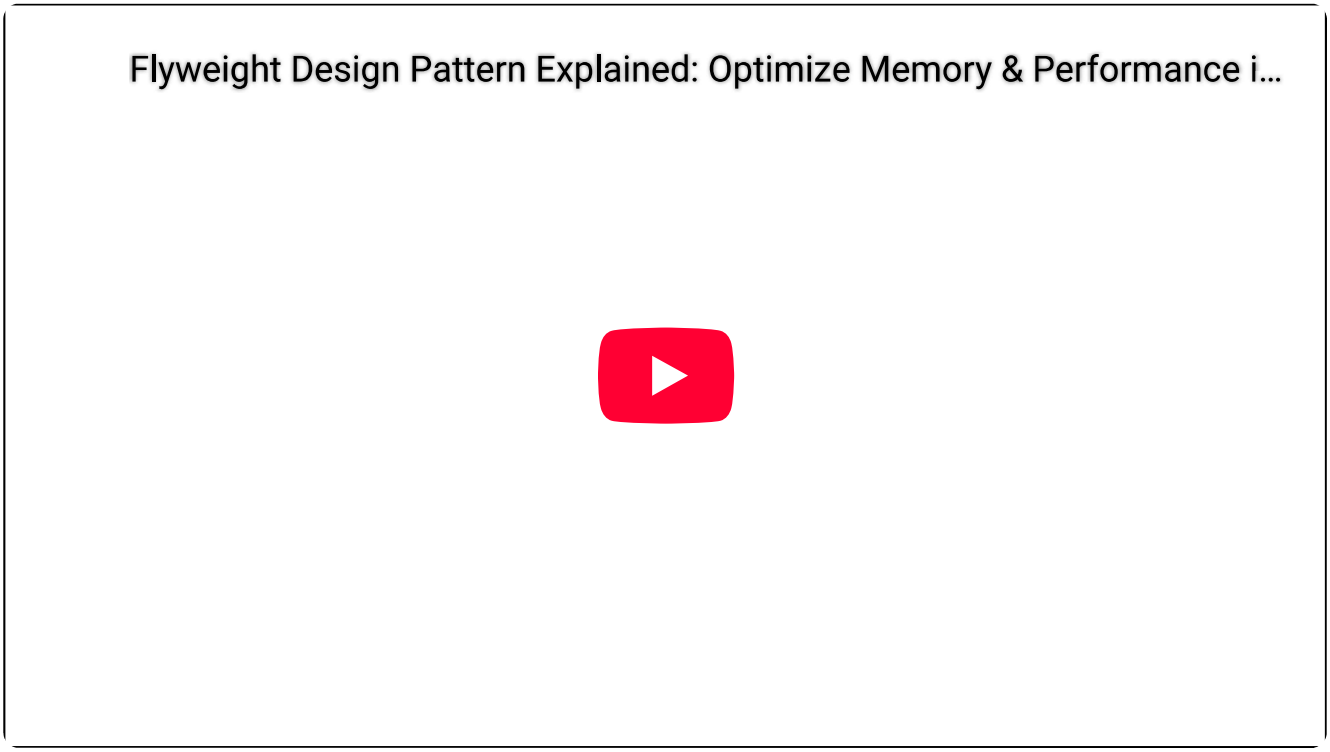
Search articles...

Flyweight Design Pattern



Topic Tags:

System design LLD

Problem Statement: Managing Memory Usage 🎮

Imagine you're developing a massive multiplayer game where thousands of particles need to be rendered simultaneously (e.g., bullets, explosions, or visual effects). Each particle object contains properties like position, velocity, and appearance. Creating separate objects for each particle would consume excessive memory.

The Problem: Creating individual objects for repetitive elements can lead to:
• Excessive memory consumption
• Poor system performance
• Increased garbage collection overhead
The Challenge: How can you create a solution that efficiently handles large numbers of similar objects while minimizing memory usage?

Solving It the Traditional Way: A Memory-Intensive Solution 🛠️

Here's a naive implementation where each particle is created as a separate object:

Particle.java

Java

```java
// Particle.java
public class Particle {
  private String color;
  private String sprite;
  private float x;
  private float y;
  private float velocity;
  public Particle(
      String color, String sprite, float x, float y, float velocity) {
    this.color = color;
    this.sprite = sprite;
    this.x = x;
    this.y = y;
    this.velocity = velocity;
  }

  public void update() {
    // Update particle position
    y += velocity;
    System.out.println(
        "Particle at position (" + x + "," + y + ") with color " + color);
  }
}
```

Game.java

Java

```java
// Game.java
public class Game {
  public static void main(String[] args) {
    List<Particle> particles = new ArrayList<>();
    // Create thousands of particles
    for (int i = 0; i < 1000; i++) {
      particles.add(new Particle("red", "explosion.png",
```

```
            (float) Math.random() * 100, (float) Math.random() * 100, 1.0f))
    }

    // Update all particles
    for (Particle particle : particles) {
      particle.update();
    }
  }
}
```

The Problems with This Approach:

1. Memory Waste: Each particle object stores identical sprite and color data
2. Object Overhead: Creating thousands of objects puts pressure on garbage collection
3. Poor Performance: Large number of objects leads to slower execution

Interviewer's Follow-up Questions: Can We Improve the Code? 🤔

An interviewer might ask:
• How can we separate intrinsic (shared) state from extrinsic (unique) state?
• What if we need to support different types of particles with shared properties?
• How can we ensure that shared properties are not duplicated in memory?

Ugly Code: When Memory Usage Becomes a Problem 🤦

Let's say we try to optimize by caching some properties:

ParticleSystem.java

Java

```java
public class ParticleSystem {
  private Map<String, String> spriteCache = new HashMap<>();
  private List<Particle> particles = new ArrayList<>();
  public void createParticle(
      String color, String spritePath, float x, float y, float velocity) {
    // Try to reuse sprite from cache
    String sprite =
        spriteCache.computeIfAbsent(spritePath, path -> loadSprite(path));
    particles.add(new Particle(color, sprite, x, y, velocity));
  }
```

```
11
12    private String loadSprite(String path) {
13      // Simulate loading sprite
14      return "Loaded: " + path;
15    }
16  }
```

Why is this Code Problematic?

• Partial Solution: Only addresses sprite sharing, not other common properties
• Complex Management: Manual caching adds complexity
• Limited Scalability: Doesn't fully solve the memory usage problem

The Savior: Flyweight Design Pattern 🙏

The Flyweight Pattern is the perfect solution for this problem. It minimizes memory usage by sharing common properties between multiple objects, while keeping the unique state external.

How the Flyweight Pattern Works 🔧

1. Intrinsic State: Shared properties stored in flyweight objects
2. Extrinsic State: Unique properties passed as parameters
3. Flyweight Factory: Manages flyweight object creation and sharing

Solving the Problem with Flyweight Pattern 🌐

Let's implement the Flyweight pattern step by step. The key idea is to split our particle system into two parts:
1. Intrinsic (shared) state:
Properties like color and sprite that can be shared among many particles

2. Extrinsic (unique) state:
Properties like position and velocity that must be unique to each particle

Step 1: Define the Particle Properties :

First, we create the ParticleType class that will act as our flyweight. This class holds only the intrinsic state - the properties that can be shared across multiple particles. Think of it as a template that defines how a particular type of particle looks and behaves.

ParticleType.java

Java

```java
// ParticleType.java (Flyweight)
public class ParticleType {
    private final String color;
    private final String sprite;
    public ParticleType(String color, String sprite) {
        this.color = color;
        this.sprite = sprite;
    }

    public void render(float x, float y, float velocity) {
        System.out.println("Rendering " + color + " particle at (" + x + "
                           ") with sprite " + sprite);
    }
}
```

Step 2: Create the Flyweight Factory :

The factory is crucial for the Flyweight pattern - it ensures we don't create duplicate flyweights. Think of it as a cache that keeps track of all particle types we've already created. When we need a particle type, we first check if we already have one with the same properties, and only create a new one if necessary.

ParticleTypeFactory.java

Java

```java
// ParticleTypeFactory.java
public class ParticleTypeFactory {
    private Map<String, ParticleType> particleTypes = new HashMap<>();
    public ParticleType getParticleType(String color, String sprite) {
        String key = color + "_" + sprite;
        return particleTypes.computeIfAbsent(key,
            k -> new ParticleType(color, sprite));
    }
}
```

Step 3: Create the Particle Using Flyweight

Now we create the actual Particle class that players will see in the game. Instead of storing color and sprite directly, it references a ParticleType flyweight. This class maintains only the extrinsic state (position and velocity) that must be unique for each particle. This separation is what makes the Flyweight pattern memory-efficient.

Particle.java

Java

```java
// Particle.java
public class Particle {
    private ParticleType type; // reference to flyweight
    private float x;
    private float y;
    private float velocity;

    public Particle(ParticleType type, float x, float y, float velocity) {
        this.type = type;
        this.x = x;
        this.y = y;
        this.velocity = velocity;
    }

    public void update() {
        y += velocity;
        type.render(x, y, velocity);
    }
}
```

Step 4: Use the Pattern in the Game

Finally, we bring it all together in the game. Notice how we create just one ParticleType for all explosion particles, but each particle has its own position and velocity. This is the Flyweight pattern in action - sharing the heavy resources (sprite and color data) while allowing for individual particle behavior.

Game.java

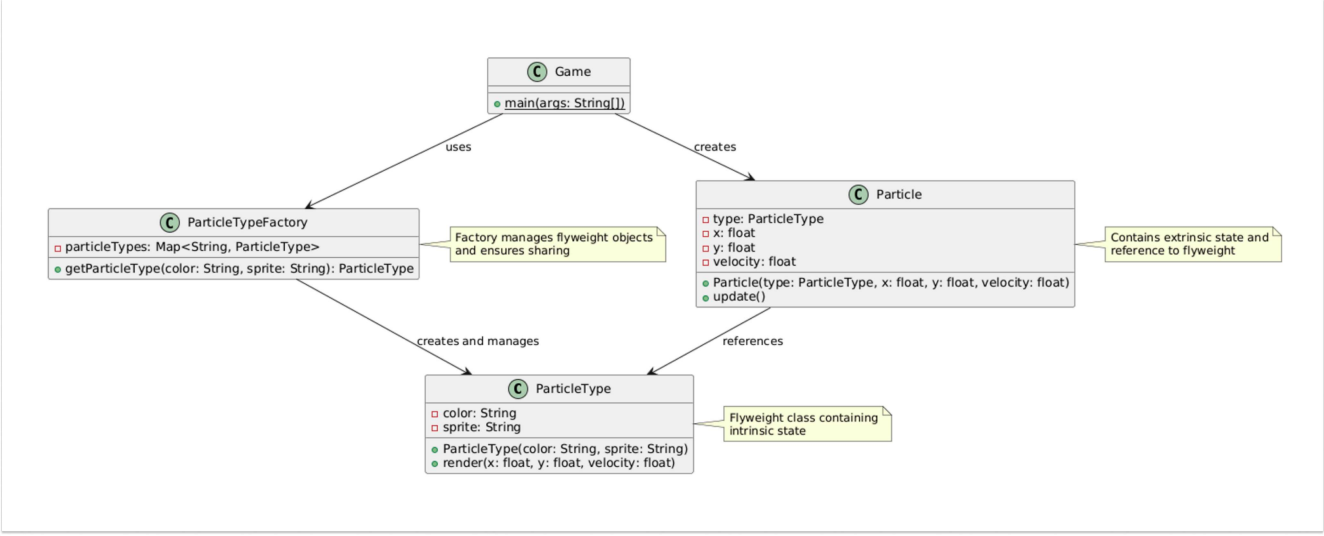
Java

```java
// Game.java
public class Game {
    public static void main(String[] args) {
        ParticleTypeFactory factory = new ParticleTypeFactory();
        List<Particle> particles = new ArrayList<>();
        // Create thousands of particles using shared flyweights
        ParticleType explosionType = factory.getParticleType("red", "explo

        for (int i = 0; i < 1000; i++) {
            particles.add(new Particle(explosionType,
                                       (float) Math.random() * 100,
                                       (float) Math.random() * 100,
                                       1.0f));
        }
        // Update all particles
        for (Particle particle : particles) {
            particle.update();
        }
    }
}
```

In this implementation, we've achieved significant memory savings. Instead of storing color and sprite data for each of the 1000 particles (which would require 1000 copies), we store this data just once in the ParticleType flyweight and share it across all particles. Each particle only needs to store its own position and velocity, plus a reference to the shared ParticleType.

The magic of the Flyweight pattern lies in this sharing mechanism. If we create 1000 red explosion particles, they all share the same ParticleType instance, dramatically reducing memory usage. If we later need blue explosion particles, we'll create just one more ParticleType instance for all blue particles to share.



Advantages of Using the Flyweight Pattern 🏆

1. Memory Efficiency: Dramatically reduces memory usage by sharing common data
2. Performance: Fewer objects mean better garbage collection performance
3. Scalability: Supports large numbers of similar objects efficiently
4. Maintenance: Clear separation between shared and unique state

Real-life Use Cases of the Flyweight Pattern 🌎

1. Text Editors: Sharing character formatting data
2. Game Development: Terrain tiles, particles, and visual effects
3. Graphics Applications: Sharing texture and material data
4. Web Browsers: Caching and reusing rendered elements

Conclusion 🎯

The Flyweight Design Pattern is an essential tool for optimizing memory usage in applications that deal with large numbers of similar objects. By separating intrinsic and extrinsic state, it allows efficient sharing of common data while maintaining the flexibility to handle unique properties for each instance.

Whether you're developing games, text editors, or graphics applications, the Flyweight Pattern helps you create memory-efficient and scalable solutions. Its ability to balance resource usage with performance makes it an invaluable pattern in modern software development.