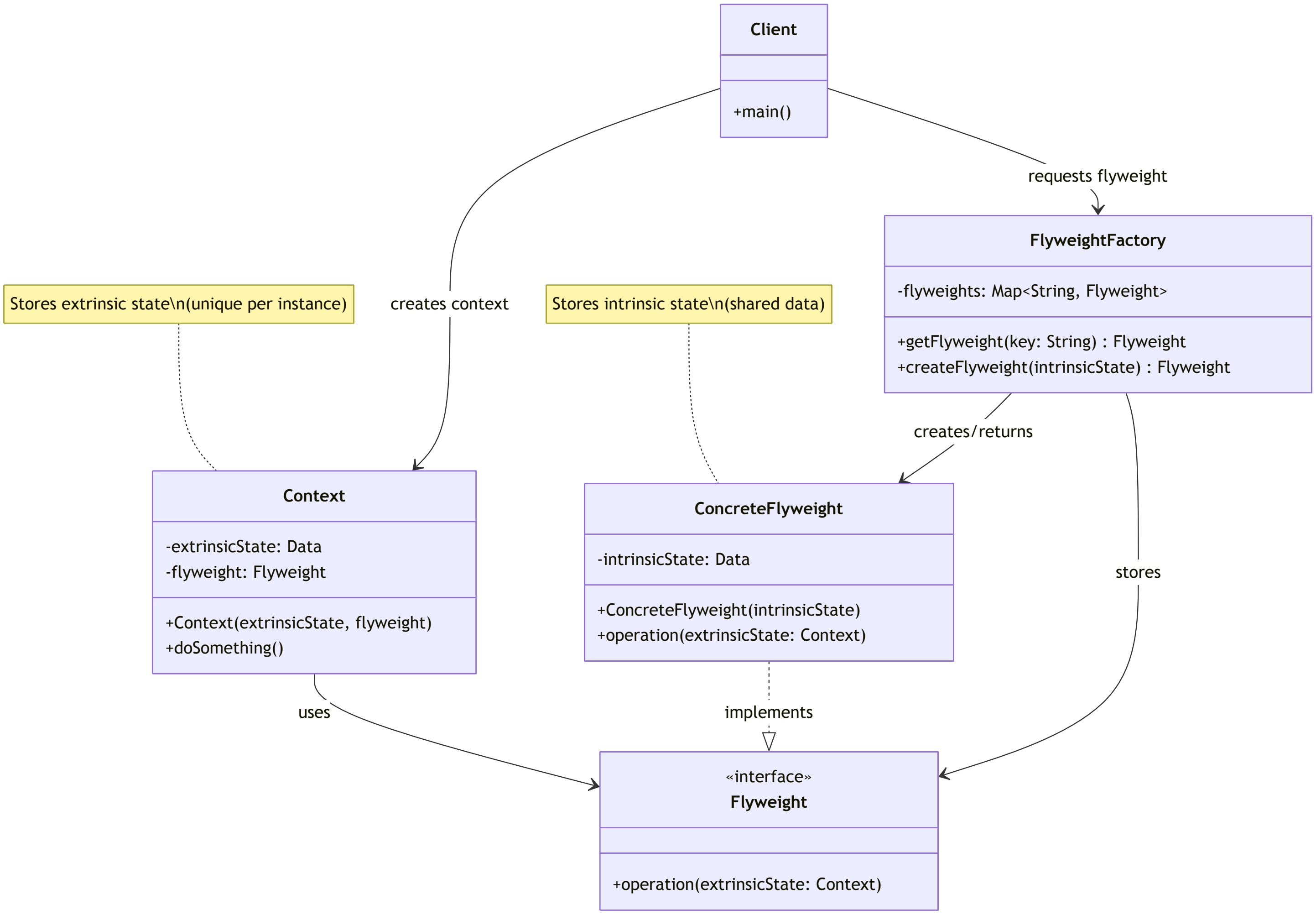


Generic Flyweight Pattern Implementation

Client
+main()
requests flyweight
creates context
FlyweightFactory
-flyweights: Map<String, Flyweight>
+getFlyweight(key: String) : Flyweight
+createFlyweight(intrinsicState) : Flyweight
Stores extrinsic state\n(unique per instance)
Stores intrinsic state\n(shared data)
creates/returns
Context
-extrinsicState: Data
-flyweight: Flyweight
+Context(extrinsicState, flyweight)
+doSomething()
ConcreteFlyweight
-intrinsicState: Data
+ConcreteFlyweight(intrinsicState)
+operation(extrinsicState: Context)
stores
uses
implements
«interface»
Flyweight
+operation(extrinsicState: Context)

Why "Flyweight" Pattern?

Etymology & Boxing Connection

The term "flyweight" comes from **boxing weight classes**, where flyweight is the **lightest category** (under 112 pounds/51kg). This metaphor perfectly captures the pattern's essence:

```
Boxing Flyweight = Minimal weight, maximum efficiency
Design Flyweight = Minimal memory, maximum sharing
```

The Naming Logic

Core Concept: Objects become "lightweight" by sharing heavy data

```
// Heavy object (heavyweight boxer)
class HeavyGraphic {
    private String content;     // 1KB
    private byte[] imageData;   // 5MB
    private RenderingRules rules; // 2MB
    private Position position;  // 8 bytes
}

// Flyweight object (flyweight boxer)
class GraphicFlyweight {
    private String content;     // 1KB (shared)
    private byte[] imageData;   // 5MB (shared)
    private RenderingRules rules; // 2MB (shared)
    // Position stored externally – only 8 bytes per instance
}
```

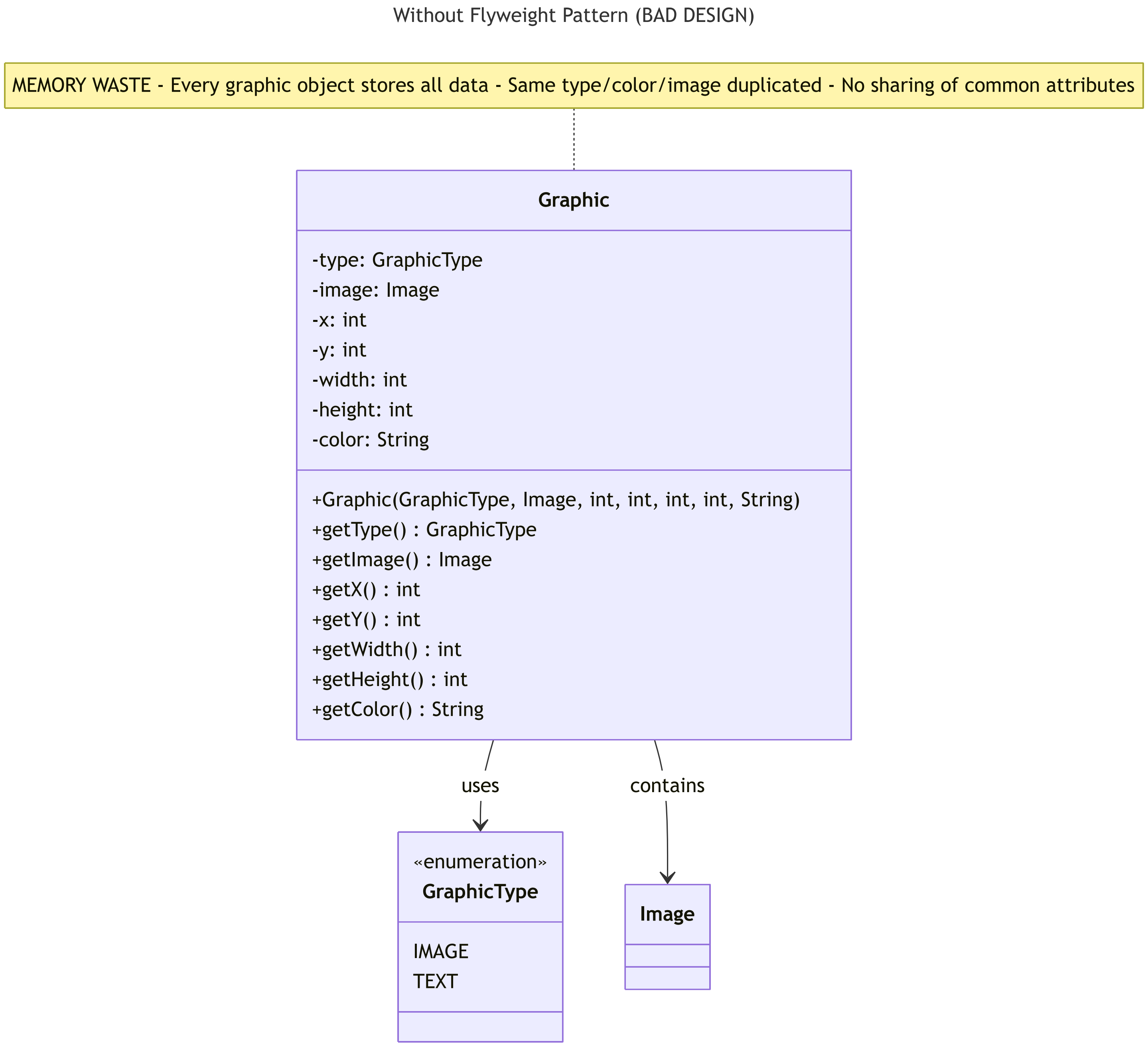
Memory Weight Comparison

Scenario	Without Flyweight	With Flyweight
1000 text objects	1000 × 7MB = **7GB**	1 × 7MB + 1000 × 8B = **7MB**
Memory "weight"	Heavy (7GB)	Flyweight (7MB)
Reduction	-	**99.9% lighter**

> **Important Note**: The pattern makes objects "lightweight" in memory consumption, not functionality. Full capabilities are preserved through the intrinsic/extrinsic state separation.

Flyweight Pattern for Graphic Elements

Without Flyweight Pattern



```
// BAD - Multiple objects with duplicate data
public class GraphicsEditor {
    public void createDocument() {
        // Multiple text graphics with same style - wasteful! ❌
        Graphic text1 = new Graphic(GraphicType.TEXT, textImage, 10, 20, 100, 30, "red"
        Graphic text2 = new Graphic(GraphicType.TEXT, textImage, 50, 60, 150, 25, "red"
        Graphic text3 = new Graphic(GraphicType.TEXT, textImage, 80, 90, 200, 35, "red"

        // Each object stores complete state independently ❌
        // type="TEXT", image=textImage, color="red" repeated 3 times
        // Memory: 3 complete objects = 3x memory usage
    }
}
```

SOLID Principles Violated Without Flyweight Pattern

🔴 Single Responsibility Principle (SRP)

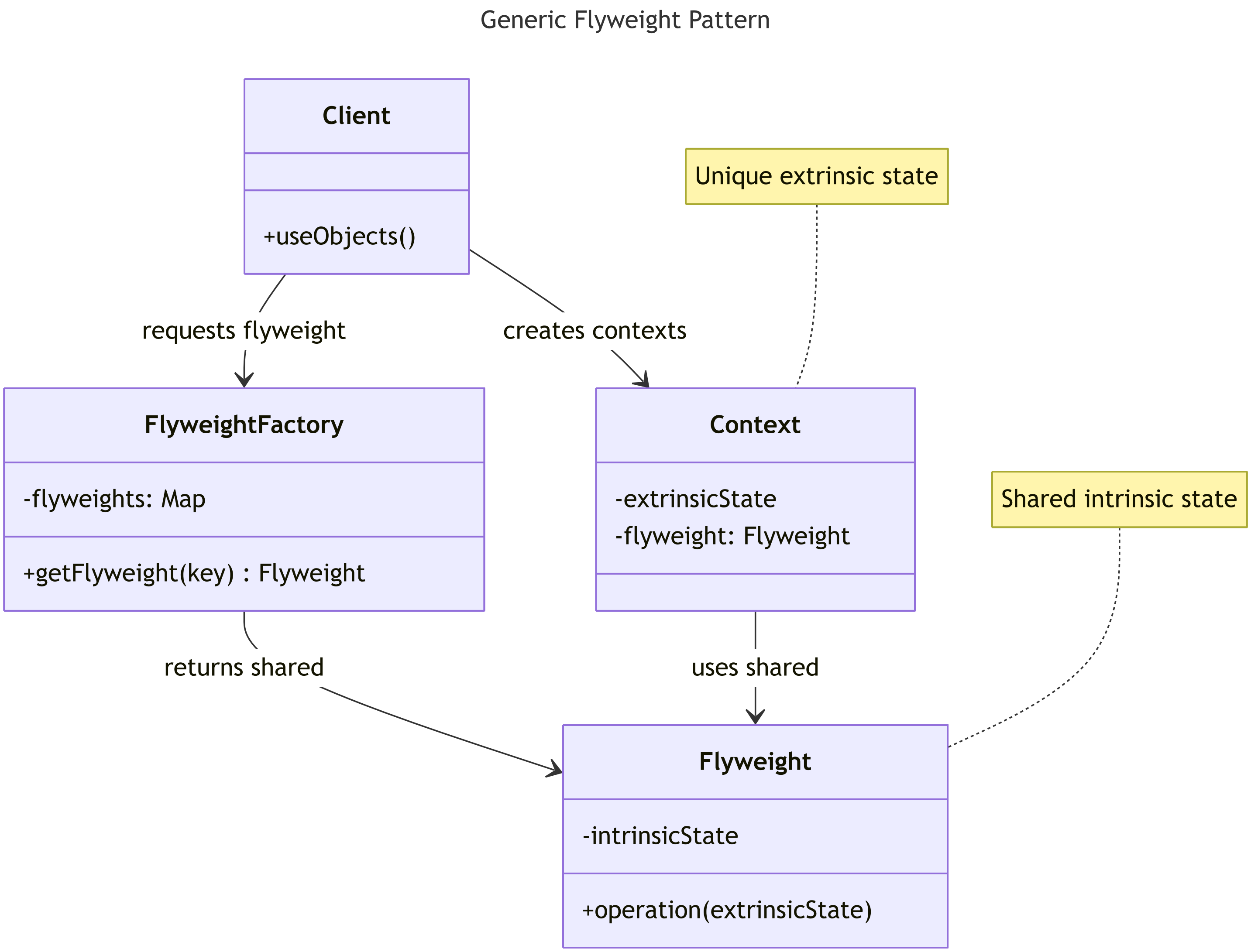
- Graphic class handles both shared attributes and positioning
- Mixed concerns: content management + spatial positioning

🔴 Open/Closed Principle (OCP)

- Adding new shared attributes requires modifying all graphic instances
- No separation between shared and instance-specific data

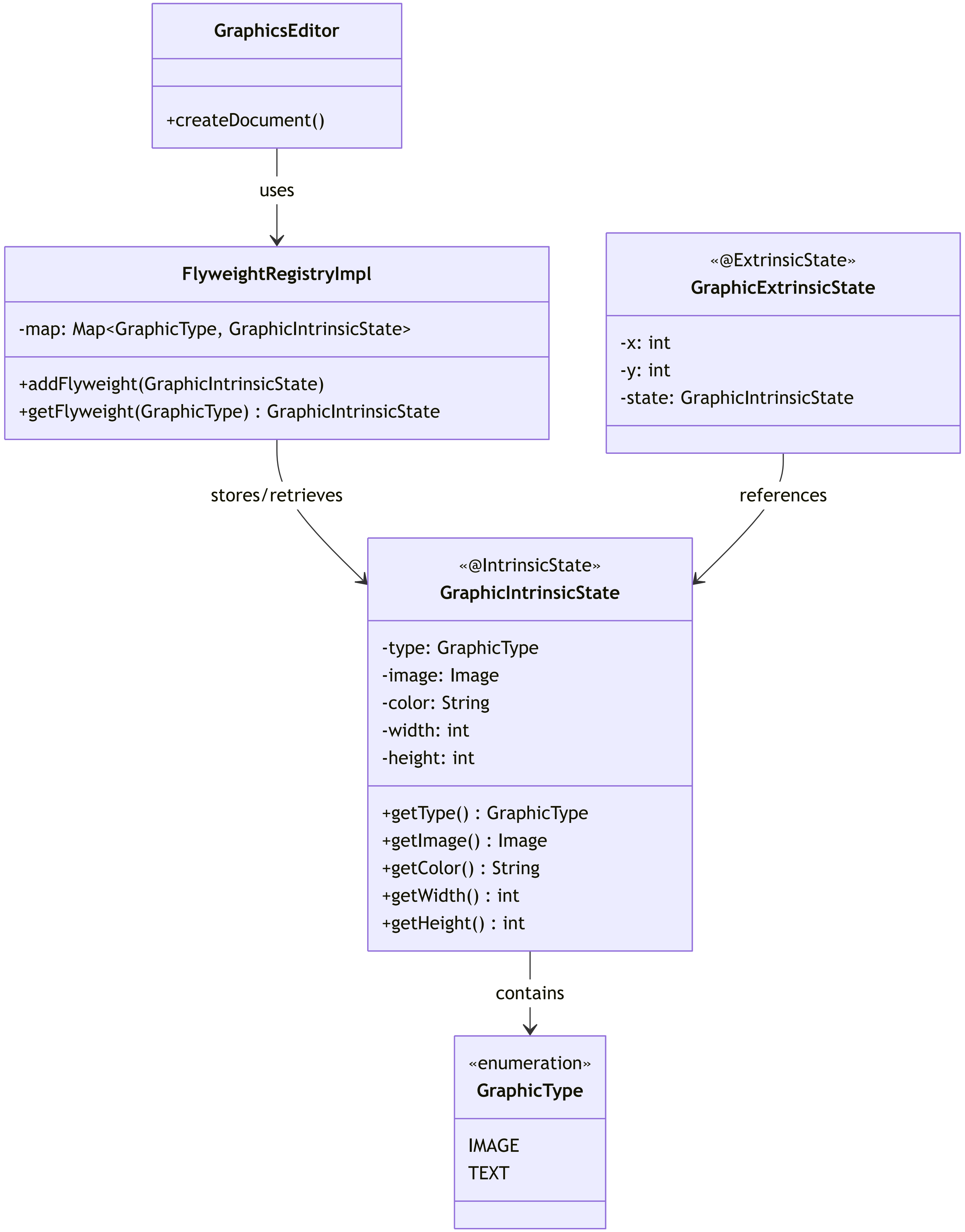
With Flyweight Pattern

Generic Diagram



Specific Diagram

Graphic Elements Flyweight Implementation



How Flyweight Pattern Helps Here

Key Benefits:

- **Memory Optimization**: Shared intrinsic state across multiple graphics of same type
- **Reduced Object Creation**: Only one intrinsic state per graphic type/style combination
- **State Separation**: Clear distinction between shared (intrinsic) and unique (extrinsic) data
- **Registry Management**: Centralized storage and retrieval of flyweight objects
- **Scalability**: Memory usage grows with unique types, not total instances

Usage Examples

```
// Create shared intrinsic states
GraphicIntrinsicState textState = new GraphicIntrinsicState(GraphicType.TEXT, image, "r
GraphicIntrinsicState imageState = new GraphicIntrinsicState(GraphicType.IMAGE, photo,

// Store in registry
registry.addFlyweight(textState);
registry.addFlyweight(imageState);

// Create multiple graphics with shared states
GraphicExtrinsicState text1 = new GraphicExtrinsicState(10, 20, textState);
GraphicExtrinsicState text2 = new GraphicExtrinsicState(50, 60, textState);
GraphicExtrinsicState image1 = new GraphicExtrinsicState(0, 0, imageState);

// Memory saved: Instead of 3 complete objects, only 2 flyweights + 3 contexts
```

Memory Optimization Example

For 100 text graphics with same style:

- **Without Flyweight**: 100 complete objects
- **With Flyweight**: 1 shared intrinsic state + 100 lightweight position contexts
- **Memory Savings**: ~80-90% reduction in shared data storage

The pattern efficiently handles scenarios where many objects share common attributes while maintaining individual positioning and context data.

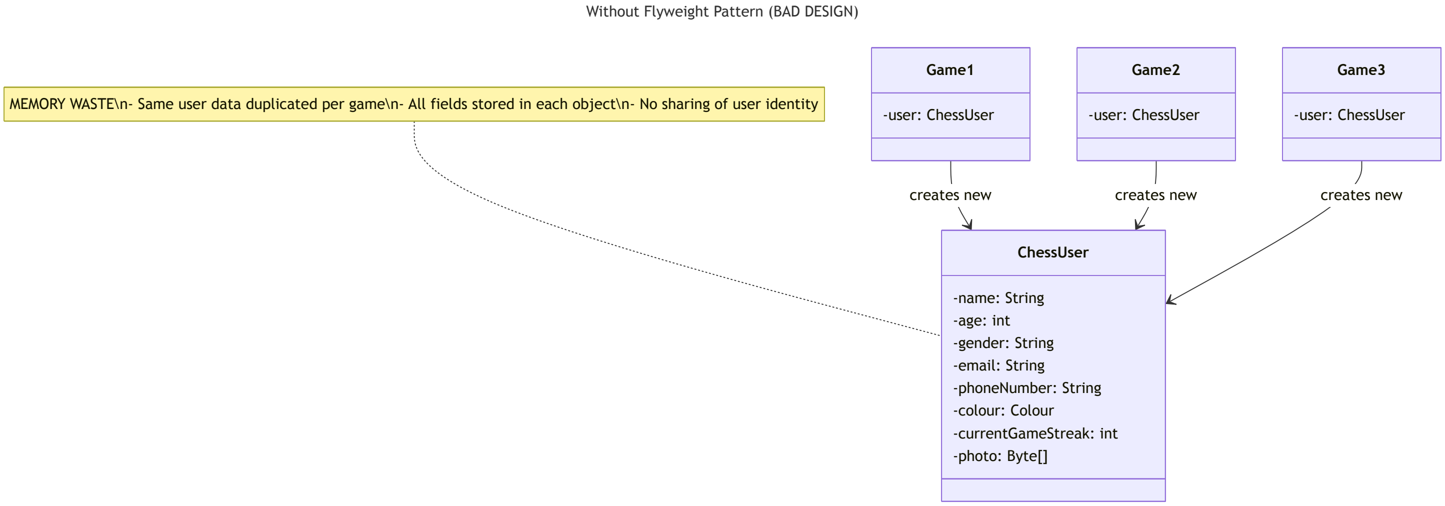
Implementation Caveats & Improvements

Current Implementation Issues:

1. **Width/Height as Intrinsic**: May limit flexibility if graphics of same type need different sizes
   - **Hint**: Consider moving to extrinsic state for variable dimensions
2. **Missing Constructors**: Classes lack proper constructors for object creation
   - **Hint**: Add `@AllArgsConstructor` or manual constructors
3. **No Getters in Extrinsic**: Missing access methods for coordinate data
   - **Hint**: Add `@Getter @Setter` annotations
4. **Basic Registry**: Simple map-based lookup without error handling
   - **Hint**: Add null checks and helper methods for robustness
5. **Single Flyweight per Type**: Current registry stores only one flyweight per GraphicType
   - **Hint**: Consider composite keys (type + color + image) for finer granularity

Flyweight Pattern for Chess User Optimization

Without Flyweight Pattern

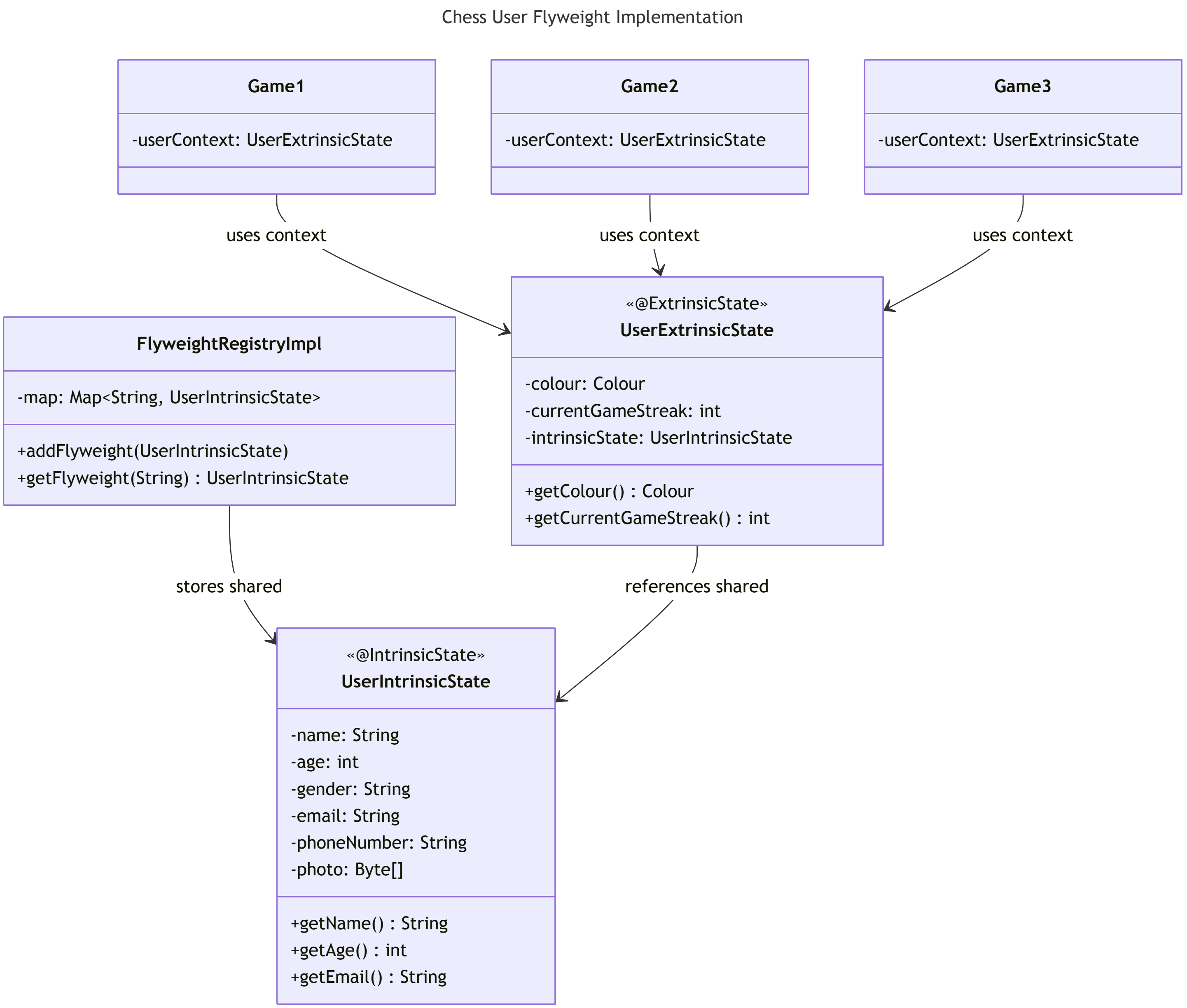


```java
// BAD - Same user, multiple complete objects
public class ChessApplication {
    public void startGames() {
        // Alice plays 3 games - creates 3 complete user objects ❌
        ChessUser alice1 = new ChessUser("Alice", 25, "Female", "alice@email.com",
                                        "123-456-7890", Colour.WHITE, 5, photo);
        ChessUser alice2 = new ChessUser("Alice", 25, "Female", "alice@email.com",
                                        "123-456-7890", Colour.BLACK, 3, photo);
        ChessUser alice3 = new ChessUser("Alice", 25, "Female", "alice@email.com",
                                        "123-456-7890", Colour.WHITE, 7, photo);

        // User identity data (name, age, email, photo) duplicated 3 times ❌
    }
}
```

With Flyweight Pattern

Specific Diagram



How Flyweight Pattern Helps Here

Key Benefits:

- **Memory Optimization**: User identity data shared across multiple games
- **State Separation**: Fixed user data vs variable game data clearly separated
- **Registry Management**: Centralized storage using email as unique key

- **Context Flexibility**: Each game has independent colour and streak data

State Classification

Intrinsic (Shared): `name, age, gender, email, phoneNumber, photo`

- User identity that never changes

Extrinsic (Variable): `colour, currentGameStreak`

- Game-specific data that varies per session

Usage Examples

```
// Alice plays 10 games
UserIntrinsicState alice = registry.getFlyweight("alice@email.com");

// Each game has different context
UserExtrinsicState[] games = {
    new UserExtrinsicState(Colour.WHITE, 5, alice),
    new UserExtrinsicState(Colour.BLACK, 3, alice),
    new UserExtrinsicState(Colour.WHITE, 8, alice),
    // ... 7 more games
};

// Memory: 1 shared Alice identity + 10 lightweight contexts
// vs 10 complete ChessUser objects
```

Memory Benefits

Same user playing multiple games:

- **Without Flyweight**: N complete user objects
- **With Flyweight**: 1 shared intrinsic state + N lightweight contexts
- **Memory Savings**: ~85% reduction in user identity data duplication

The pattern optimizes chess applications where users participate in multiple simultaneous games, eliminating redundant storage of unchanging user identity information.