D7032E - Apples to Apples

Martin Askolin, marsak-8@student.ltu.se

October 11, 2024

Contents

1 Unit Testing

Requirement 4 is unfulfilled. The amount of players able to join is unrestricted making it possible to run out of cards before each has been dealt seven. We can not test this requirement with JUnit without modifying the code. This is because we don't have access to the game's state until the game is over. We could however catch the byproduct of the failed requirement by expecting a *IndexOutOfBoundsException* for a certain amount of players but at that point you are already aware of the problem so it's to no use.

Requirement 15 is not fulfilled for fewer than 8 players. Currently you are able to win holding only four green apples regardless of the player count. Since the logic for winning the game is put in the constructor we can never test the logic itself with JUnit. However, the game's state is stored in the `Apples2Apples`'s public variables.. If JUnit could play the game (which it cant), and then access the `players ArrayList` it could assert that the player with the highest number of green apple (the winner) should have the correct number corresponing to the number of players. However I don't believe this to be possible without changing the current code.

2 Quality attributes, requirements and testability

Requirement 17 - 18 state that it should be easy to modify, extend, test and to some extent debug. The problem with quality attributes is how inherently subjective they are, how would you measure these things? To make them more measurable and as a result more objective, metrics where introduced. These metrics are cohesion, coupling, sufficiency, completeness, and primitiveness. If the requirements would have mentioned these instead it would have made it a lot easier to see if the requirements were met or not. The result of these poorly written requirements is that we wont be able to objectively tell if we achieved them or not.

3 Software Architecture and code review

The project is written in one file and spread out over three classes; `Player`, `PlayedApple`, and `Apples2Apples`. Each class exhibits low primitiveness and cohesion, e.g they have too much responsibility. As an example, the `Player` class currently integrates logic for both human and bot players, networking, and checking communication types. This mixture of responsibilities complicates understanding and maintaining the code, as developers must navigate through unrelated functionalities to implement changes or add new features. As a result of this mixed responsibility, extensibility, modifiability and testability will all suffer.

Additionally, a significant barrier to the testability of the code is the presence of all game logic within the constructor. This design choice leads to a monolithic structure, making it challenging to isolate and test individual components of the system. I believe this relates to poor cohesion but I find it hard to explain this major problem as a quality attribute due to the intertwining of various functionalities in one place. For example, the constructor initializes the game state, manages player actions, and establishes networking protocols all at once. Ideally, each of these responsibilities should be handled in separate methods or classes, promoting a more modular design.

The project also suffers from tight coupling. The classes are strongly interdependent on each other and manipulate each other directly instead of going through encapsulated methods. An example of this is the global variable `Apples2Apples.playedApple`, and the player's `hand` and `greenApples` variables. This results in severely reduced modifiability and testability, as any change in one class can lead to unexpected behaviors in others, creating a fragile system where modifications are risky. For instance, if the logic for managing the player's hand needs to be altered, it may require changes in both the `Player` and `Apples2Apples` classes, making the process cumbersome and error-prone.

4 Software Architecture design and refactoring

Figure 1 shows a module diagram of the project where the arrows indicate communication flow. The game module contains the core elements of the game such as phases, and a mediator class (mediator pattern) between the games information and the human players. The main method (`Program.cs`) initiates necessary networking and communication components such as a server and information broadcaster and injects dependencies to the game such as the players. This separation of concern aim to provide loose coupling and high cohesion and should result with code that's easy to modify without breaking dependencies.

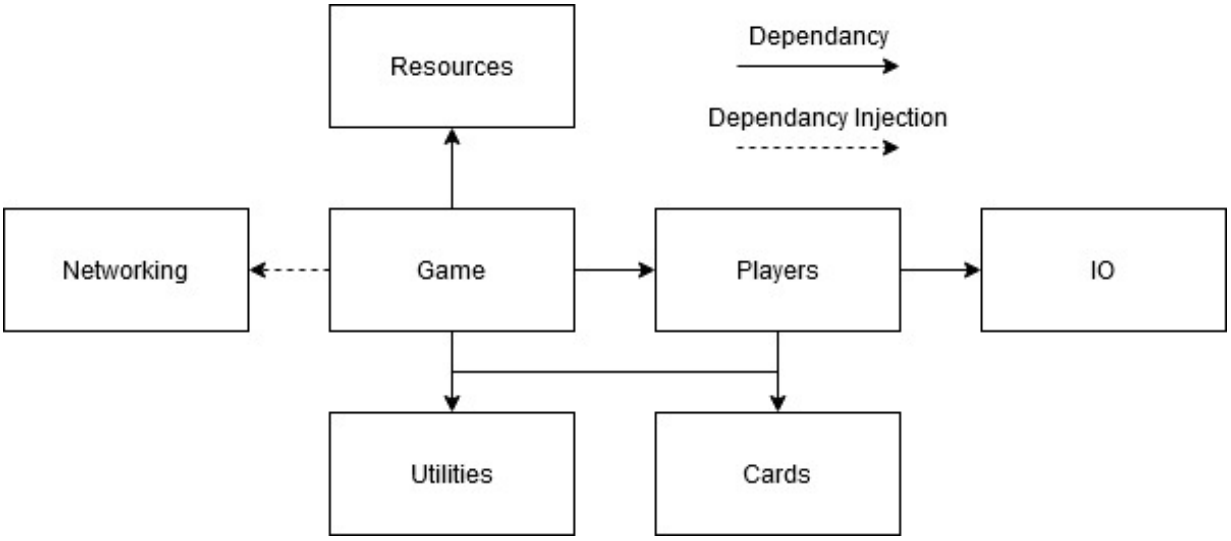

Figure 1: Module diagram showing communication flow and dependencies

A state pattern is used for the game's different phases as described in the scenario[1]. Since the standard game loop is simple a sequential state machine should suffice. Figure 2 describes the states and their order. The states have no direct reference to the game nor other states and rely on dependency injections for it's context. The states then use a observer pattern to inform listeners of key information calculated in the state such as the new green apple, new judge, verdicts, etc. This reduces coupling and keeps classes primitive and complete and allows the game to be easily extended and modified while keeping functionality easy to test. As mentioned earlier this information is then broadcasted to the players through a mediator. The game variation class is responsible for gluing everything together and should inject dependencies as needed.

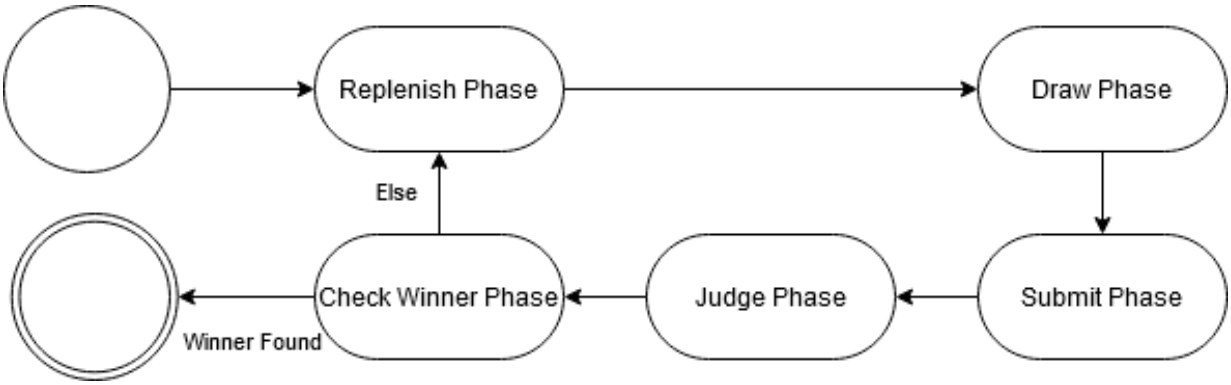

Figure 2: Phase diagram for the different phases of a game

By looking at the UML diagram in Figure 3 we see that the player is separated into a controller and a pawn. This is a pattern popularized by the Unreal Engine and is used to separate the control logic of a player (the Pawn) from the decision-making logic (the Controller). This allows the controller to be swapped out mid game if for example a player disconnects without disrupting the game. More over the input and output is standardized for different sources using a abstract class called `ClientIO`. Finally interface segregation has been used to create different sorts of red apples and phases while keeping them related to one another and interchangeable promoting extensibility. I suggest you read this paper[2] published by Unity (a popular game engine) as it's explanation of the SOLID principles and different design patterns in relation to game development has been a huge inspiration for this design.

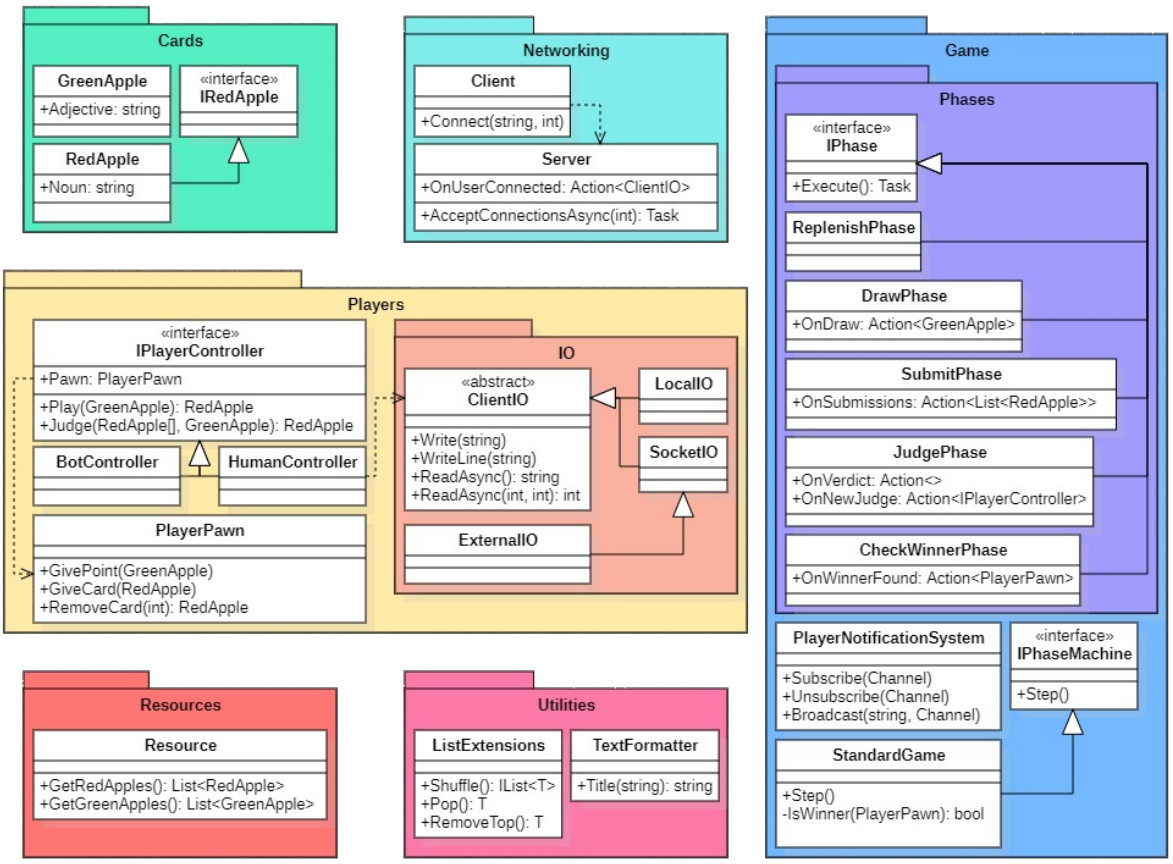

Figure 3: UML diagram of the entire project

5 Quality Attributes, Design Patterns, Documentation, Re-engineering, Testing

Build and run instructions are included in the README.

References

[1] Josef Hallberg. Home exam – software engineering – d7032e. https://staff.www.ltu.se/~qwazi/d7032e2018/D7032E-2018-HomeExam.pdf, 2018. Accessed: 2024-10-11.

[2] Unity. Level up your code with design patterns and solid. https://unity.com/resources/design-patterns-solid-ebook, 2024. Accessed: 2024-10-11.