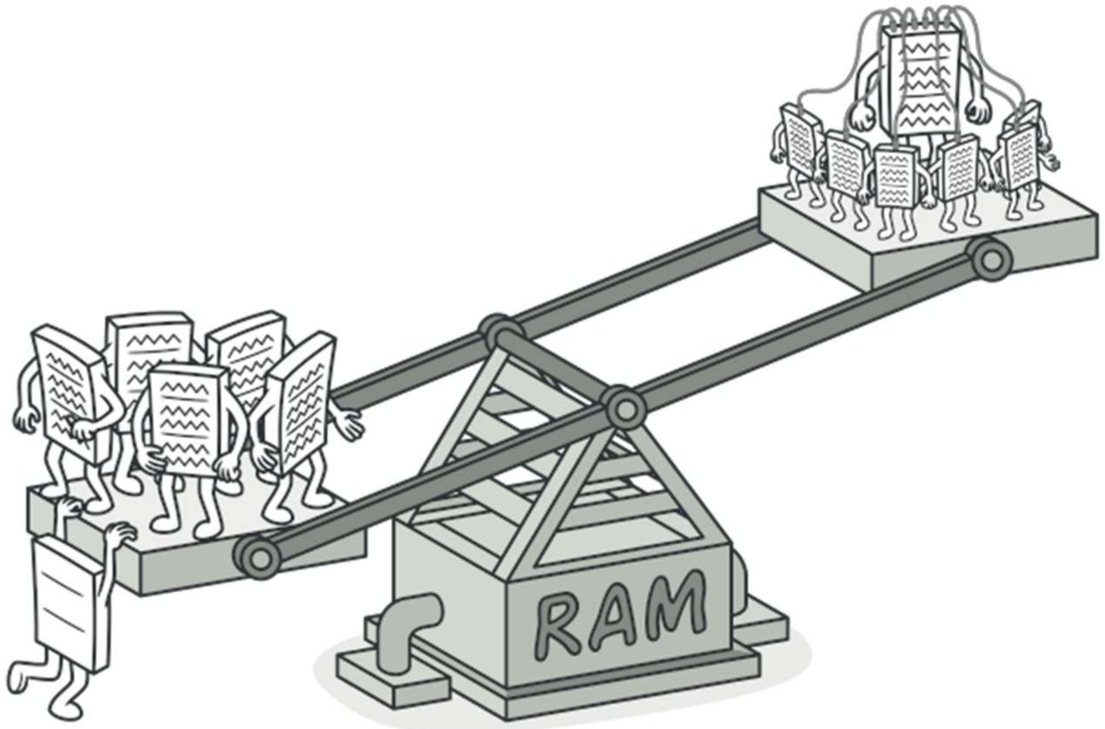
Flyweight Design Pattern



Definition

- **Flyweight** is a structural design pattern
- That lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Intent

“Use Sharing to support large numbers of fine-grained objects efficiently.”

Simply put, a method for storing a small number of complex objects that are used repeatedly

Keywords

Flyweight

A shared object that can be used in multiple contexts simultaneously.

Intrinsic

State information that is independent of the flyweights context. Shareable Information.

Extrinsic

State information that depends on the context of the flyweight and cannot be shared.

Applicability

Use the Flyweight when all of the following are true:

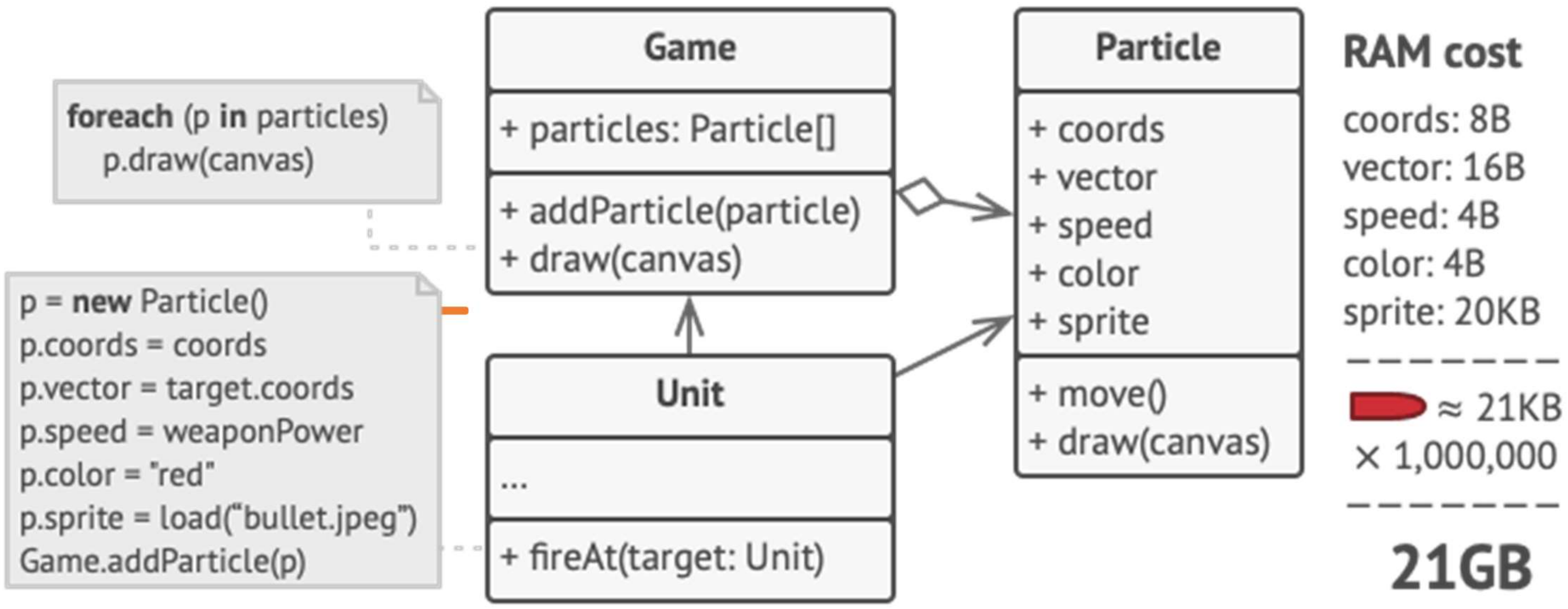
- Application has a large number of objects.
- Storage costs are high because of the large quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few once you remove their extrinsic state.
- The application doesn’t depend on object identity

Case study

- Design A game such as counter strike
- players would be moving around a map and shooting each other.
- You chose to implement a realistic particle system and make it a distinctive feature of the game.
- Vast quantities of bullets, missiles, and shrapnel from explosions should fly all over the map and deliver a thrilling experience to the player.

Problem

- The game kept crashing after a few minutes of gameplay.
- After spending several hours digging through debug logs, you discovered that the game crashed because of an insufficient amount of RAM



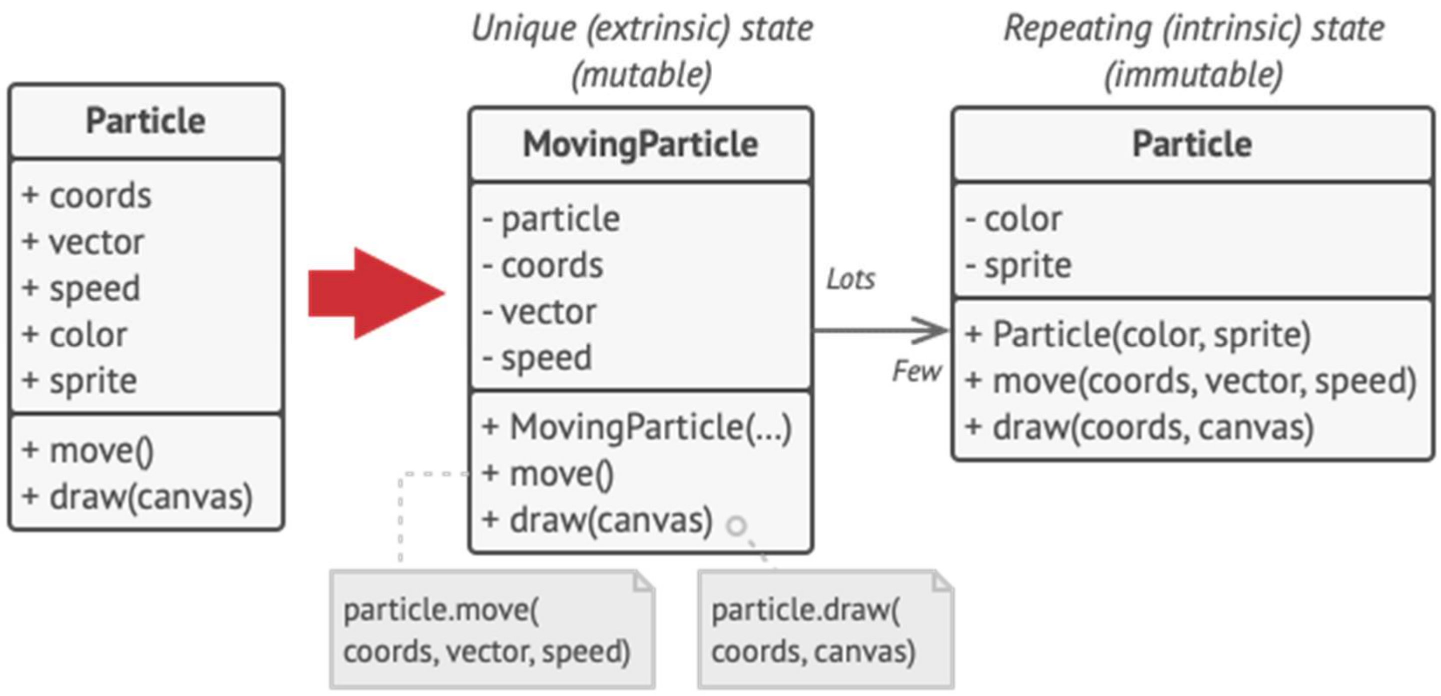
```
foreach (p in particles)
    p.draw(canvas)
```

```
p = new Particle()
p.coords = coords
p.vector = target.coords
p.speed = weaponPower
p.color = "red"
p.sprite = load("bullet.jpeg")
Game.addParticle(p)
```

- The actual problem was related to your particle system. Each particle, such as a bullet, a missile or a piece of shrapnel was represented by a separate object containing plenty of data.

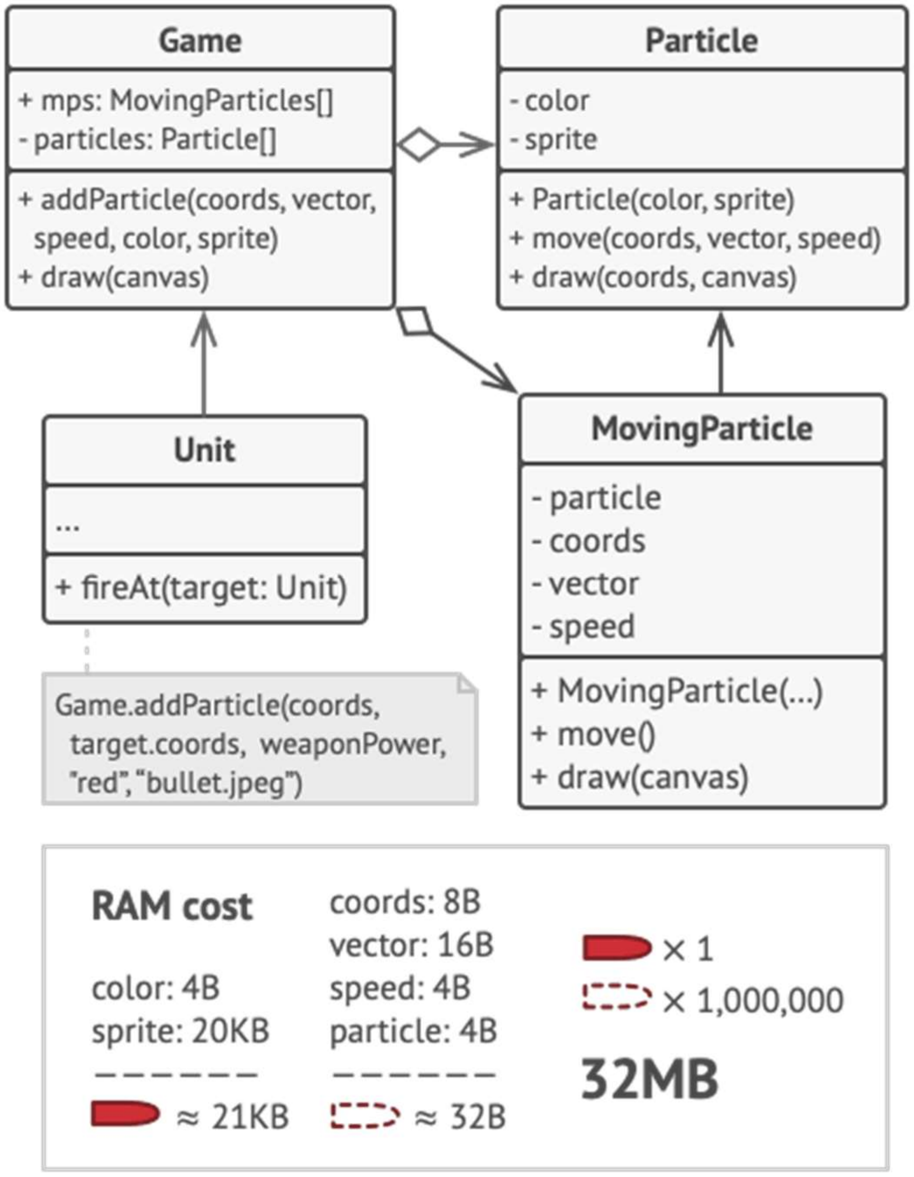
Solution

- On closer inspection of the Particle class, you may notice that the color and sprite fields consume a lot more memory than other fields. What’s worse is that these two fields store almost identical data across all particles.

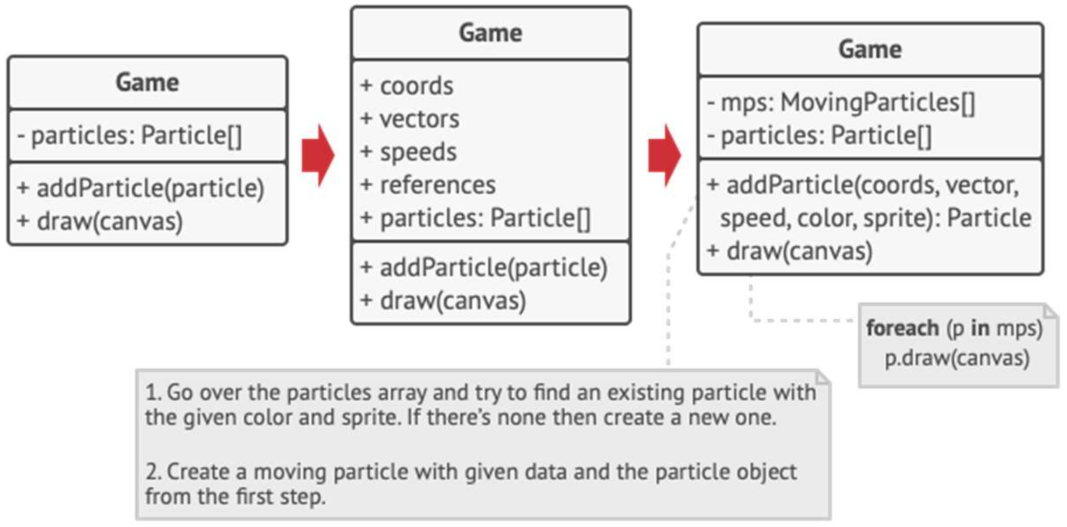


Use Flyweight

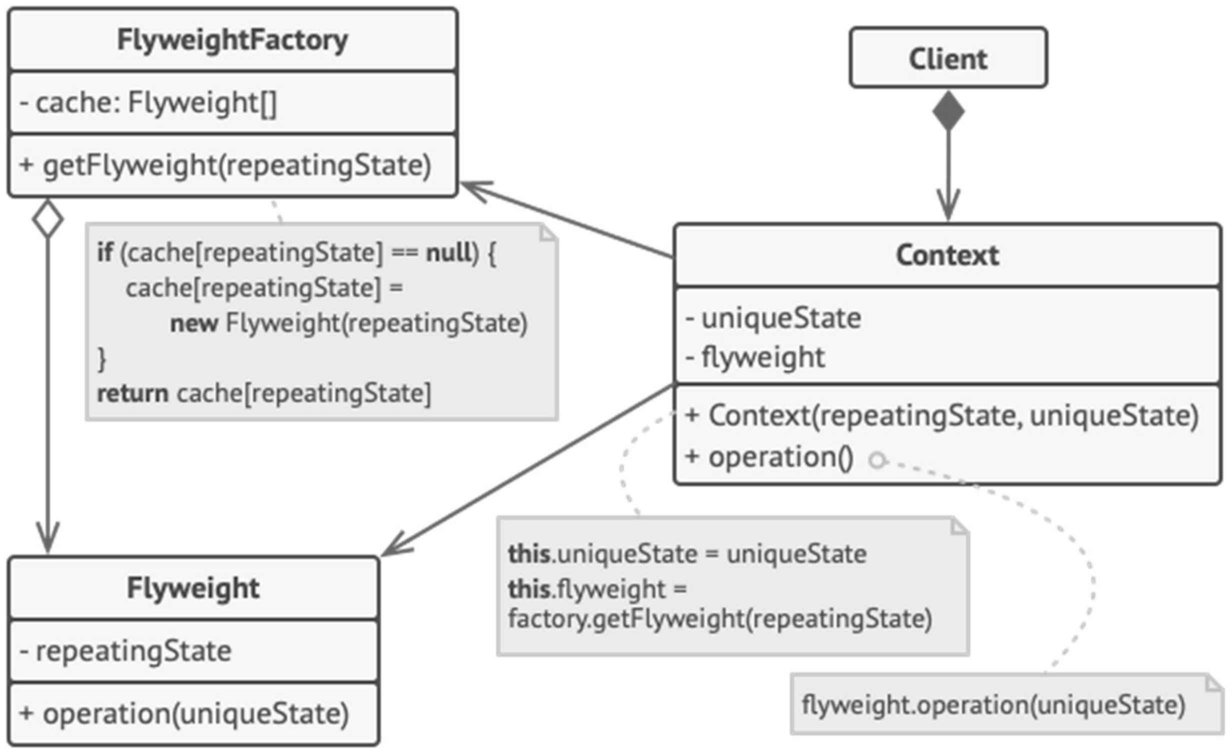
- The Flyweight pattern suggests that you stop storing the extrinsic state inside the object.



Extrinsic state storage


Game
- particles: Particle[]
+ addParticle(particle)
+ draw(canvas)
Game
+ coords
+ vectors
+ speeds
+ references
+ particles: Particle[]
+ addParticle(particle)
+ draw(canvas)
Game
- mps: MovingParticles[]
- particles: Particle[]
+ addParticle(coords, vector, speed, color, sprite): Particle
+ draw(canvas)
foreach (p in mps) p.draw(canvas)
1. Go over the particles array and try to find an existing particle with the given color and sprite. If there's none then create a new one.
2. Create a moving particle with given data and the particle object from the first step.

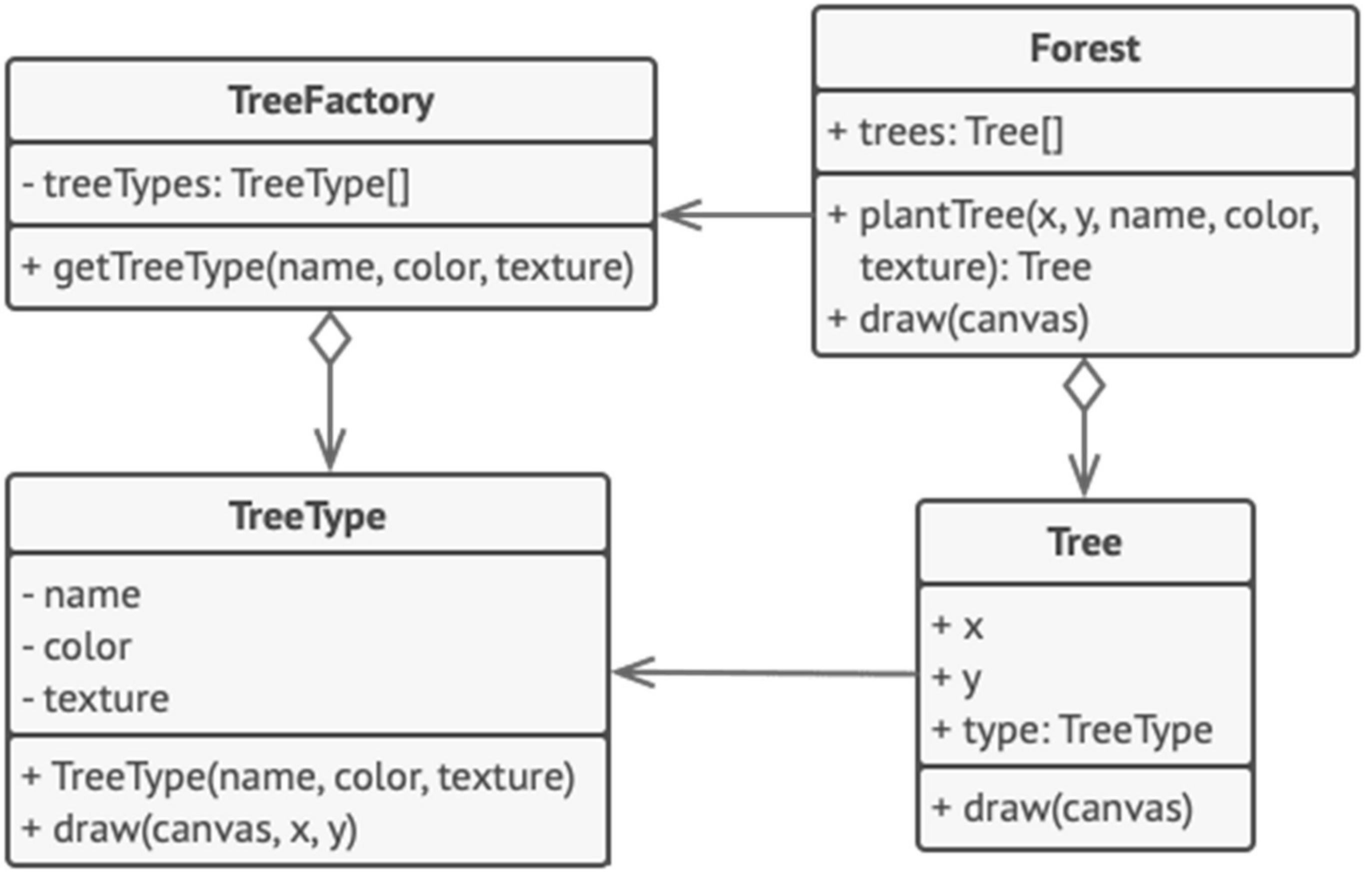
Class diagram of Flyweight



```
if (cache[repeatingState] == null) {
   cache[repeatingState] =
        new Flyweight(repeatingState)
}
return cache[repeatingState]
```

```
this.uniqueState = uniqueState
this.flyweight =
factory.getFlyweight(repeatingState)
```

Example code



```java
import java.awt.*;

public class Tree {
    private int x;
    private int y;
    private TreeType type;

    public Tree(int x, int y, TreeType type) {
        this.x = x;
        this.y = y;
        this.type = type;
    }

    public void draw(Graphics g) {
        type.draw(g, x, y);
    }
}
```

```java
import java.awt.*;

public class TreeType {
    private String name;
    private Color color;
    private String otherTreeData;

    public TreeType(String name, Color color, String otherTreeData) {
        this.name = name;
        this.color = color;
        this.otherTreeData = otherTreeData;
    }

    public void draw(Graphics g, int x, int y) {
        g.setColor(Color.BLACK);
        g.fillRect(x - 1, y, 3, 5);
        g.setColor(color);
        g.fillOval(x - 5, y - 10, 10, 10);
    }
}
```

```java
import java.awt.*;
import java.util.HashMap;
import java.util.Map;

public class TreeFactory {
    static Map<String, TreeType> treeTypes = new HashMap<>();

    public static TreeType getTreeType(String name, Color color, String otherTreeData) {
        TreeType result = treeTypes.get(name);
        if (result == null) {
            result = new TreeType(name, color, otherTreeData);
            treeTypes.put(name, result);
        }
        return result;
    }
}
```

```java
import javax.swing.*;
import java.awt.*;
import java.util.ArrayList;
import java.util.List;

public class Forest extends JFrame {
    private List<Tree> trees = new ArrayList<>();

    public void plantTree(int x, int y, String name, Color color, String otherTreeData)
        TreeType type = TreeFactory.getTreeType(name, color, otherTreeData);
        Tree tree = new Tree(x, y, type);
        trees.add(tree);
    }

    @Override
    public void paint(Graphics graphics) {
        for (Tree tree : trees) {
            tree.draw(graphics);
        }
    }
}
```

```java
import java.awt.*;

public class Demo {
    static int CANVAS_SIZE = 500;
    static int TREES_TO_DRAW = 1000000;
    static int TREE_TYPES = 2;

    public static void main(String[] args) {
        Forest forest = new Forest();
        for (int i = 0; i < Math.floor(TREES_TO_DRAW / TREE_TYPES); i++) {
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Summer Oak", Color.GREEN, "Oak texture stub");
            forest.plantTree(random(0, CANVAS_SIZE), random(0, CANVAS_SIZE),
                    "Autumn Oak", Color.ORANGE, "Autumn Oak texture stub");
        }
        forest.setSize(CANVAS_SIZE, CANVAS_SIZE);
        forest.setVisible(true);

        System.out.println(TREES_TO_DRAW + " trees drawn");
        System.out.println("---------------------");
        System.out.println("Memory usage:");
        System.out.println("Tree size (8 bytes) * " + TREES_TO_DRAW);
        System.out.println("+ TreeTypes size (~30 bytes) * " + TREE_TYPES + "");
        System.out.println("---------------------");
        System.out.println("Total: " + ((TREES_TO_DRAW * 8 + TREE_TYPES * 30) / 1024 / 1
                "MB (instead of " + ((TREES_TO_DRAW * 38) / 1024 / 1024) + "MB)");
    }

    private static int random(int min, int max) {
        return min + (int) (Math.random() * ((max - min) + 1));
    }
}
```

Demo code

- https://refactoring.guru/design-patterns/flyweight/java/example