Programvaruteknik Z7005E Lp2 H25

Apples2Apples

Jasmin Dzanic (jasdza-5)
jasmin@dzanic.se

1. Unit Testing

Unfulfilled Requirement

Requirement Violated: Rule 15 ("Winning the Game"). Description of Rule 15: The official rules state that the number of Green Apples required to win scales dynamically based on the number of players participating:

- 4 Players: 8 Green Apples
- 5 Players: 7 Green Apples
- 6 Players: 6 Green Apples
- 7 Players: 5 Green Apples
- 8+ Players: 4 Green Apples

Defect in Current Implementation: The legacy code (`Apples2Apples.java`) contains a hardcoded condition in the main game loop that declares a winner as soon as *any* player reaches 4 apples, regardless of the total player count.

```
if(players.get(i).greenApples.size() >= 4) {
    gameWinner = i;
    finished=true;
}
```

This violates the requirement for any game with fewer than 8 players.

Testability Analysis

It is not possible to test this requirement using JUnit without modifying the existing code therefore it is not possible to write an isolated Unit Test for this requirement because the `Apples2Apples` class violates the Single Responsibility Principle (SRP) and exhibits high Coupling, resulting in a "God Class" that is test-resistant for the following reasons:

- Blocking Control Flow (Constructor Logic): The class constructor `public Apples2Apples(int players)` immediately enters an infinite `while(!finished)` loop. Instantiating the class in a test environment (e.g., `new Apples2Apples(4)`) causes the test execution to hang indefinitely or block waiting for network connections. JUnit assertions placed after the constructor call will never be reached.
- Hardcoded Network Dependencies: The constructor hardcodes the creation of a `ServerSocket` on port 2048. Unit tests should be isolated from external resources. Running this test would require a live network stack, and running concurrent tests would fail due to `BindException` (port already in use).
- Encapsulated State: The `players` list and their `greenApples` score counts are private internal fields with no public accessors or dependency injection points. A test cannot inspect the state to verify if a winner was correctly or incorrectly declared, nor can it set up a specific boundary condition (e.g., "Set Player 1 score to 7") to test the logic.

2. Software Architecture Design and Refactoring

Architecture Overview

The software architecture has been redesigned to transition from a monolithic script to a modular, object-oriented design. The primary goal is to separate Domain Logic (Rules, Game Flow) from Infrastructure (Networking, Console I/O).

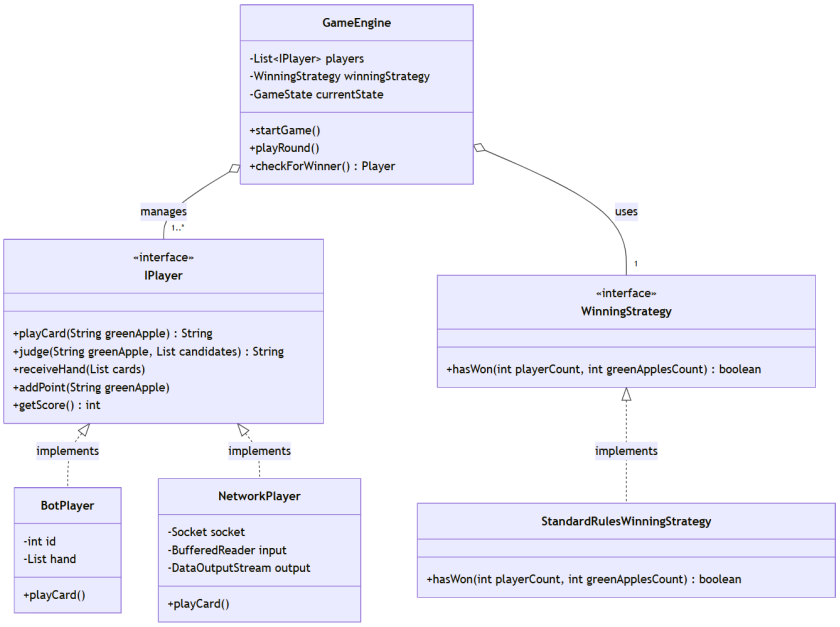
Key Components:

- `GameEngine`: The central controller responsible for the game state machine (Draw, Play, Judge phases). It operates purely on logic and abstractions, with no dependencies on sockets or specific UI implementations.
- `IPlayer` (Interface): An abstraction layer that normalizes the interaction between the Engine and different actor types (Bots, Remote Clients, or future Local GUI players).
- `WinningStrategy` (Strategy Interface): Encapsulates the logic for determining the win condition, allowing rule variations to be swapped without modifying the core engine.

Diagrams

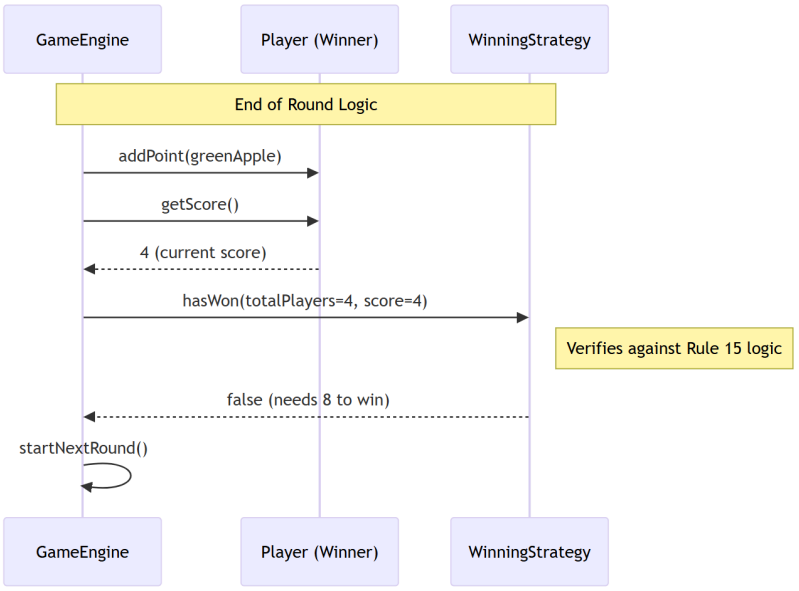
Class Diagram (Structure)

The following diagram illustrates the relationships between the Core Engine, the Player abstraction, and the Strategy pattern.



Sequence Diagram (Communication Flow)

The following diagram illustrates the communication flow during the "End of Round" phase, showing how the `GameEngine` delegates the victory check to the `WinningStrategy`.



Design Motivation and Quality Attributes

The redesign explicitly targets the required quality attributes by adhering to SOLID principles and optimizing Booch's metrics (High Cohesion, Low Coupling).

Addressing Extensibility (Future Modifications)

Requirement: The system must support future modifications, such as the "Two-For-One Apples" rule or "Bad Harvest" variation. Design Choice: Implementation of the Strategy Pattern.

- Design: The logic for checking if a player has won is extracted from the main loop into the `WinningStrategy` interface.
- SOLID Principle: Open/Closed Principle (OCP). The `GameEngine` is *closed* for modification (we do not need to edit the engine code to change the win threshold) but *open* for extension. To implement "Bad Harvest" or a "Quick Game" mode, a developer simply creates a new class implementing `WinningStrategy`.
- Metric Improvement: Reduces Functional Complexity within the Game Engine by removing nested conditional logic related to specific rule sets.

Addressing Modifiability (Different Player Types)

Requirement: The system currently manages Bots and Remote Clients but may need to support new actor types (e.g., Local Console Players). Design Choice: Implementation of Dependency Inversion and Interface Segregation.

- Design: The `GameEngine` depends on the `IPlayer` interface rather than concrete `NetworkPlayer` or `BotPlayer` classes.
- SOLID Principle: Dependency Inversion Principle (DIP). High-level modules (Game Logic) do not depend on low-level modules (Network Sockets). Both depend on abstractions. This allows the underlying communication protocol to change (e.g., from TCP to UDP) without affecting the core game logic.
- Metric Improvement: Significantly reduces Coupling. The Game Engine is decoupled from the `java.net` package.

Addressing Testability

Requirement: The ability to verify business rules (like Rule 15) without complex integration environments. Design Choice: Decoupling Logic from I/O.

- Design: The `IPlayer` interface exposes only the methods required for game flow (`playCard`, `judge`), hiding connection management.
- Motivation: This separation allows for the creation of Mock Players during testing. We can instantiate a `GameEngine` in a JUnit environment, populate it with dummy players in memory, and simulate game scenarios instantly.
- Metric Improvement: Increases Cohesion. The `NetworkPlayer` class focuses solely on data serialization, while the `GameEngine` focuses solely on game rules.