Flyweight Design Pattern

Concept

The Flyweight pattern minimizes memory usage by sharing common parts of state between multiple objects instead of storing all data in each object. It separates intrinsic state (shared) from extrinsic state (unique to each object).

UML Class Diagram

```
┌─────────────────┐
│   Flyweight     │ (Interface)
├─────────────────┤
│ + operation()   │
└─────────────────┘
        △
        │
┌───────┴─────────┐
│ ConcreteFlyweight │
├─────────────────┤
│ - intrinsicState │
├─────────────────┤
│ + operation()     │
└─────────────────┘

┌─────────────────┐    ┌─────────────────┐
│ FlyweightFactory│◇───│   Flyweight     │
├─────────────────┤    │                 │
│ + getFlyweight()│    │                 │
└─────────────────┘    └─────────────────┘

┌─────────────────┐
│   Context       │◇───┐
├─────────────────┤    │
│ - extrinsicState│    │
│ - flyweight     │────┘
├─────────────────┤
│ + operation()   │
└─────────────────┘
```

Key Components

1. Flyweight Interface

- Declares methods that flyweights can use to receive and act on extrinsic state

2. Concrete Flyweight

- Implements flyweight interface
- Stores intrinsic state (shared among objects)
- Methods accept extrinsic state as parameters

3. Flyweight Factory

- Creates and manages flyweight objects
- Ensures flyweights are shared properly
- Returns existing flyweights or creates new ones

4. Context

- Contains extrinsic state (unique to each object)
- Maintains reference to flyweight object

State Types

Intrinsic State

- **Shared** among multiple objects
- **Immutable** - doesn't change
- Stored **inside** the flyweight
- Examples: Tree type, sprite image, font family

Extrinsic State

- **Unique** to each object
- **Changeable** - can be modified
- Stored **outside** the flyweight (in context)
- Passed to flyweight methods as parameters
- Examples: Position coordinates, color, size

Flow Diagram

```
Client Request → Factory → Check if flyweight exists
                │           │
                │           ▼
                │      ┌─────────┐    ┌──────────────┐
                │      │ Exists? │───▶│Return Existing│
                │      └─────────┘    └──────────────┘
                │           │
                ▼           ▼
        ┌──────────────┐ ┌──────────────┐
        │Create Context│ │Create New    │
        │with extrinsic│ │Flyweight     │
        │state         │ └──────────────┘
        └──────────────┘
```

Advantages

- **Memory Efficiency**: Dramatically reduces memory usage when dealing with large numbers of similar objects
- **Performance**: Fewer objects mean less memory allocation and garbage collection
- **Centralized Management**: Factory pattern provides centralized control over object creation
- **Scalability**: System can handle millions of objects efficiently
- **Resource Sharing**: Common resources are shared rather than duplicated

Disadvantages

- **Complexity**: Adds complexity by separating intrinsic and extrinsic state
- **CPU vs Memory Trade-off**: May increase CPU usage as extrinsic state is passed frequently
- **Design Constraints**: Objects must be designed to work with external state
- **Debugging**: Harder to debug due to state separation and object sharing
- **Thread Safety**: Shared flyweights must be thread-safe

When to Use Flyweight Pattern

Use When:

- Application needs to spawn huge number of similar objects
- Storage costs are high due to large quantity of objects
- Most object state can be made extrinsic
- Groups of objects can be replaced by few shared objects
- Application doesn't depend on object identity

Don't Use When:

- Application typically uses only small numbers of objects
- Storage costs are not a concern
- Objects contain mostly intrinsic state
- Object identity is important to the application

Common Use Cases

1. Text Editors

- Character objects sharing font, style information
- Position and actual character as extrinsic state

2. Game Development

- Bullets, particles, trees in a forest
- Sprite/texture shared, position/velocity extrinsic

3. Web Browsers

- DOM elements sharing styles
- CSS classes as flyweights, specific properties extrinsic

4. Graphics Applications

- Shapes with same properties
- Color, stroke as flyweight, coordinates extrinsic

5. Data Visualization

- Chart elements (bars, points, lines)
- Style information shared, data points extrinsic

Implementation Best Practices

1. Immutable Flyweights

java

```java
// Make flyweight immutable
public final class CharacterFlyweight {
    private final String font;
    private final int size;

    // No setters - immutable
}
```

2. Thread-Safe Factory

```java
public class FlyweightFactory {
    private static final ConcurrentHashMap<String, Flyweight> flyweights =
        new ConcurrentHashMap<>();
}
```

3. Clear State Separation

```java
// Intrinsic - shared, immutable
class TreeType {
    private final String species;
    private final Texture bark;
}

// Extrinsic - unique, passed as parameters
void render(int x, int y, Color seasonColor) { ... }
```

Memory Usage Example

Without Flyweight:

- 1,000,000 trees
- Each tree object: 100 bytes
- Total memory: ~100MB

With Flyweight:

- 1,000,000 contexts: 20 bytes each = ~20MB
- 10 flyweights: 50 bytes each = ~0.5KB
- Total memory: ~20MB (80% reduction!)

Sequence Diagram

```
Client → Factory: getFlyweight(key)
Factory → Factory: check cache
alt flyweight exists
    Factory → Client: return cached flyweight
else flyweight not found
    Factory → Factory: create new flyweight
    Factory → Factory: store in cache
    Factory → Client: return new flyweight
end

Client → Context: create with extrinsic state
Client → Context: operation()
Context → Flyweight: operation(extrinsicState)
```

Related Patterns

- **Singleton**: Factory often implements singleton pattern
- **Factory Method**: Used to create flyweight objects
- **Composite**: Flyweights can be used in composite structures
- **State/Strategy**: Can be implemented using flyweights

Summary

The Flyweight pattern is essential for memory-constrained applications dealing with large numbers of similar objects. By separating shared (intrinsic) state from unique (extrinsic) state, it achieves significant memory savings while maintaining object-oriented design principles.