Viktor Fällman
Vikfll-0@student.ltu.se

HOME ASSIGNMENT D7032E

Viktor Fällman
Vikfll-0@student.ltu.se

Question 1

On line 224 of Server.java the first card is drawn, there is no check to see if it is a special card, which is in violation of rule 3. We can also see that online 260 of Server.java, the checking of a bot calling uno is done by checking that it has sent something1;something2;something3 and not that something3 equals UNO. Therefore the bot doesn't have to say uno to actually call uno. This disregards rule 11. In the source code we can also see that a player can say uno with more than 1 card in its hand, and then finish by playing 2 cards which is against rule 12. As for the cards stacking, a player can play multiple cards that perform an action and only the last card will be handled as seen on line 128 of Server.java, violating rule 6. Rule 9 is broken on line 185 and 169, where there is checks to see if the deck contains cards, but no implementation of what happens in case of the deck running out of cards. This also makes Rule 7 fail, since if the deck will run out eventually then the player can not draw cards until a play is possible.

JUnit

The Server class can not be tested since upon creating the server object it starts running the game, and you will not be able to use the server object's methods. Therefore it is impossible to check any rules within the Server.java code as well as within the client.java code without changing the code. For us to even be able to test some of the Server.java could the code within the constructor should be moved to a separate method, so that we can instantiate a server without running it. This is the same for the client.java code.

Even with that change, since so much of the rules is handled within a everlasting while loop of the server, there are not that many rules you can check. canPlay method could be used to test if there is a play available for different played cards and hands, and viable Choice for card specific tests.

Question 2

Testability, there are no specific guidelines in requirements 15,16,17 that allow the developer to test if the requirements are met. Easy to add cards? what does that even mean, the definition of easy is debatable and therefore not concretely testable. This will lead to different understandings of the requirement depending on the developer.

Viktor Fällman
Vikfll-0@student.ltu.se

Question 3

Extensibility quality attribute

From an extensibility standpoint, we want high cohesion and low coupling [1]. With preferably a modular design. In the source code, if you were to want to implement new rules, you would have to go into the server to design this, as well as in the client since it handles the bot and the bots rules. This is strong coupling, where different “modules” of the system depend on each other.
Since the design of the first uno game is not modular in the sense that there are packages, it’s modular in the fact that a class file has multiple classes in the example of the server.java file. Here we see low cohesion in the fact that the Server class and the Player class does not belong together.

We can also see that the Client class depends on global variables in server which in turn creates common coupling (Global coupling) [2] which is not a form of low coupling.

Modifiability quality attribute

Modifiability is measured in time to make a change [3]. First of, the card is very hard to read and understand. For the player class, Sufficiency is very low since it does not include all the functionality of a player [4]. Therefore modifying the functionality of the player is not possible by changing the player. There are also coupling from the Server class to the player class that depend on variables and not methods. Using an interface of the player would allow for modifications to be done where the server uses a method and the internal structure of the method could be changed to perhaps return different values that still follow the methods guidelines but can be modified to achieve new functionality.

This goes hand in hand with the fact that the completeness[4] of both the Card and Player classes is minimal. The use of variables from both classes and the fact that the server and client handle comparisons and hinders the completeness of the classes.

Question 4

The redesign of the project can be seen on figures one through 4, figure 1,4 was created before implementation, figure 2,3 used built in UML in intellij to create. It includes UML diagrams with method names, dependencies and how classes interact and a sequence diagram of what should happen if a player wants to play a card.

To adhere to the requirements set 15-17 i've decided to implement a play method within the cards so that they themself can alter the state of the game. This allows the creation of new cards without having to change the alteration on multiple places. I've also implemented basic abstract classes from which to inherit so that functionality of the game can remain the same and it hides the underlying method functionality from the other classes.

Modifying the effect of current cards is done in the play method of every card, this allows for easy updates and fixes on the functionality of the card. The bots extend an Abstract player class and so the internal workings of a bot can be changed without other classes noticing. This eases modifiability of cards and bots.

I've added a comprehensive javadoc that completely explains the methods, classes and packages. There is also a Communication structure file that contains information about how to structure messages being sent from the client to the server. This allows for ease of use and good comprehensibility.

There is dependency injection [5] in the client when creating the player, the creation of the objects is not part of the internal client code and is handled externally and injected into the client. The server uses dependency injection for the game, by creating the rules and gamestate and injecting that into the game. This is done so that the internal UnoGame code does not have to be changed if a new ruleset was to be used. It would only have to adhere to the interface and the internal structure can be changed.

The CardFactory serves as a builder of Cards, where the type of Card is hidden from the other classes, and it will return a type that is needed. I can't specify which if any patterns this implementation follows but it allows for other classes to not worry about the creation of cards and it obfuscates the creation of cards from classes that need to build Cards.

Command pattern [6] is used in the Cards, with the play method, were the internal state of the card is hidden from other classes but still encapsulates all the necessary information to provide its functionality.

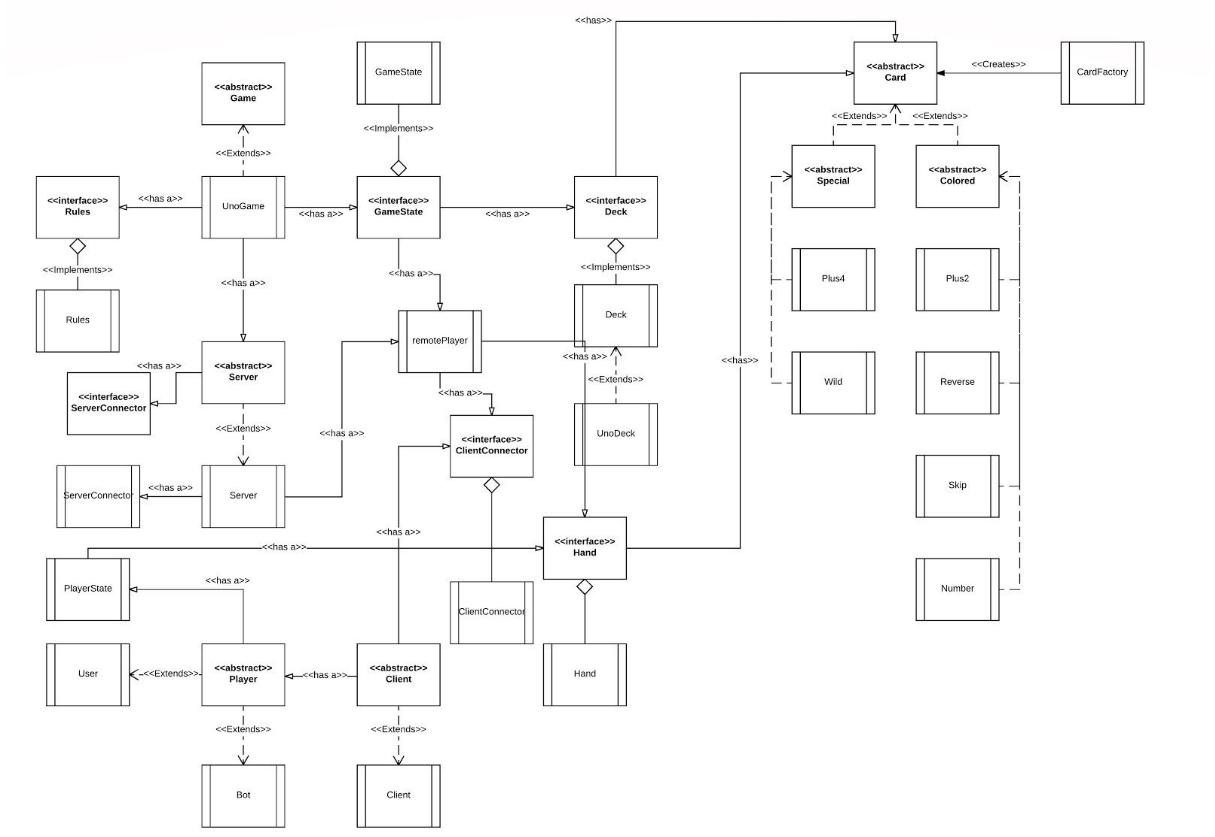

Figure 1. UML of classes and dependencies.

Figure 1 shows an overview of the structure in the project. Here we can see which classes that extend and implement other classes. It also shows which classes has a or communicates with other classes and therefore has a dependency. The implementation has tried to create bridges between classes, so that a user has a hand, that has cards. That way the user can handle its cards without directly communicating with the card classes. The client class also acts as a form of mediator between the user and the network communication with the server, but in hindsight could have implemented structure in the protocol there instead of at the user.

Viktor Fällman
Vikfll-0@student.ltu.se

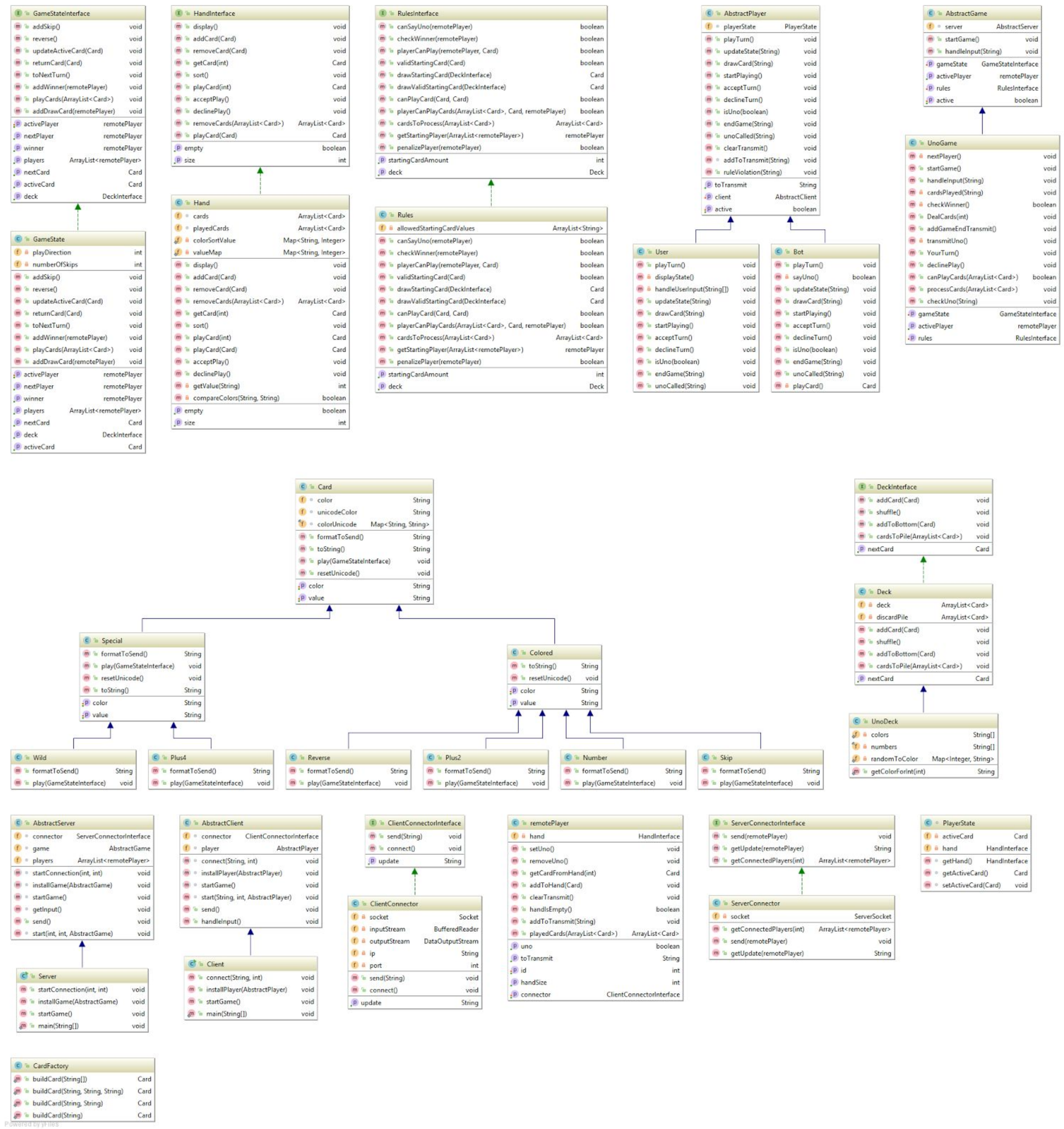

Figure 2. Methods and functionality of classes and their inheritance and extension to other classes.

Overview of the complete project, with methods and shows extending and implementing dependencies. Shows how the cards take use of abstract classes that specify some functionality which in turn take use of the base card class which implements the most basic functionality and acts as a form of interface on which to base cards.

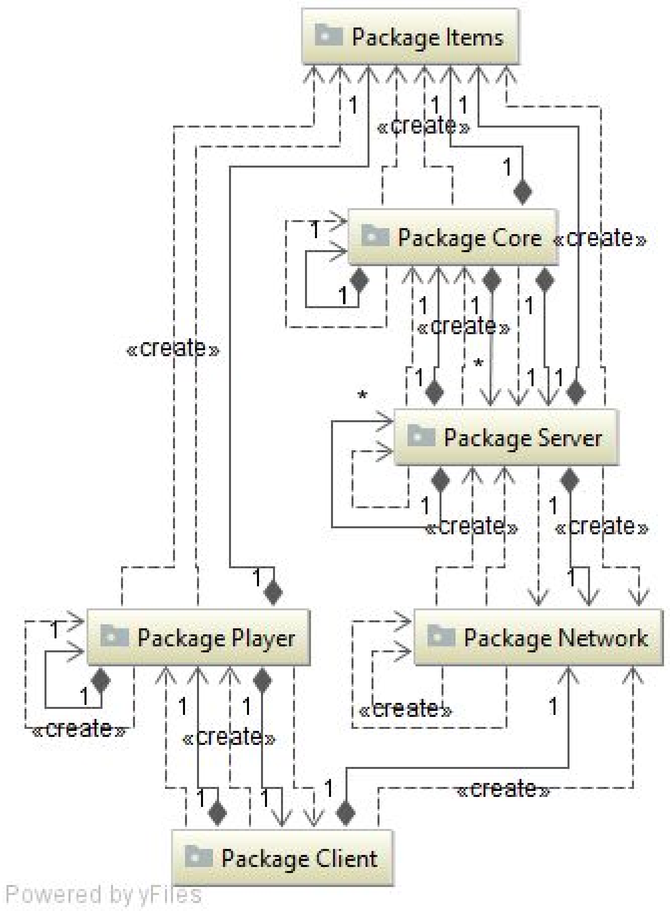

Figure 3. Package dependencies of the UnoGame project.

Package dependencies in the final project, where it shows that there are intramodular dependencies, but a server client infrastructure with a game running and hosts playing would not be feasible without some dependability. What have been done in this implementation is trying to minimize the dependencies and adhere to interfaces that can be replaced with updated functionality.

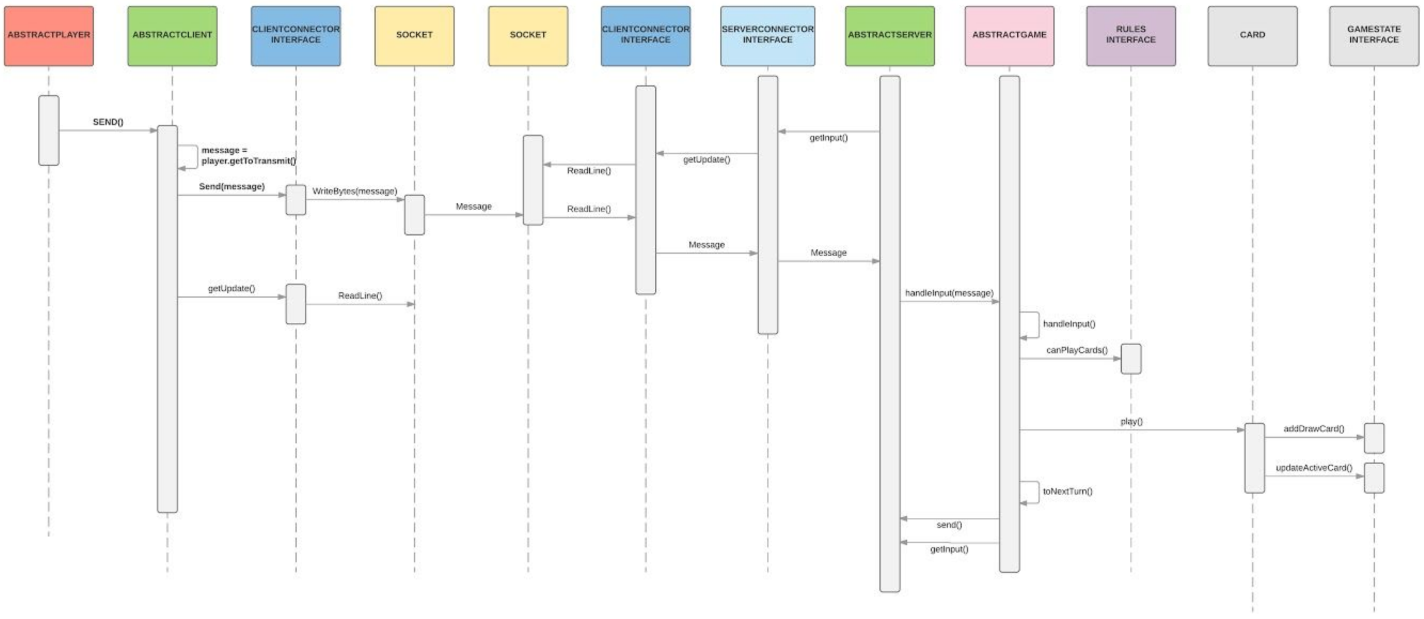

Figure 4. Sequence diagram of a player playing a card, shows how different classes depend and communicate with each other.

Figure 4 displays a diagram of the communication aspect and internal structure in order for the use to play a card. In this figure the message and cards have already been selected by the user/bot and it shows the route the message takes from the abstract player on the client to the abstract game on the server and how the server should handle the message.

Question 5

With the new requirements stated in question 5, the decision to have Rules be a separate class was decided. This allows for ease of use in implementing new rules into the game without changing the rest of the code.

Modifiability on the network functionality led to the creation of the ClientConnectorInterface and ServerConnectorInterface which allows for other types of connections to be created implementing this interface. The bot modifiability was already discussed earlier in the text. As well as the comprehensibility aspect of the implementation. With a complete javadoc and text file describing the protocol on which to base communication on.

Viktor Fällman
Vikfll-0@student.ltu.se

References

[1] https://en.wikipedia.org/wiki/Extensibility
[2] https://en.wikipedia.org/wiki/Coupling_(computer_programming)
[3] http://www.idt.mdh.se/kurser/cdt413/V08/lectures/l3.pdf
[4] https://atomicobject.com/resources/oo-programming/oo-quality
[5] https://en.wikipedia.org/wiki/Dependency_injection
[6] https://en.wikipedia.org/wiki/Command_pattern