Software engineering and improvement of program by using knowledge learnt from the course.

D7032E: Apples2apples

Home Exam

Måns Oskarsson

Måns Oskarsson
mnsosk-7

CONTENT

INTRODUCTION

This project was done in Java and all resources for this project is accompanied in the same folder where this document is included. Java SDK version 17 was used but not any other to old version should suffice.

HIERARCHY

The folder src contains all source code both old and new, all changes that has been made by the student to the original code has been migrated to a new package called project (in src). The text document of all different apples can be found under resources.
A Javadoc have also been generated hence Javadoc formatting has been used when commenting. Therefore, some comments may be quite extensive since when reading documentation, one may simple not have access to the source code or just want to get a clearer overview by only reading the documentation.

HOW TO RUN

It's recommended to import the project into the IDE called IntelliJ since executions profiles has been made for playing and testing. However, this is not required, in order to run it manually one must open Apples2apples.java and run the main function with different parameters:

Multiplayer:

1. Start a server session.
   Arguments: 'HOST 1', host for one client. Will wait for one player, then start game.
2. Connect to server session.
   Arguments: 'CONNECT localhost', connects to localhost.

Or (only bots):

1. Run with no parameters. Will run server and fill with bots only, then start game.
   Debug in Environment.java need to be true in order to see any prints whatsoever in server window. Will be completely blank otherwise when running with bots only.

In the same working directory as Apples2apples, a file called Environment.java can be found. In this file necessary parameters can be edited in order to change aspects of the program/game during execution or testing. The original thought was to make an option feature for server before hosting, but in the end, this was unnecessary given the assignment. Recommended to inspect this file.

Running tests has a bundled into a test profile in IntelliJ, which makes it possibly to run every test at once. In Environment.java some parameters for testing can be modified. Such as how many times a test should be executed per function and max size allocation for test samples, max card in decks.

QUESTIONS

1 UNIT TESTING

Requirements tagged with weak are requirements that need some clarification or improvement.

1. Fulfilled. All cards are read and added to an ArrayList.
2. Fulfilled. Same.
3. Fulfilled. Both the green apples and red apples are shuffled. Swapped.
4. Fulfilled. 7 cards are dealt to all player correctly.
5. Fulfilled. Judge is randomized.
6. Fulfilled (weak). Green apple is not showed to bots. Test is *AssertEquals* where expected is green apple being dealt to everyone and actual is card dealt (player.greenApple). In order to use this test, one must modify the code since most functionality is coupled behind user input (keyboard). Or make it possible to only fill game with bots where no user input is needed.
7. Fulfilled. All players get request to play a card.
8. Fulfilled. Random order occurs.
9. Fulfilled. All players get to play their card before result is shown.
10. Fulfilled. Green apple is given to player as point.
11. Fulfilled. All red apples are discarded.
12. Fulfilled (weak). One card is given to the player if not judge, but not checked whatever the player has n number of cards. In order to test this one must use *AssertEquals* where expected value is max allowed cards in hand and actual hand.size(). Most functionality coupled behind user input. Run with only bots to exclude user input part or fixed input from user.
13. Fulfilled. Next player in list becomes judge.
14. Fulfilled. All green apples won are stored.
15. Unfulfilled. Fixed win condition where number of players is not considered. This can be tested by using formula $win = 6 + (6 - k)$ for win condition where k is number of players, lower bound and upper bound of 4 and 8+ people. Or just have fixed win values for amount players. Then use *AssertEquals* to check actual amount of green apples player has after win condition has triggered or when to expected. Or *AssertTrue* where logic for win condition is input. Modification of code is required since coupled behind user input when testing, also not any single function to simply test this requirement.

2 POOR REQUIREMENTS (16-17).

First, it's clear that these so called in addition requirement hasn't been followed. In terms of modifiability and extensibility the code is very monolithic, hard to extract and reuse functionality and parts of the program when all server and major game logic is written in a single function. Because of this testing will also be hard when everything is monolithic, like described earlier the design is also coupled behind user input making it impossible to test without modifying existing code.
Important design criterions should not be described as additional requirements since design patterns is the important skeleton of the whole project. The design pattern should be

extracted from the same requirement section as game logic and rules, since they are different. Maybe these weak “additional” requirements are why they haven’t been followed in the first place.

3 REFLECTION AND CODE REVIEW

In an Extensibility quality attribute standpoint.
Its very hard to alter and extend upon game logic since primitiveness is almost non-existing, meaning that it’s smaller, easier to understand and more likely to be reuse (code base is opposite). There isn’t any reusability since parts of the game logic is coupled inside a single function. Hence when extending one must either copy or modify large portions of code.
An example of poor extensibility will be demonstrated on this code base, extending upon game logic, where all players can vote on other players played apple (as described in assignment).

Original game logic can exist where judge votes and all non-judge players play and receive a card, but afterwards one must add functionality where negation of gamelogic is applied. Judge should be able to play and receive a card but not vote, and other players should not beable to play and receive a card but instead vote on their favorite apple.
This in theory can be added as bonus/extra phases after the original phases (or ofcourse modify already existing). But in terms of resuability and completeness this is an very good example of poor extensibility, because either some specific cases of useful functionality can be extracted or nothing at all.
No tests, no verifiablity for expanding upon software/hardware or game = very bad.

In a Modifiability quality attribute standpoint.
This is probably the worst quality attribute being fulfilled. So, the question is how receptive the system for change is when new technology is introduced or game logic. The answer to this question is very bad.
One can clearly see by analysing the code that changing technology to different network solutions will require massive changes since there exist such high coupling and low cohesion between state and functionality that changes like this makes it very complicated. One must modify and implement functionality on a lot of parts of the code where readability is insufficient making it hard to locate side effects on the system and worst of all no tests making large changes completely unknown, lacking verifiability. Sufficiency in the code is good since all functionality is in Apples2Apples and captures all the details, but like described earlier readability is horrible and its unclear what the class is modelled to be used for, so this is not necessarily good.

Extensibility and modifiability in this code base goes hand in hand by lack of thought when designing this game because in order to expand one must modify large portions of code, where lack of modifiability. Making parts of the code base object-oriented will be a good solution to overcome this.

In a Testability quality attribute standpoint.
Testability is quite poor. The problem space in the code is to messy and most important of all, main functionality for the game is coupled behind user input and there is no other way

than removing this for making testing to work or making it possible to add only bots and removing user input. Therefore, testing is impossible without fixing this.

Other than that testing is feasible, you can compare states in parts of the code base. Modularity will however make it much easier to understand and construct tests which is favourable.

4 SOFTWARE ARCHITECTURE

Making it object oriented will be good, since it will increase reusability and completeness but it's important that still some degree of sufficiency is kept meaning that functions capture the important details and do what they are intended to do. Primitiveness on top of this can also be improved in the code as described in section 3. Coupling should be reduced, and cohesion be increased since this is something one should strive for when designing an object-oriented solution.

The focus in development will be on page two, after the requirements in the assignment paper. Were the developer is considering adding or replacing phases.
Here a base/parent class called stage (renamed of phase) can be created, then use inheritance for creating new phases where some basic functionality is kept in the parent class, in terms of sufficiency, make it enough to define and capture necessary details for what the definition of a stage is. But also have reusability and completeness so it will be easy to create new stages, have a handler of some sort where you can build these stages like building blocks independently.
This handler will be running the game loop and each stage should have a flag of some sort where next stage is called if flagged. Here its important that all stages can be built and moved completely independently of each other but however the game logic must make sense, one cannot play cards if they haven't been dealt any in the first place. This should be verified by test.

Also, the developer wants to create and modify the cards, create so called wild cards where one may replace the card content with any content of their own choice.

Here a parent class called Card can be created, were functionality and definitions describing what a card is. This design will make it possible to inherit and create any own card type, generic card or wild card.
What is extremely important in this design is that polymorphism denies the potential of coupling since if an object Card is declared, it doesn't matter if it is a generic card or wild card. This will increase readability since you will have a clear definition of each component, and primitiveness will therefore increase.

Afterwards a class called Cards can be created which works as a parent class, a collection of many cards where you can have functionality such as pickOne, pickMany or shuffle.
Here child's, to the parent class Cards can be created.
A child called DrawPile for example, which contains an inner class discard pile (inherited from Cards) for all green and red apples. If one draws a green apple its also added to a discard pile which later can be restored.

A child called Board for example, which binds a player id to a card in the card collection (Cards).

Regarding extensibility this is good since you can expand with ease by creating any new collections of cards without altering the original design.
Now when a good design for all possible game objects have been introduced which is designed using object-oriented design pattern. Which encourage reuse of components (primitiveness) and having a sufficient game object design that makes it solid in terms of completeness.

The same methodology as before will be applied upon creating player objects, which contains an id, green apples hand, boolean for judge and other useful functionalities. Here two child classes called HumanPlayer and Bot can be created from a parent class called Player which results in improved readability and completeness than before.

In terms of the top of the hierarchy of the program, whenever the program is run.

When the server is about to run, and a class containing the game stages should run. This class will be called Game and here one may construct any game loop of their own choice, like described before where stages being building blocks. This flag as described earlier can be triggered in the current game stage, which will result in the Game class running the next stage in the game loop.
A class (named State) holding the different important states such as Board (cards being played by player), green apples and red apples draw pile etc. will be passed into as argument for the stages. One may be worried that this design choice may cause coupling between different states and functionality in the game stages, but this is not the case.
In terms of modifiability and testability it has no affect whatsoever. It's just increase readability so all states arrent all around the place. This state class will probably be accessible in the parent class for own created stages and be package visible where custom created stages are located. In terms of testability, every state is accessible with ease because of how references work in java, so this will not be a problem.

When the client is about to run and connect to the server, a class which will act as a terminal will run. This terminal will contain user input and handle traffic from the server for the client. In order to apprehend incoming traffic for the client a Message service will be introduced where an instruction can be added and content of any type. Instructions will be handled and depending on instruction the data inside the message will be digested in a specific way.

Server and client network functionality should be separated into different classes so when the developer wants to change technology it should be with ease in terms of extensibility and modifiability.

Own custom exceptions have also been introduced, in terms of being able to locate errors upon execution when testing.

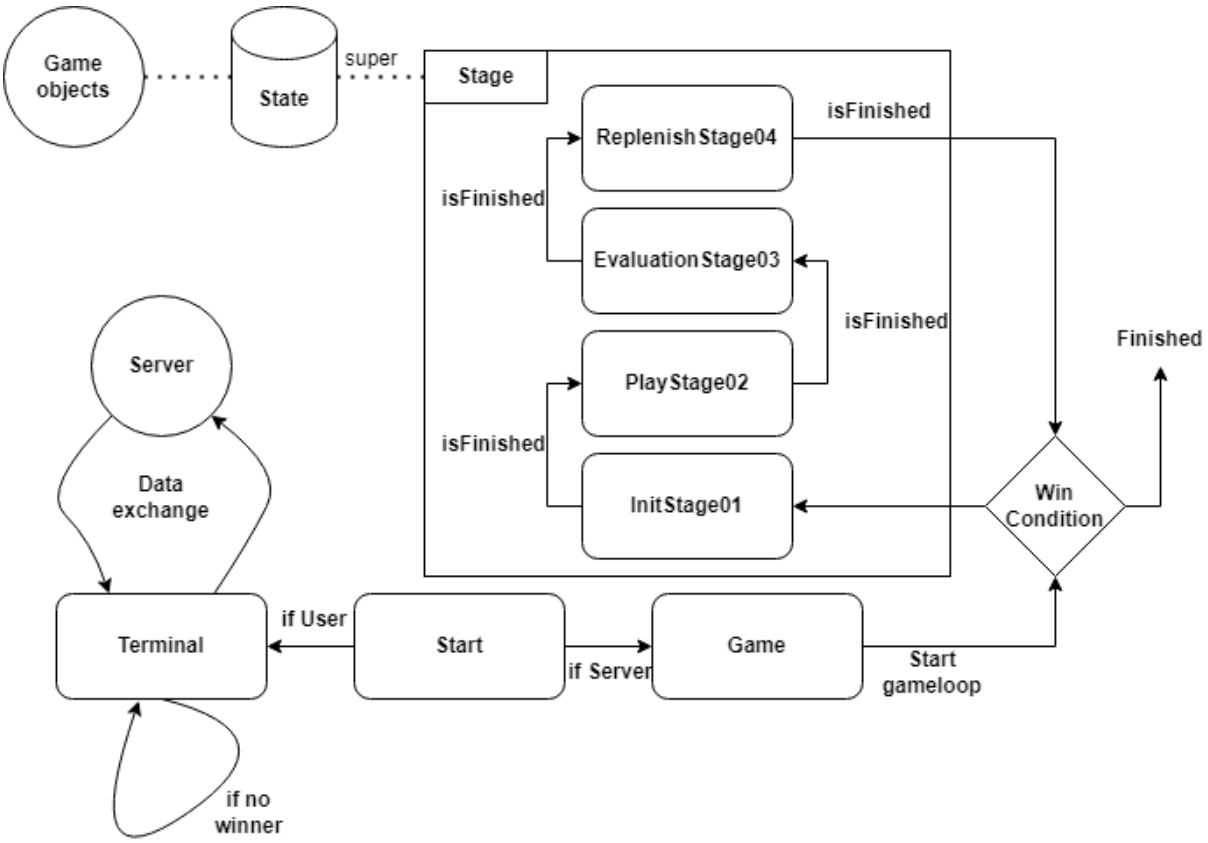

FIGURE 1: SIMPLE STATE FLOW

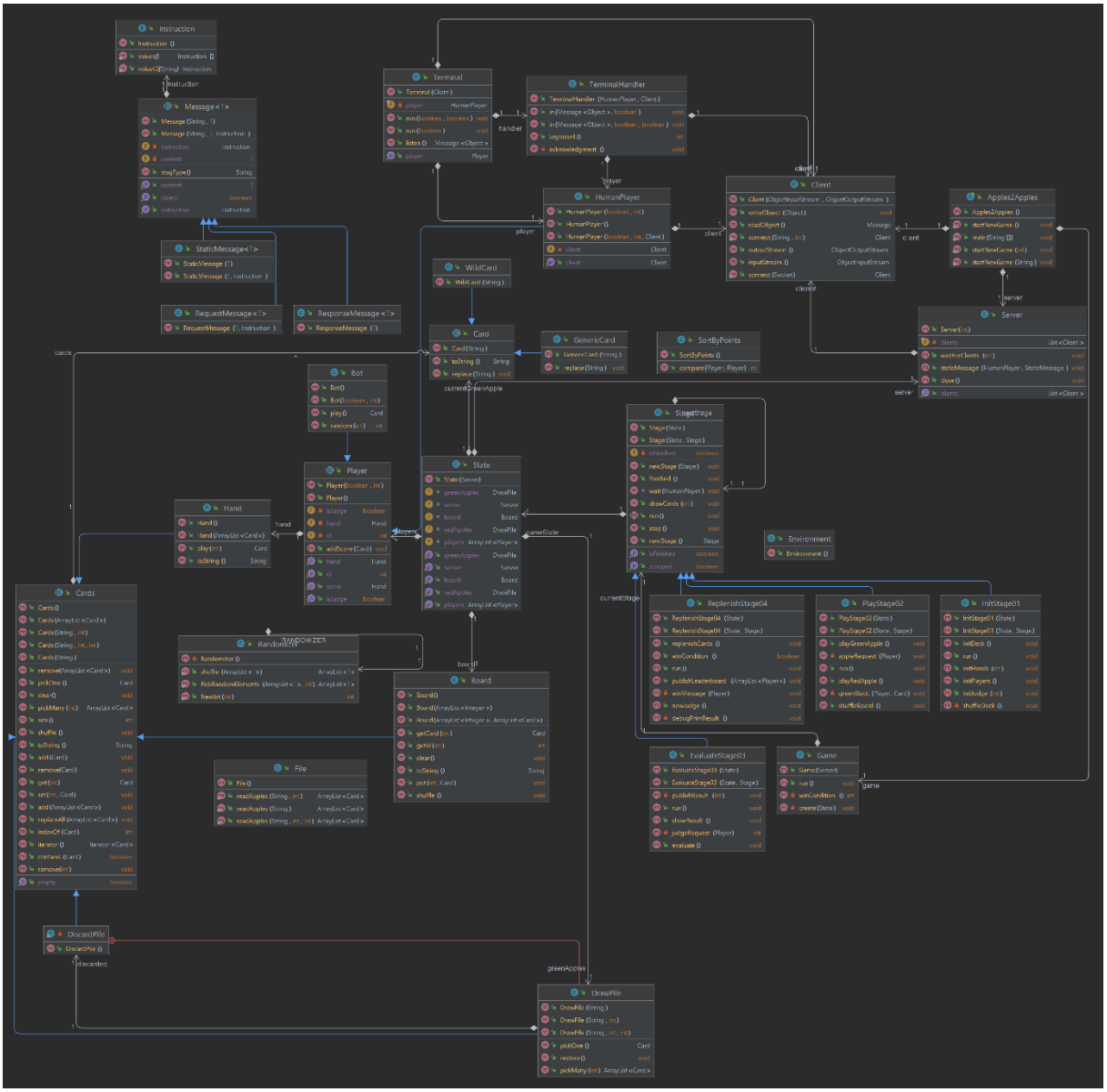

FIGURE 2: UML DIAGRAM OF PROJECT