UNIVERSITAS INDONESIA

DEPARTEMEN TEKNIK ELEKTRO

Pemrograman Berorientasi Objek dan Praktik-ENCE603010 (3 SKS)

The Game Loop & Singleton Pattern

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

Semester Gasal 2025 – 2026

Minggu 9

Pengantar: Perjalanan Pembelajaran Progresif

Pengantar: Perjalanan Pembelajaran Progresif | Week 09: Game Loop & Singleton Pattern

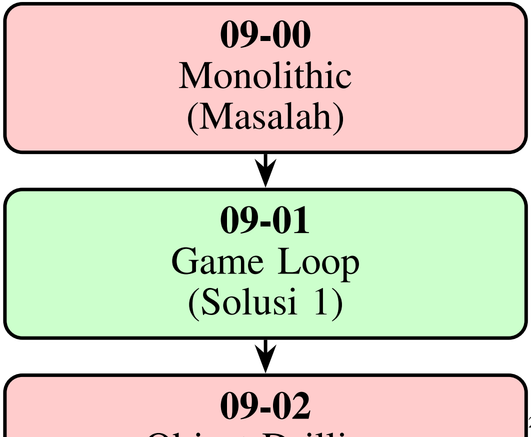
Tujuan Pembelajaran:

- Memahami **Game Loop Pattern**
- Memisahkan update dan rendering
- Mengidentifikasi anti-pattern **Object Drilling**
- Mengimplementasikan **Singleton Pattern**
- Menganalisis trade-off arsitektur

Pendekatan Pembelajaran:

- Progresif: Setiap branch menyelesaikan masalah sebelumnya
- Problem-first: Tunjukkan masalah, baru solusi
- Evidence-based: Gunakan metrik konkret (FPS, LOC)

Perjalanan 4 Branch



Branch 09-00: Masalah Monolithic

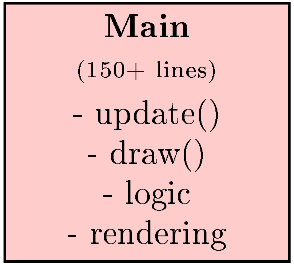
Branch 09-00: Masalah Monolithic | Branch 09-00: Desain Monolithic

Apa itu Monolithic?

- Semua kode dalam satu method `main()`
- 150+ baris dalam satu method
- Update logic tercampur dengan rendering
- Tidak ada separation of concerns

Masalah yang Ditimbulkan:

- **Frame rate coupling**: Rendering lambat → logic lambat
- **Untestable**: Tidak bisa test tanpa rendering
- **Poor maintainability**: 150+ baris sulit dipahami
- **No scalability**: Menambah entity = slowdown eksponensial



All in one!

09-00: Monolithic

Dampak Kritis

2 FPS achieved!

Branch 09-00: Masalah Monolithic | Branch 09-00: Struktur Kode Monolithic

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

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

            // Check collisions
            if (npc.collidesWith(coin)) {
                score += 10;
            }

            // Render (SLOW - menyebabkan masalah!)
            clearScreen();  // 50ms delay!
            drawNPC(npc);
            drawCoin(coin);
            Thread.sleep(50);  // Flickering!
```

Analisis Masalah:

Metrik	**Nilai**
Lines of Code	150+
FPS	2
Test Coverage	0%
Maintainability	Sangat Rendah

Masalah Utama

Tidak bisa unit test!
Logic tidak bisa ditest tanpa memicu rendering.

Branch 09-01: Solusi Game Loop

Branch 09-01: Solusi Game Loop | Branch 09-01: Game Loop Pattern

Solusi: Separation of Concerns
Pisahkan kode monolithic menjadi class-class khusus:

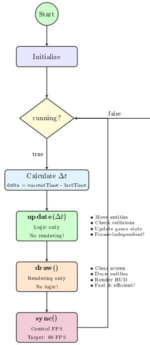
- `GameEngine`: Kontrol game loop
- `GameLogic`: Update game state
- `GridRenderer`: Handle rendering saja

Konsep Kunci:

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

Benefits:

- Testable (no display needed)



Key Benefits:
✓ Testable (update() without display)
✓ Frame-rate independent (delta time)
✓ Predictable performance (60 FPS)
✓ Clean separation of concerns

Branch 09-01: Solusi Game Loop | Branch 09-01: Implementasi Game Loop

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

    public void start() {
        lastTime = System.currentTimeMillis();

        while (running) {
            long currentTime =
            ↪  System.currentTimeMillis();
            float delta = (currentTime - lastTime) /
            ↪  1000.0f;
            lastTime = currentTime;

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

Main.java sekarang SANGAT sederhana:

```java
public class Main {
    public static void main(String[] args)
    ↪  {
        GameEngine engine = new
        ↪  GameEngine();
        engine.start();
    }
}
```

Hanya 3 baris! Dari 150+ baris menjadi 3 baris.

Achievement Unlocked!

Clean, testable, professional 60 FPS architecture!

Branch 09-01: Solusi Game Loop | Branch 09-01: Performance Improvement

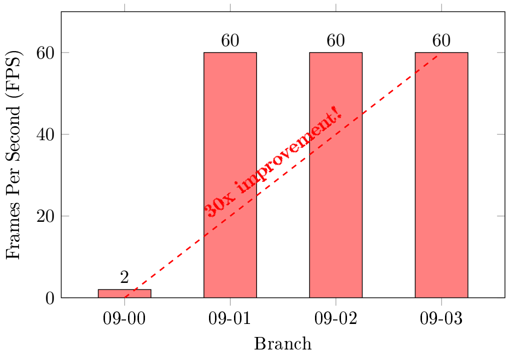
Komparasi 09-00 vs 09-01

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

Hasil

30x peningkatan FPS!
50x pengurangan kompleksitas!

FPS Performance Comparison



Branch 09-02: Masalah Baru (Object Drilling)

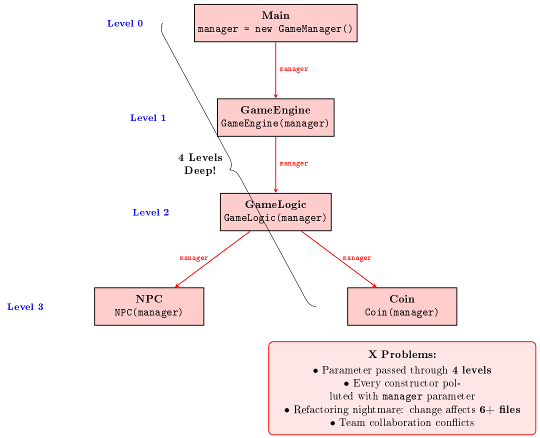
Branch 09-02: Masalah Baru (Object Drilling) | Branch 09-02: Ekspansi Game - Requirement Baru

Requirement Baru:
Tambahkan HUD (Heads-Up Display) untuk menampilkan:

- Current score
- Game time
- Player level

Design Challenge:
Multiple class perlu akses `GameManager`:

- `GameLogic` → update score
- `HUD` → display score
- `NPC` → check game state
- `Coin` → add points



X Problems:
- Parameter passed through **4 levels**
- Every constructor polluted with `manager` parameter
- Refactoring nightmare: change affects **6+ files**
- Team collaboration conflicts

Branch 09-02: Masalah Baru (Object Drilling) | Branch 09-02: Object Drilling Anti-Pattern

Implementasi dengan Object Drilling:

```java
// Main.java
GameManager manager = new GameManager();
GameEngine engine = new GameEngine(manager);
engine.start();

// GameEngine.java
public GameEngine(GameManager manager) {
    this.logic = new GameLogic(manager);
    this.hud = new HUD(manager);
}

// GameLogic.java
public GameLogic(GameManager manager) {
    this.npc = new NPC(manager);
    this.coins.add(new Coin(manager));
}

// 4 LEVELS DEEP!
```

Consequences:

Bug Kritis!

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

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

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

Output

Branch 09-03: Solusi Singleton

Branch 09-03: Solusi Singleton | Branch 09-03: Singleton Pattern

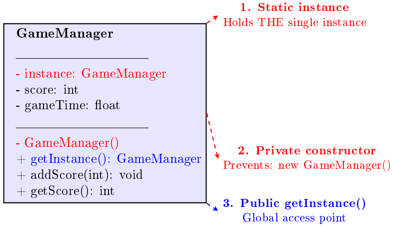
Solusi: Guarantee Single Instance
Singleton pattern memastikan class hanya punya SATU instance dan menyediakan global access point.

Tiga Komponen Kunci:

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

Benefits:

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



Implementation:

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

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

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

Usage:

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

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

Branch 09-03: Solusi Singleton | Branch 09-03: Implementasi Singleton

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

    private int score;
    private float gameTime;
    private int level;

    // 2. Private constructor
    //    (prevents: new GameManager())
    private GameManager() {
        this.score = 0;
        this.gameTime = 0.0f;
        this.level = 1;
        System.out.println("[GameManager]
            Singleton created: " + this.hashCode());
    }

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

Usage yang Clean:

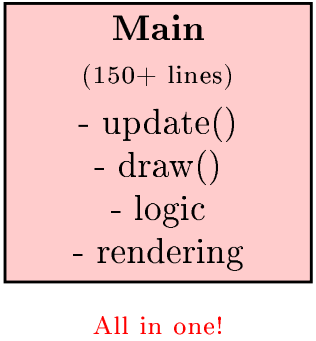
```
// Main.java - No parameters!
public class Main {
    public static void main(String[] args)
    ↪   {
        GameEngine engine = new
        ↪   GameEngine();
        engine.start();
    }
}

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

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

Analisis Komparatif

Analisis Komparatif | Evolusi Arsitektur: 09-00 → 09-03



09-00: Monolithic

Transformasi Progresif:

09-00 → 09-01:

- Monolithic → Separated concerns
- 2 FPS → 60 FPS
- Untestable → 100% testable

09-01 → 09-02:

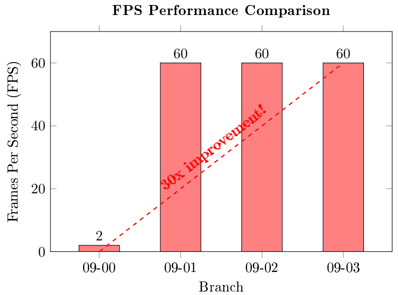
- Add new feature (HUD)
- Introduces object drilling
- Bug: multiple instances

09-02 → 09-03:

- Singleton pattern
- Zero parameters
- Single instance guaranteed

Analisis Komparatif | Comprehensive Metrics: All Branches

Metrik	**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



Design Patterns Deep Dive

Design Patterns Deep Dive | Game Loop Pattern: Intent & Structure

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

Structure:

- `update(deltaTime)`: Update game state
- `draw()`: Render current state
- `sync()`: Control frame rate

Participants:

- **GameEngine**: Orchestrates the loop
- **GameLogic**: Implements game rules
- **Renderer**: Draws to screen

Consequences:

Benefits:

- Frame-rate independence
- Testability tanpa display
- Clear separation of concerns
- Predictable performance

Liabilities:

- More classes (increased complexity)
- Initial learning curve
- Need to manage delta time

Design Patterns Deep Dive | Singleton Pattern: Intent & Trade-offs

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 to sole instance
- No global variables
- Lazy initialization possible
- Easy to refactor

Liabilities:

- Global state (testing harder)
- Hidden dependencies
- Thread safety concerns
- Violates Single Responsibility Principle

Alternatives:

- **Dependency Injection**: Pass dependencies explicitly
- **Service Locator**: Registry of services
- **Static Class**: No instantiation needed

Trade-off
Singleton menyelesaikan object drilling, tapi introduces global state. Gunakan dengan bijak!

Diskusi & Assessment

Diskusi & Assessment | Discussion Points untuk Kelas

Pertanyaan untuk Mahasiswa:

1. Mengapa frame rate coupling adalah masalah kritis dalam game?
2. Apa trade-off dari Singleton pattern?
3. Kapan Anda TIDAK akan menggunakan Singleton?
4. Bagaimana delta time memungkinkan frame-rate independence?
5. Apa alternatif selain Singleton untuk mengatasi object drilling?

Critical Thinking:

- Apakah global state selalu buruk?
- Bagaimana cara test class yang menggunakan `GameManager.getInstance()`?
- Apa yang terjadi di environment multi-threaded?
- Kapan Dependency Injection lebih baik daripada Singleton?

Diskusi & Assessment | Assessment Rubric (100 points)

Component	**Points**	**Criteria**
Code Implementation	40	• Correct Singleton implementation (10) • Working game loop (15) • Proper separation of concerns (15)
Testing	20	• Unit tests for GameLogic (10) • Test coverage > 80% (10)
Design	20	• UML diagrams (10) • Architecture explanation (10)
Documentation	10	• JavaDoc comments (5) • README with design decisions (5)
Code Quality	10	• Style compliance (5) • No compiler warnings (5)
Total	**100**	

Summary

Summary | Week 09 Summary: Key Takeaways

Konsep Utama:

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

Perjalanan Pembelajaran:
`09-00` (Problem) → `09-01` (Solution) →
`09-02` (New Problem) → `09-03` (Final Solution)

Hasil Akhir:

Achievement	**Status**
60 FPS	**OK**
Testable	**OK**
Maintainable	**OK**
Scalable	**OK**
Zero Object Drilling	**OK**
Single Instance	**OK**

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

Referensi

Referensi | Referensi dengan Format *Numbered Citation*

The End

Questions? Comments?