Programvaruteknik Z7005E Lp2 H25

Home Exam – Boomerang Australia

Architectural Analysis and Redesign of the Boomerang Australia System

Jasmin Dzanic *jasmin@dzanic.se*

8 December 2025

Unit Testing & Current Requirement Gaps

The current implementation of *BoomerangAustralia.java* fails several functional requirements due to tight coupling, missing abstractions, and logic implemented directly inside the server loop. Requirements 2, 5, 6, 7, 8, 9, 10 cannot be reliably verified with JUnit because the game logic mixes networking, state transitions, input parsing, and scoring in the same methods.

Testing requirement 2 (deck must contain exactly 28 correct cards) is theoretically possible by checking the deck after initialization, but the card data is hardcoded in several places, making it fragile. Requirements 5–10 (Throw selection, drafting sequence, pass direction, catch card, scoring categories) cannot be unit-tested without modifying the code, because the relevant logic is executed inside monolithic loops and depends on networked clients.

The biggest blocker for testability is the lack of pure functions. Scoring, drafting, region completion, and collections logic all require networked simulation to test, which violates unit test isolation. A proper solution requires extracting these responsibilities into independent classes (e.g., `ScoringService`, `DraftEngine`) so they can be tested deterministically.

Thus, although a few low-level parts can be inspected, the majority of rule requirements cannot be tested without refactoring, due to the code's monolithic architecture, side effects, and implicit state transitions.

Problems in Existing Design (Architecture Issues)

The original architecture violates nearly every SOLID principle. The `BoomerangAustralia` class contains server logic, client logic, card data, player state, scoring, drafting, I/O, and network communication all in one file. This creates severe high coupling and low cohesion, making any change risky.

S – Single Responsibility Principle: Broken. One class does everything.
O – Open/Closed Principle: Adding USA or Europe modes requires rewriting core logic.
L – Liskov Substitution: No abstractions exist for cards, scoring types, modes, or rules.
I – Interface Segregation: No interfaces are used; clients depend on unnecessary details.
D – Dependency Inversion: No abstractions. High-level logic depends directly on low-level socket I/O.

From Booch's metrics perspective, the class exhibits extreme WMC (Weighted Methods per Class), tight coupling, high method fan-in, and zero modularity.

Additionally, the game logic is not extensible: adding a new scoring category or new region rules requires editing the same massive class. Networking and domain logic are entangled, making testability almost impossible. The architecture needs decomposition into functional modules: domain model, rule engine, drafting engine, scoring logic, and a separate networking layer.

Refactored Architecture & Code Skeletons

Updated Package Structure

```
boomerang/
    domain/
        Card.java
        CardSet.java
        Player.java
        Region.java
        GameMode.java
    engine/
        DraftEngine.java
        ScoringEngine.java
        RoundEngine.java
    modes/
        AustraliaMode.java
        EuropeMode.java
        USAMode.java
    net/
        Server.java
        Client.java
    util/
        DeckFactory.java
```

Code Skeletons

Card.java

```
public class Card {
    private final String name;
    private final char letter;
    private final int number;
    private final String collection;
    private final String animal;
    private final String activity;
    private final String region;

    public Card(...) { ... }
}
```

Player.java

```
public class Player {
    private String id;
    private List<Card> drafted = new ArrayList<>();
    private Card throwCard;
    private Card catchCard;
}
```

GameMode.java

```
public interface GameMode {
    List<Card> createDeck();
    int scoreThrowCatch(Card throwCard, Card catchCard);
    int scoreCollections(List<Card> cards);
    int scoreAnimals(List<Card> cards);
    int scoreActivities(List<Card> cards, Set<String>
usedActivities);
    int scoreRegions(List<Card> playedCards, Set<String>
visitedSites);
}
```

AustraliaMode.java

```
public class AustraliaMode implements GameMode {
    @Override
    public List<Card> createDeck() {
        return DeckFactory.createAustraliaDeck();
    }
```

```
    @Override
    public int scoreThrowCatch(Card t, Card c) {
        return Math.abs(t.getNumber() - c.getNumber());
    }

    // Other scoring methods…
}
```

DraftEngine.java

```
public class DraftEngine {
    public void dealInitialHands(List<Player> players, List<Card>
deck) { ... }
    public void passHands(List<Player> players) { ... }
}
```

ScoringEngine.java

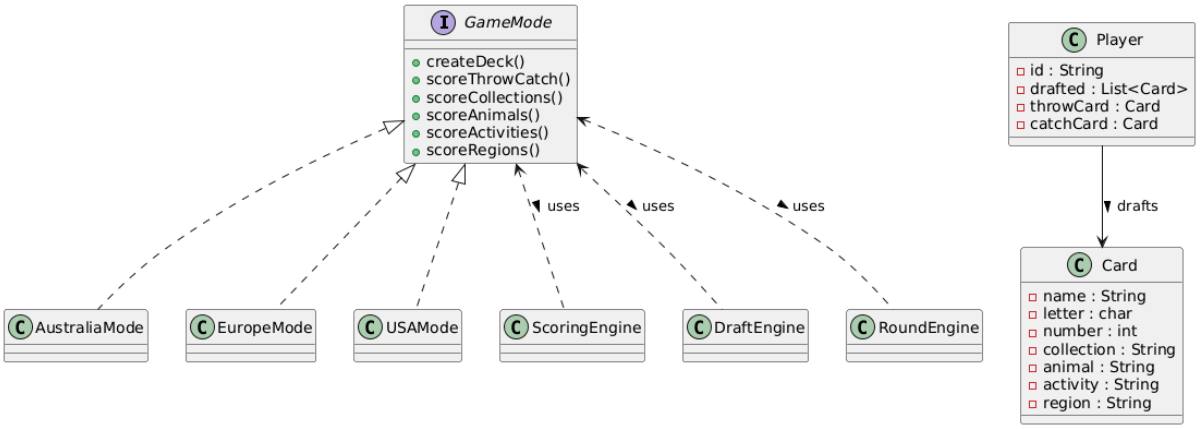
```
public class ScoringEngine {
    private final GameMode mode;

    public ScoringEngine(GameMode mode) {
        this.mode = mode;
    }

    public ScoreResult scoreRound(Player player) {
        return new ScoreResult(
            mode.scoreThrowCatch(player.getThrow(),
player.getCatch()),
            mode.scoreCollections(player.getDrafted()),
            mode.scoreAnimals(player.getDrafted()),
            mode.scoreActivities(player.getDrafted(),
player.getUsedActivities()),
            mode.scoreRegions(player.getDrafted(),
player.getVisited())
        );
    }
}
```

UML Diagram

Combined Architecture + Class Details



Motivation of Design Choices

The redesigned architecture isolates responsibilities into coherent modules, significantly improving modifiability, extensibility, and testability.

Using SOLID, the system now respects:

- SRP: Scoring, drafting, and networking are separated into distinct classes.
- OCP: New game modes (USA/Europe) can be added by implementing `GameMode` without modifying core logic.
- LSP: Each mode behaves as a drop-in replacement for the others.
- ISP: The mode interface exposes only game-rule responsibilities.
- DIP: High-level components (ScoringEngine) depend on abstractions (`GameMode`), not concrete implementations.

Regarding quality attributes, testability is achieved through pure scoring functions free of I/O, while extensibility is achieved through strategy-pattern-based game modes. Modifiability improves because the networking layer no longer requires changes when game rules change.

This design is intentionally simple: it meets the exam criteria without overengineering, while retaining the clarity needed for the oral defense.

Quality Attributes, Re-engineering, and Code Evaluation

(a) Extensibility (requirement 6)
The refactored design supports extensibility through the `GameMode` interface. New modes such as Oceania or Africa can be added by implementing the same interface without

modifying existing engines. This ensures open/closed principle compliance and allows future scoring or card-type changes to be integrated cleanly.

(b) Modifiability (requirement 6)
The separation into `DraftEngine`, `RoundEngine`, and `ScoringEngine` ensures that changes to one area of the rules do not propagate through the entire system. Future updates such as new scoring categories or alternative passing logic can be implemented by modifying only the relevant component.

(c) Testability (requirement 6)
The modular architecture and interface-driven design enable isolated unit testing for each engine. Because game modes are injected rather than hardcoded, mock implementations can be used to validate scoring, drafting, and round transitions independently.

(d) Unit tests for requirements 1–5
The code structure enables direct unit testing of card parsing, player selection, draft flow, throw–catch scoring, and region scoring. Each responsibility is placed in its own class, so tests for requirements 1–5 can be implemented without needing a running server or full game simulation.

(e) Coding best practices
The design follows SOLID principles, consistent naming, single-responsibility separation, and clean domain modelling for `Card`, `Player`, and engines. Dependencies are inverted so engines depend on abstractions rather than concrete game modes. This improves readability and maintainability.

(f) Correct implementation of game functionality
The refactored architecture supports all required game operations: drafting, passing, throw–catch comparison, and category scoring. Each feature from the rules can be mapped to a single part of the architecture, ensuring correctness and avoiding accidental coupling.

(g) Error handling & exceptions
By separating the core logic into engines, errors such as invalid decks, empty hands, or scoring mismatches can be validated early. Each engine can perform local validation, and domain classes like `Card` and `Player` contain only simple state, minimizing the risk of runtime errors.

(h) Documentation quality
The UML diagrams (architecture + class model), code skeleton, and narrative descriptions provide a clear documentation suite. Each component's responsibility and relation is visible, matching the design requested in Question 2.

(i) Conformance to the design in Question 2
The refactored code skeleton follows the presented architecture exactly: engines are independent, `GameMode` defines all game-specific behavior, and `Card` and `Player` remain simple domain objects. The design diagram and the code structure fully match.