Flyweight Pattern

Scenario:

Imagine you are developing a document editor that needs to manage and render a large number of text elements. Each character or word in the document might have associated properties like font, size, color, and style. Storing these properties individually for each character or word could lead to significant memory overhead. You want to optimize memory usage by sharing common properties among similar text elements.

Purpose:

The Flyweight Pattern is used to minimize memory usage by sharing as much data as possible with similar objects. Instead of creating a large number of similar objects, the Flyweight Pattern reuses existing instances of these objects. It is particularly useful in scenarios where a large number of objects are created, and many of them share common data.

When to Use:

- When you need to create a large number of similar objects and want to reduce memory consumption.
- When most of the object state can be shared among multiple instances.
- When you can distinguish between intrinsic (shared) and extrinsic (unique) properties of the objects.

Key Concepts:

- Flyweight: The shared object that contains intrinsic state. It is immutable and can be reused across different contexts.
- Flyweight Factory: Manages the creation and reuse of Flyweight objects. It ensures that Flyweight objects are shared whenever possible.
- Client: Uses Flyweight objects and supplies the extrinsic state (context-specific information) when needed.

How It Works:

1. Flyweight Interface: Defines the operations that can be performed on the Flyweight objects. It also

distinguishes between intrinsic and extrinsic state.

2. Concrete Flyweight: Implements the Flyweight interface and contains the intrinsic state that can be shared.

3. Flyweight Factory: Responsible for creating and managing Flyweight objects. It ensures that shared Flyweights are reused and new Flyweights are created only when necessary.

4. Client: Uses Flyweight objects, providing the extrinsic state when required.

Implementation Example:

Here is how the implementation might look in Java:

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

// Flyweight Interface
interface TextStyle {
    void applyStyle(String text);
}

// Concrete Flyweight
class ConcreteTextStyle implements TextStyle {
    private String font;
    private int size;
    private String color;

    public ConcreteTextStyle(String font, int size, String color) {
        this.font = font;
        this.size = size;
        this.color = color;
```

```
    }

    @Override
    public void applyStyle(String text) {
        System.out.println("Applying style: [Font: " + font + ", Size: " + size + ", Color: " + color + "] to text: " + text);
    }
}

// Flyweight Factory
class TextStyleFactory {
    private Map<String, TextStyle> styles = new HashMap<>();

    public TextStyle getStyle(String font, int size, String color) {
        String key = font + size + color;
        if (!styles.containsKey(key)) {
            styles.put(key, new ConcreteTextStyle(font, size, color));
        }
        return styles.get(key);
    }
}

// Client Code
public class DocumentEditor {
    public static void main(String[] args) {
        TextStyleFactory styleFactory = new TextStyleFactory();
```

```
        TextStyle style1 = styleFactory.getStyle("Arial", 12, "Black");
        TextStyle style2 = styleFactory.getStyle("Arial", 12, "Black");
        TextStyle style3 = styleFactory.getStyle("Times New Roman", 14, "Blue");

        style1.applyStyle("Hello, World!");
        style2.applyStyle("Flyweight Pattern Example");
        style3.applyStyle("Design Patterns in Java");

        System.out.println("Are style1 and style2 the same object? " + (style1 == style2)); // True
    }
}
```

Key Points to Remember:

- Flyweight Pattern optimizes memory usage by sharing common data among multiple objects. It reduces the number of objects created and minimizes memory usage by reusing existing objects whenever possible.
- Intrinsic vs. Extrinsic State: The Flyweight Pattern distinguishes between intrinsic state (shared, immutable data) and extrinsic state (context-specific, mutable data). Intrinsic state is stored in the Flyweight, while extrinsic state is provided by the client when needed.

Advantages:

- Memory Efficiency: The pattern significantly reduces memory usage by sharing common data among similar objects.
- Performance: Reduces the overhead associated with creating and managing a large number of objects.

Disadvantages:

- Complexity: The pattern can introduce additional complexity, especially when managing the extrinsic state.
- Potential Overhead: Managing the shared objects and ensuring they are reused appropriately can

add some overhead to the system.