Pemrograman Berorientasi Objek dan Praktik-ENCE603010 (3 SKS)

Factory Method & Object Pool Pattern

Tim Dosen:
I Gde Dharma Nugraha, S.T., M.T., Ph.D.
Muhammad Firdaus Syawaludin Lubis, Ph.D.

Semester Gasal 2025 – 2026

Introduction

Introduction | Week 10 Overview

Topic: Creational Design Patterns & Performance Optimization

Two Key Patterns:

1. **Factory Method Pattern** - Flexible object creation
2. **Object Pool Pattern** - Performance optimization

Four Branches

10-01 → 10-02 → 10-03 → 10-04 (Problem → Solution → Problem → Solution)

Introduction | What We'll Build

Dungeon Escape Game Features:

- Dynamic obstacle spawning
- Three obstacle types (Spike, Goblin, Wolf)
- Collision detection
- Performance monitoring
- Safe spawning algorithm
- Object pooling for efficiency

Learning Outcomes:

1. Understand extensibility problems
2. Apply Factory Method pattern
3. Identify performance bottlenecks
4. Implement Object Pool pattern

Branch 10-01: The Problem

Branch 10-01: The Problem | 10-01: Hard-Coded Object Creation

Problem: Switch statement for object creation
[language=Java,basicstyle=] `// WorldController.java - ANTI-PATTERN private void spawnRandomObstacle()  int type = random.nextInt(3); int x = random.nextInt(25); int y = random.nextInt(25);`

`Obstacle newObstacle = null; switch (type)  case 0:  newObstacle = new Spike(x, y); break; case 1:  newObstacle = new Goblin(x,`

`y); break; case 2:  newObstacle = new Wolf(x, y); break;  activeObstacles.add(newObstacle);`

Issues:

- Tight coupling to concrete classes
- Violates Open/Closed Principle
- Hard to add new obstacle types
- Causes merge conflicts

Branch 10-01: The Problem | 10-01: Why This is Bad

Scenario: Team wants to add new obstacle types

What happens:

1. Developer A adds `case 3: new Dragon()`
2. Developer B adds `case 3: new Trap()`
3. Developer C adds `case 3: new Boulder()`
4. **Merge conflict!** All editing same switch statement

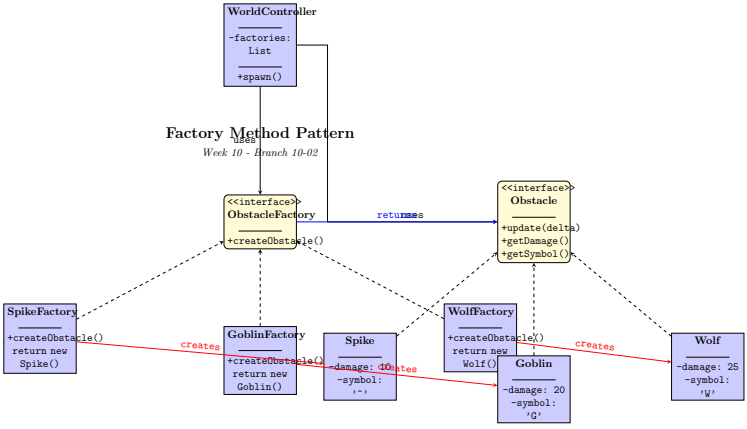
Open/Closed Principle Violated

Classes should be **open for extension** but **closed for modification**.
Switch statement forces modification of existing code.

Branch 10-02: Factory Method Solution

Branch 10-02: Factory Method Solution | 10-02: Factory Method Pattern

Solution: Delegate object creation to factory classes



Branch 10-02: Factory Method Solution | 10-02: Factory Implementation

Factory Interface: [language=Java,basicstyle=] public interface ObstacleFactory Obstacle createObstacle(int x, int y);

Concrete Factory: [language=Java,basicstyle=] public class GoblinFactory implements ObstacleFactory @Override public Obstacle createObstacle(int x, int y) return new Goblin(x, y);

Usage in WorldController: [language=Java,basicstyle=] private List<ObstacleFactory> factories; private void spawnRandomObstacle() ObstacleFactory factory = factories.get(random.nextInt(factories.size())); Obstacle obstacle = factory.createObstacle(x, y); activeObstacles.add(obstacle);

Branch 10-02: Factory Method Solution | 10-02: Benefits

Advantages of Factory Method:

1. **Loose coupling** - WorldController doesn't know concrete types
2. **Open/Closed** - Add new obstacles without modifying existing code
3. **No conflicts** - Each developer creates their own factory
4. **Easy testing** - Mock factories for unit tests

Adding New Obstacle

To add `Dragon`:

1. Create `Dragon.java` class
2. Create `DragonFactory.java`
3. Register factory in WorldController constructor
4. Done! No switch statement modification

Branch 10-03: Performance Problem

Branch 10-03: Performance Problem | 10-03: The GC Problem

New Requirement: Spawn 20 obstacles per second

What happens:

- 20 spawns/second × 50 seconds = 1000+ object creations
- Objects go off-screen and get garbage collected
- **Garbage Collection pauses freeze the game!**

Performance Impact

Branch 10-03: 1000+ allocations, 9ms GC time
In larger games: Can cause frame drops, stuttering, poor UX

Branch 10-03: Performance Problem | 10-03: GC Monitoring

Performance Monitor Implementation:

Metrics Tracked:

- Frame time (milliseconds)
- Average frame time
- Worst frame time
- Slow frames (< 30 FPS)
- GC pause count
- Total GC time

Uses: `GarbageCollectorMXBean` API to track GC events

Console Output:

[language=,basicstyle=,frame=single]

```
+==================+ | PERFORMANCE | +==================+ |
Frame:  60 | | Avg:  1.2ms | | Worst:  18.3ms | | Slow:  0
(0.0+==================+
```

Branch 10-03: Performance Problem | 10-03: Safe Spawning Algorithm

Challenge: Don't spawn obstacles on top of NPC or walls

[language=Java,basicstyle=] private boolean isSafePosition(int x, int y) // Check 1: Must be walkable floor if (!DungeonMap.isWalkable(x, y)) return false;
// Check 2: Must not be too close to NPC (minimum 3 tiles) int distance = Math.abs(x - npc.getX()) + Math.abs(y - npc.getY()); if (distance < 3) return false;

// Check 3: Must not overlap with existing obstacles for (Obstacle obs : activeObstacles) if (obs.getX() == x obs.getY() == y)

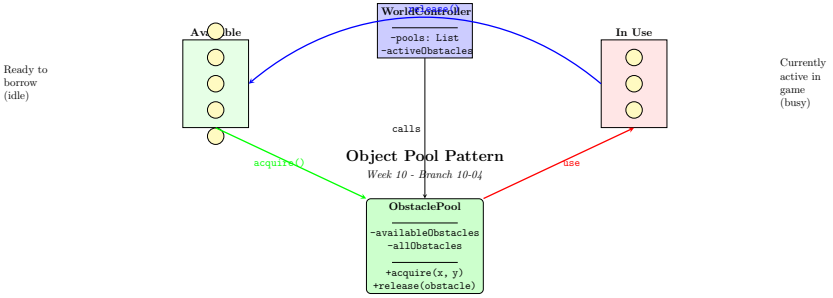
return false; return true;

Algorithm: Try max 10 times to find safe position, give up if not found

Branch 10-04: Object Pool Solution

Branch 10-04: Object Pool Solution | 10-04: Object Pool Pattern

Idea: Reuse objects instead of creating new ones



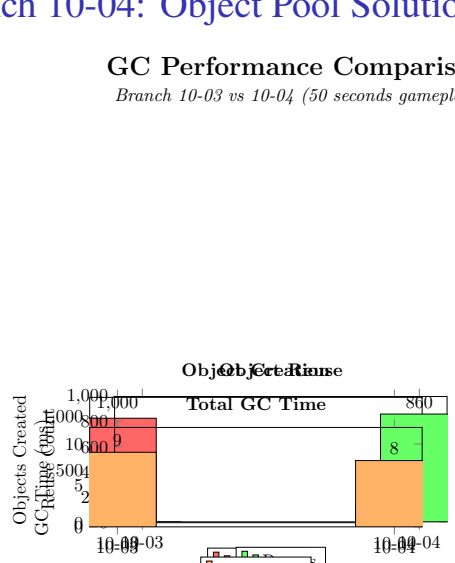
Object Lifecycle:
1. Startup: Pre-allocate
2. Acquire: Remove from available
3. Reset: Clean state

Branch 10-04: Object Pool Solution | 10-04: Pool Implementation

[language=Java,basicstyle=] public class ObstaclePool private final List<Obstacle> availableObstacles; private final List<Obstacle> allObstacles; private final ObstacleFactory factory;
public ObstaclePool(ObstacleFactory factory, int initialSize) this.availableObstacles = new ArrayList<>(); this.allObstacles = new ArrayList<>();
// Pre-allocate at startup for (int i = 0; i < initialSize; i++) Obstacle obstacle = factory.createObstacle(0, 0); obstacle.setActive(false); availableObstacles.add(obstacle); allObstacles.add(obstacle);
public Obstacle acquire(int x, int y) if (!availableObstacles.isEmpty()) Obstacle obstacle = availableObstacles.remove(availableObstacles.size() - 1); obstacle.reset(x, y); obstacle.setActive(true); return obstacle; return null; // Pool exhausted

public void release(Obstacle obstacle) obstacle.setActive(false); availableObstacles.add(obstacle);

Branch 10-04: Object Pool Solution | 10-04: Performance Comparison



Memory Pattern:

10-03 (No Pool):
Continuous allocate/free cycles
→ GC pressure increases over time

10-04 (With Pool):
One-time allocation at startup
→ Zero GC pressure during gameplay

Branch 10-04: Object Pool Solution | 10-04: Pool Statistics

Branch 10-04 tracks pool usage:

[language=,basicstyle=,frame=single] === POOL STATISTICS === Spike Pool Stats: Total Created: 50 Acquire Calls: 334 Release Calls: 284 Total Size: 50 Available: 0 In Use: 50

Goblin Pool Stats: Total Created: 50 Acquire Calls: 334 Release Calls: 284 ...

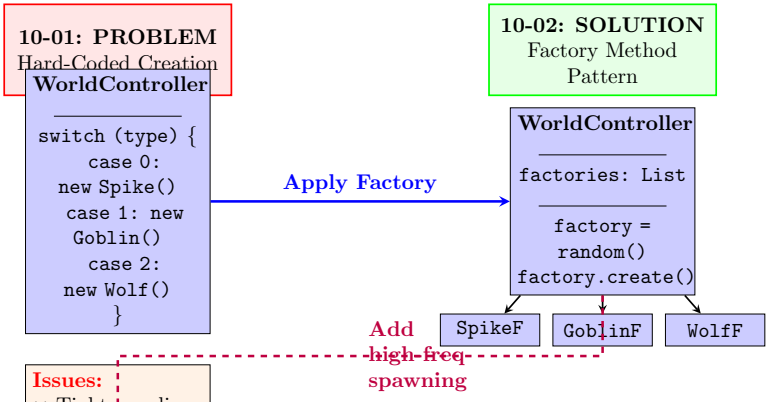
Analysis:

- 50 created (at startup)
- 334 acquires = 334 times spawned
- 284 releases = 284 returned
- **Result: 284 reuses! (85% reuse rate)**

Design Evolution

Design Evolution | Architecture Evolution

Architecture Evolution: Week 10



Design Evolution | Key Takeaways

Factory Method Pattern:

- Solves: Tight coupling, extensibility problems
- Benefits: Loose coupling, Open/Closed, no merge conflicts
- Trade-off: More classes (but worth it!)

Object Pool Pattern:

- Solves: GC pressure from frequent allocation
- Benefits: Stable performance, minimal allocation
- Trade-off: Memory overhead, complexity

When to Use

Factory: When you need flexible object creation
Pool: When you create/destroy many short-lived objects

Design Evolution | Real-World Applications

Factory Method:

- GUI frameworks (button factories)
- Database connectors (driver factories)
- Logging systems (logger factories)

Object Pool:

- Database connection pools
- Thread pools
- Game engines (particle systems, projectiles)
- Network socket pools

Summary

Summary | Week 10 Summary

What We Learned:

1. Identified extensibility problems (switch statements)
2. Applied Factory Method for flexible creation
3. Measured GC performance with monitoring tools
4. Implemented Object Pool for performance optimization

Design Principles Applied:

- Open/Closed Principle
- Dependency Inversion Principle
- Separation of Concerns
- Performance-aware design

Branch Structure

`10-01` (problem) → `10-02` (factory) → `10-03` (GC) → `10-04` (pool)

Referensi

Referensi | Referensi dengan Format *Numbered Citation*

The End

Questions? Comments?