };

    public static void main(String[] args) {
        for (int i = 0; i < 15; ++i) {
            Character character =
(Character)CharacterFactory.getCharacter(getRandomColor());
            character.setX(getRandomX());
            character.setY(getRandomY());
            character.render();
        }
    }

    private static String getRandomColor() {
        return colors[(int)(Math.random() * colors.length)];
    }
}

```

```
}  
  
private static int getRandomX() {  
    return (int)(Math.random() * 100);  
}  
  
private static int getRandomY() {  
    return (int)(Math.random() * 100);  
}  
  
}
```

Conclusion

In this article, we saw what the *Flyweight Pattern* is and how it works. This pattern is used to minimize memory usage by sharing as much as possible with similar objects. We saw that it is based on the following things: *Flyweight*, *Concrete Flyweight*, *Unshared Concrete Flyweight*, *Flyweight Factory* and *Client*.

One last word

If you like this article, you can consider supporting and helping me on [Patreon](#)! It would be awesome! Otherwise, you can find my other posts on [Medium](#) and [Tumblr](#). You will also know more about myself on [my personal website](#). Until next time, happy headache!

[Design Patterns](#)[Programming](#)[Development](#)[Java](#)[Follow](#)

Written by Mátyás Lancelot Bors

283 Followers

WebDeveloper / Writer / Musician / <https://www.mlbors.com> / <https://medium.com/@mlbors>

More from Mátyás Lancelot Bors



Open in app ↗

Sign up

Sign In



Mátyás Lancelot Bors

Using the Repository Pattern with the Entity Framework

Through this article, we are going to see how to use the Repository Pattern with the Entity Framework in an ASP.NET MVC application.

7 min read · Mar 10, 2018



222



4





Mátyás Lancelot Bors

What is a Finite State Machine?

In this article, we are going to see what a Finite State Machine is.

4 min read · Mar 10, 2018



446



4



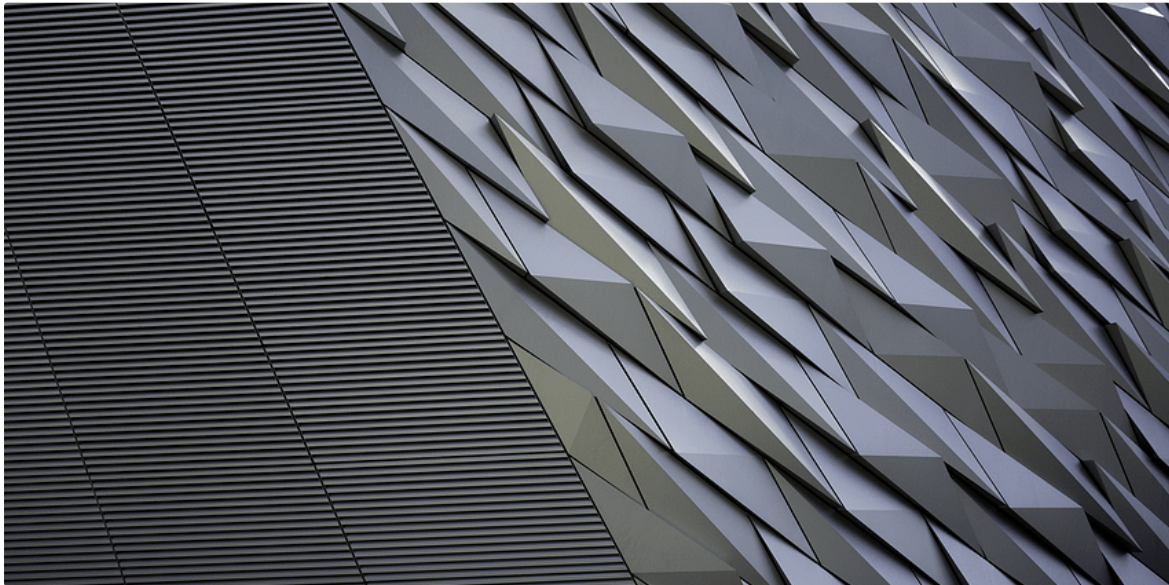
Mátyás Lancelot Bors

Preconditions and Postconditions

In this article, we are going to talk about the terms “precondition” and “postcondition”.

3 min read · Mar 10, 2018

 72  2



 Mátyás Lancelot Bors

Architectural Styles and Architectural Patterns

Through this article, we are going to take a look at what we call Architectural Styles and Architectural Patterns.

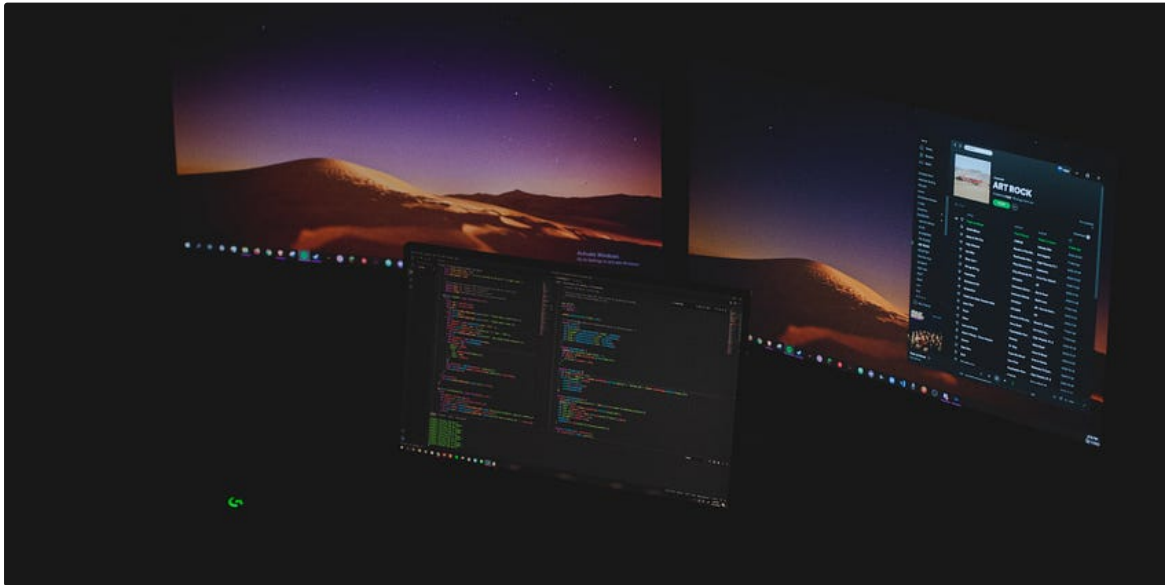
5 min read · Feb 11, 2018

 313  2



See all from Mátyás Lancelot Bors

Recommended from Medium



Larry | Peng Yang in Mastering Java

Understanding Java threads and concurrency — Part 2


Thread safety and how to achieve it

★ · 16 min read · Nov 26, 2022



4



 Gerardo Fernández

The SOLID principles


The SOLID principles explained in detail for the development of maintainable and robust applications


★ · 9 min read · Jan 9


 2 




Lists

- 

Stories to Help You Grow as a Software Developer
19 stories · 24 saves
- 

Leadership
30 stories · 9 saves
- 

How to Run More Meaningful 1:1 Meetings
11 stories · 14 saves
- 

Stories to Help You Level-Up at Work
19 stories · 20 saves



Larry | Peng Yang in Mastering Java

Mastering Java Streams API with Examples

A guide to Java stream operations and respective functions

★ · 13 min read · Nov 18, 2022



56



2



Ivan Polovyi in Javarevisited

Java CompletableFutures in Spring Boot

When Java 8 was released developers got many game-changing features. Among them was a CompletableFuture a significant improvement in...

★ · 9 min read · Dec 24, 2022

 95  1



Larry | Peng Yang in Mastering Java


Understanding Java threads and concurrency — Part 4

Let's dig into Executor Framework

★ · 15 min read · Feb 10

 3 



 Suraj Mishra in Level Up Coding

How to Implement Idempotent API (Part 1)

Discussing idempotency concept and implementation design

★ · 5 min read · Jan 27



84



1



See more recommendations