Week 09: Design Patterns in Game Development

Game Loop Pattern & Singleton Pattern

Object-Oriented Programming Course

Dungeon Escape: Progressive Learning

November 2, 2025

Outline

1. Introduction
2. Branch 09-00: The Problem
3. Branch 09-01: Game Loop Solution
4. Branch 09-02: New Challenges
5. Branch 09-03: Singleton Solution
6. Comparative Analysis
7. Design Patterns Deep Dive
8. Discussion Points
9. Assessment
10. Summary

Week 09 Overview

Learning Objectives

- Understand the **Game Loop Pattern**
- Learn separation of update and rendering
- Identify the **Object Drilling** anti-pattern
- Implement the **Singleton Pattern**
- Compare architectural trade-offs

Progressive Branches

- 09-00: Monolithic design (the problem)
- 09-01: Game Loop pattern (first solution)
- 09-02: Without Singleton (new problem)
- 09-03: With Singleton (final solution)

Branch 09-00: Monolithic Design

What is it?

- All code in one giant `main()` method
- 150+ lines in single method
- Update logic mixed with rendering
- No separation of concerns

Problems Demonstrated:

- Frame rate coupling
- Untestable code
- Poor maintainability
- No scalability

Key Issue:

Frame Rate Coupling

Rendering delays slow down game logic by 80%!

- Render takes 50ms (flickering)
- Only 2 FPS achieved
- Game logic blocked by rendering

Branch 09-00: Code Structure

```java
public class Main {
    public static void main(String[] args) {
        // Initialize
        NPC npc = new NPC();
        Coin coin = new Coin();

        while (running) {
            // Update logic
            npc.move();
            coin.fall();
            checkCollisions();

            // Render (SLOW - causes problems!)
            clearScreen();
            draw(npc);
            draw(coin);
            Thread.sleep(50); // Flickering!
        }
    }
```

Branch 09-00: Performance Impact

Performance Metrics

Metric	Value
Lines of Code (Main.java)	150+
Frames Per Second	2 FPS
Testability	0%
Maintainability	Very Low

Critical Issue

Cannot unit test logic without triggering rendering!

Branch 09-01: Game Loop Pattern

Solution: Separation of Concerns

Split monolithic code into specialized classes:

- `GameEngine`: Controls the game loop
- `GameLogic`: Updates game state
- `GridRenderer`: Handles rendering only

Key Concepts:

- `update()` - Logic only
- `draw()` - Rendering only
- Delta time (Δt)
- Frame rate independence

Benefits:

- Testable (no display needed)
- 60 FPS performance
- Clean separation
- Predictable behavior

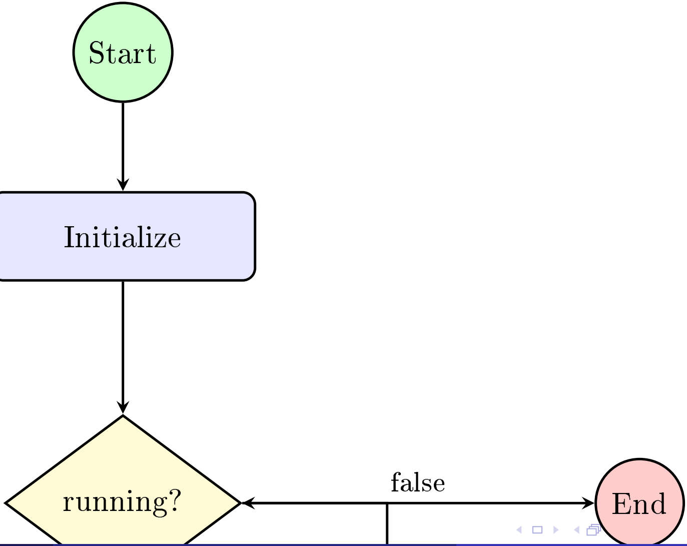
Branch 09-01: Game Loop Structure

```java
public class GameEngine {
    private GameLogic logic;
    private boolean running = true;

    public void start() {
        while (running) {
            float delta = calculateDeltaTime();

            update(delta);  // Logic only
            draw();         // Render only
            sync();         // Control FPS (60 target)
        }
    }
}
```

Branch 09-01: Game Loop Flow Diagram



Branch 09-01: Performance Improvement

09-00 vs 09-01 Comparison

Metric	09-00	09-01	Change
Lines in Main	150+	3	50x reduction
FPS	2	60	30x improvement
Testability	0%	100%	Perfect
Flickering	Yes	No	Fixed

Achievement Unlocked

Clean, testable, professional 60 FPS architecture!

Branch 09-02: Expanding the Game

New Requirement

Add a HUD (Heads-Up Display) to show:

- Current score
- Game time
- Player level

Design Challenge

Multiple classes need to access the `GameManager`:

- `GameLogic` needs it to update score
- `HUD` needs it to display score
- `NPC` needs it to check game state
- `Coin` needs it to add points

Branch 09-02: Object Drilling Problem

The Anti-Pattern:

- Pass `manager` through constructors
- 4 levels deep!
- `Main` → `Engine` → `Logic` → `NPC`
- Every class polluted with parameters

Consequences:

- 6+ files affected by changes
- Team collaboration conflicts
- Refactoring nightmare
- Constructor pollution

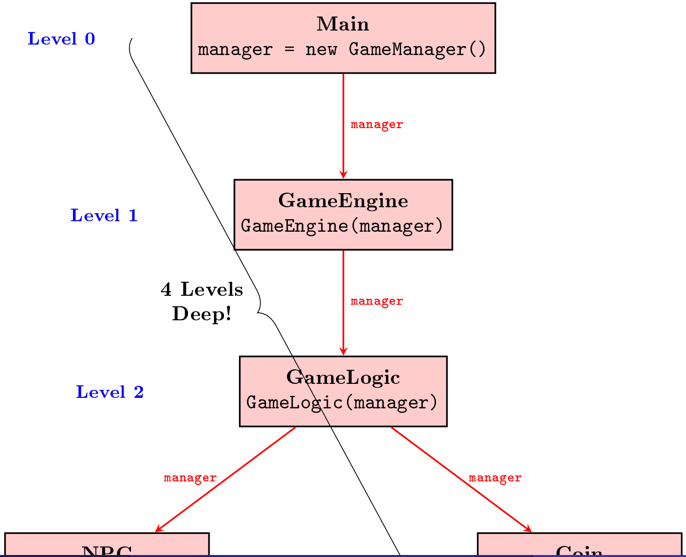
Critical Bug!

`HUD` creates its own `GameManager` instance instead of using the shared one!

Result:

- Score updates in instance A
- HUD displays from instance B
- Score shows 0 forever!

Branch 09-02: Object Drilling Visualization



Level 0

Main
manager = new GameManager()

manager

Level 1

GameEngine
GameEngine(manager)

4 Levels
Deep!

manager

Level 2

GameLogic
GameLogic(manager)

manager

manager

NPC

Coin

Branch 09-02: The Bug

```java
public class HUD {
    // BUG: Creates NEW instance!
    private final GameManager manager = new GameManager();

    public HUD(GameManager passedManager) {
        // Intentionally ignore the parameter!
        System.out.println("Using own instance!");
    }

    public void draw() {
        // Reads from WRONG instance!
        int score = manager.getScore();  // Always 0!
        System.out.println("Score: " + score);
    }
}
```

Output

```
[GameManager:498931366] Score updated:  10
```

Branch 09-03: Singleton Pattern

Solution: Guarantee Single Instance

The Singleton pattern ensures a class has only ONE instance and provides global access to it.

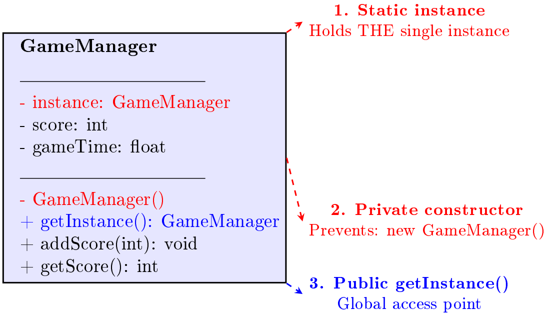
Three Key Components:

1. Private static instance
2. Private constructor
3. Public static getInstance()

Benefits:

- Zero constructor parameters
- Guaranteed single instance
- Global access point
- Easy refactoring

Branch 09-03: Singleton Pattern Diagram



GameManager

- instance: GameManager
- score: int
- gameTime: float

- GameManager()
+ getInstance(): GameManager
+ addScore(int): void
+ getScore(): int

1. Static instance
Holds THE single instance

2. Private constructor
Prevents: new GameManager()

3. Public getInstance()
Global access point

Implementation:

```
public class GameManager {
    private static GameManager instance = null;

    private GameManager() { /* ... */ }

    public static GameManager getInstance() {
        if (instance == null) {
            instance = new GameManager();
        }
        return instance;
```

Usage:

```
// X Compiler error!
GameManager m = new GameManager();

// OK Correct way:
GameManager mgr = GameManager.getInstance();
mgr.addScore(10);
```

Branch 09-03: Implementation

```java
public class GameManager {
    // 1. Static instance (lazy initialization)
    private static GameManager instance = null;

    // 2. Private constructor (prevents: new GameManager())
    private GameManager() {
        this.score = 0;
        this.gameTime = 0.0f;
        this.level = 1;
    }

    // 3. Global access point
    public static GameManager getInstance() {
        if (instance == null) {
            instance = new GameManager();
        }
        return instance;
    }
}
```

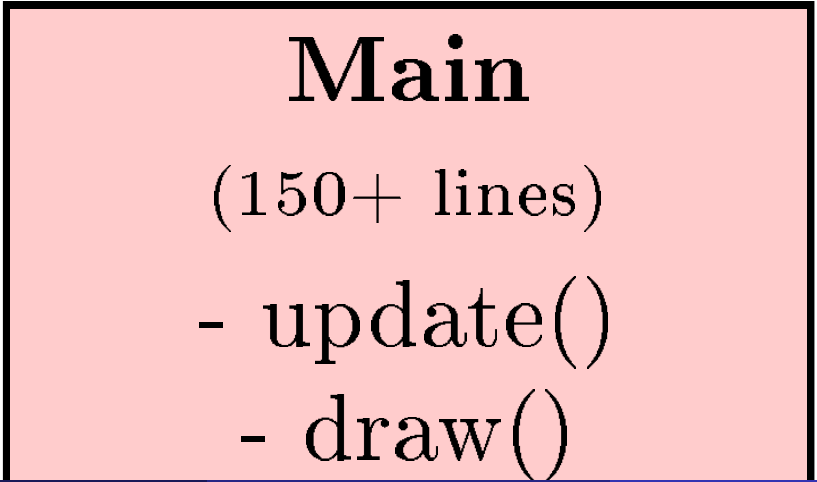
Branch 09-03: Clean Usage

```java
// Main.java - No parameters!
public class Main {
    public static void main(String[] args) {
        GameEngine engine = new GameEngine();
        engine.start();
    }
}

// HUD.java - Direct access!
public class HUD {
    public HUD() {
        // No parameters needed!
    }

    public void draw() {
        // Guaranteed to be THE instance
        int score = GameManager.getInstance().getScore();
        System.out.println("Score: " + score);
    }
```

Architecture Evolution

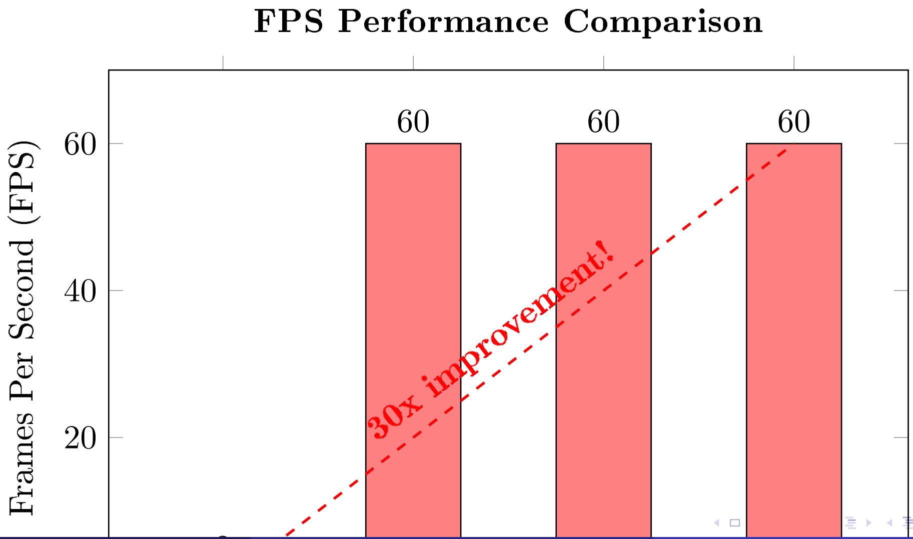


Main

(150+ lines)

- update()
- draw()

Performance Comparison



Comprehensive Metrics

Metric	09-00	09-01	09-02	09-03
Lines in Main	150+	3	32	3
FPS	2	60	60	60
Testability	0%	100%	100%	100%
Constructor Params	0	0	6	0
GameManager Instances	0	0	2 (BUG)	1
Object Drilling Depth	N/A	N/A	4 levels	0

Final Achievement

Clean architecture with 60 FPS, zero object drilling, and guaranteed single instance!

Game Loop Pattern

Intent

Decouple the progression of game time from user input and processor speed.

Structure:

- `update(deltaTime)`
- `draw()`
- `sync()`

Participants:

- GameEngine
- GameLogic
- Renderer

Consequences:

Benefits:

- Frame-rate independence
- Testability
- Clear separation

Liabilities:

- More classes
- Initial complexity

Singleton Pattern

Intent

Ensure a class has only one instance and provide a global point of access to it.

When to Use:

- Shared resource management
- Global state needed
- Exactly one instance required

Benefits:

- Controlled access
- No global variables
- Lazy initialization

Liabilities:

- Global state (testing harder)
- Hidden dependencies
- Thread safety concerns

Alternatives:

- Dependency Injection
- Service Locator
- Static Class

Discussion Questions

For Students:

1. Why is frame rate coupling a critical problem in games?
2. What are the trade-offs of the Singleton pattern?
3. When would you NOT use a Singleton?
4. How does delta time enable frame-rate independence?
5. What alternative to Singleton could we use?

Critical Thinking:

- Is global state always bad?
- How would you test a class that uses GameManager.getInstance()?
- What happens in a multi-threaded environment?

Assessment Rubric (100 points)

Component	**Points**	**Criteria**
Code Implementation	40	Correct Singleton, working game loop
Testing	20	Unit tests for GameLogic, coverage > 80%
Design	20	UML diagrams, architecture explanation
Documentation	10	JavaDoc, README, design decisions
Code Quality	10	Style, no warnings, clean code
Total	**100**	

Week 09 Summary

Key Takeaways

- **Game Loop Pattern**: Separates update from rendering
- **Delta Time**: Enables frame-rate independence
- **Object Drilling**: Anti-pattern to avoid
- **Singleton Pattern**: Guarantees single instance
- **Trade-offs**: Every pattern has benefits and costs

Progressive Learning Journey

09-00 (Problem) → 09-01 (Solution) → 09-02 (New Problem) → 09-03 (Final Solution)

Result

Professional game architecture: 60 FPS, testable, maintainable, scalable!

Thank You!

Questions? Comments?

Next Week: Observer Pattern & Event Systems