m

Master in Advanced Computer Science

Names:

Petter Kogler

Tania Ugarte

Professors:

Lesley De Cruz

Simon De Kock

FLUXX

Course: Programming in Java

**Contents**

[**I. Work process 2**](#_Toc123601039)

[**II. Teamwork 3**](#_Toc123601040)

[**III. Design choices 3**](#_Toc123601041)

[**IV. Special elements 4**](#_Toc123601042)

[**V. Steps to play FLUXX 4**](#_Toc123601043)

**FLUXX**

This report will explain the functionality of a Fluxx game developed in java language for the assignment of Programming in java course.

It is important to mention that this is an adaptation of the real Fluxx game, therefor, it has some of the functionalities not all of them.

Important features to consider:

\* Number of players: Between 0 and 6.

\* Rules: Only the ones related with limitation of cards.

\* Goals: The ones that are composed of card keepers and special goals related to the maximum quantity of card keepers for each player.

# Work process

To develop this software project, mainly a Waterfall Model[[1]](#footnote-1) was followed as it is explained next.

1. **Analysis (Knowing the game):**

To know the game and have an idea about how to develop the project, two actions were taken:

\* First the physical cards have been checked.

\* Then an app of the game was played.

1. **Designing the UML:**

The UML designing process had 3 versions which are described next:

\* First one🡪 The classes and only the main methods included.

\* Second one 🡪 Depuration of classes to avoid redundancy but inclusion of more methods.

\* Before designing the last version, it was necessary to try some methods in coding stage to define what would be the best practical option, and after that process, the final UML version was defined (This is an exception of the engineering software model defined).

1. **Implementation**

It started by creating the classes with the main variables and methods and some basic functionality to test the design.

After making some changes in the UML design the main functionality of the implementation was defined and coded, adding some interesting features to make more personal the game (which will be explained later).

Finally, some extra functionalities were added and some designing ideas were left behind.

In this stage the Waterwall model was combined with a Kanban model in order to apport ideas as a team and moreover to test continuously.

1. **Testing**

As was mentioned above, the testing stage came together with the implementation work, in order to define what was need to solve, but at the same time it was not an impediment to continue working in other features of the code.

A final test process was followed for each member of the group in order to improve the performance of the code and increase the scope of error detection.

# Teamwork

The team worked in a collaborative way, starting early the designing and code stages in order to be able to present some extra features and to provide a personalized work.

The UML is a creation and a definition by the two members of the team.

The coding design was split as follows:

\* Petter: Interaction interface (creation and management features), Rule cards (creation and control, including the rule area), Keeper cards (creation and control), definition of the phases of the game, improvement of the coding style and design, testing.

\* Tania: Goal cards (creation and control), support play phase (coordination and control of winning process in the game methods involved), support discard phase, adding control processes in the program, testing.

The team worked with the support of github, therefor, different changes and suggestions has been done, by the coordination and feedback provision to each other.

# Design choices

After analysing the UML design, the team decided to finally maintain 9 classes in the code, the class with major functionalities will be game and the rest of classes will support the minor processes such as; updating lists, displaying features and so on, the description of the classes is:

1. **Main:** Only to make run the program.
2. **UserInterface:** To have control of the interaction with the players by scanner method.
3. **Game:** This class is defined as the more active one, were most of the principal functionality has been placed.
4. **Player:** This class is the second one in term of managing functionalities for the project.
5. **Card:** For cards, the program it is managing inheritance, where the father class is Card and the child classes are CardKeeper, CardRule, CardGoal.
6. **CardKeeper:** Basic functionality to define the keepers of the game.
7. **CardRule:** The main difference with the other card classes is related to the fact that this class has the attribute “which” to take different names of cards as rules and limit the presence of those in the game, for example:

\* Card Rule🡪 Limit keepers (2,3,4).

\* Card Rule🡪 Limit play (0 which means play all,2,3,4).

\* Card Rule🡪 Limit hand (0,1,2).

\* Card Rule🡪 Limit draw (2,3,4,5).

1. **CardGoal:** (15 combinations of pair keepers) This class is special because is composed for 2 card keepers, this decision was taken on the bases of providing scalability possibilities to the project, on the top of that, the combination of the Card Keepers is made randomly to add some interesting, personal and not predictive features to the game, therefor if in the future it would necessary to have more or less than 15 Card Goals, it would be a matter of only changing an integer variable.

There is not going to have the same combination of pairs per game (the program is controlling that).

Another type of card goals was created, the ones that allows to choose a winner when all the players have a defined amount of card keepers, for that, this class manage polymorphism in the constructor.

1. **RuleArea:** This class will have the List of Rule Cards that are in play. It works with the class CardRule, therefor it is just allowed to have in the rule area a max of 4 cards for each type of Card rule (example: It is not possible to have 2 different limit draw cards because there is a contradiction).

# Special elements

As it was mentioned before the designing includes the possibility of define more card goals by updating only one variable.

# Steps to play FLUXX

* 1. **Choose the number of players** 🡪 **between 0 and 6.**
  2. **Write down the nick name of each player**🡪 It is not allowed to have repetitive nicknames for a single game.
  3. **The Tutorial is displayed:**

\* At all times, typing r shows the rules, k the keepers, g the goals and h your hand cards.

\* Selecting cards will be done by typing the according number.

\* Typing 'help' will display all possible input options.

Turn for “Nickname” player:

🡪 Type anything except k, g, r, h and 'help' to continue (this will start the turn).

* 1. **Start the turn**

Per each turn(automatically in the order that the players register the nicknames), the player will see:

“Nickname”, you must play “Number of cards to be played”

For example:

Your hand cards are:

0: Rule Play 4

1: Keeper Death

2: Keeper The Moon

3: Rule Play 2

Choose a card to play by entering its number.

By choosing a card the play continues with its normal dynamic till the goal is accomplished by a player, when the goal is reached automatically a message similar to next is displayed:

You played card: “Name of goal card”

Player “Nickname” wins!!!

* 1. **Case of Rules**

Limit cards in hand and in keeper list will be immediately updated by each player when the corresponding cards are been selected, so that to have a winner with more keepers that the limit will be avoid.

1. The waterfall model is a software engineering model in which tasks are executed sequentially, starting from the top with feasibility and flowing down through various tasks with implementation into the live environment.

   Source: https://www.tutorialspoint.com/sdlc/sdlc\_waterfall\_model.htm [↑](#footnote-ref-1)