Master in Advanced Computer Science

FLUXX

Course: Programming in Java

Professors:
Lesley De Cruz
Simon De Kock

Names:
Peter Kögler
Tania Ugarte

Contents

FLUXX

This report explains the functionality of a Fluxx game developed in java language for the assignment of the Programming in Java course.
It is important to mention that this is an adaptation of the real Fluxx game, meaning it does not provide the game's full functionality.
Important features to consider are:

* Number of players: Between 2 and 6 players
* Rules: Standard rules, i.e. limits for hand cards, keeper cards, draw and play phase
* Goals: Regular goals combining 2 keepers as well as special goals related to the quantity of keepers in a player's keeper area

I. Work process

To develop this software project, mainly a Waterfall Model[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 were checked.
   * Then, an app of the game was played.
2. **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. After that process, the final UML version was defined (This is an exception of the engineering software model defined).
3. **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 the game more personal (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.
4. **Testing**
   As mentioned above, the testing stage came together with the implementation work in order to define the features whose realization is still pending. This was not found to be an impediment to continue working on other features of the code.
   A final test process followed for each member of the group in order to improve the performance of the code and increase the scope of error detection.

[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

II. Teamwork

The team worked in a collaborative way, starting early with 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:

* Peter: Interaction interface (creation and management of 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, meaning the development went forth in a gradual, coordinated process with a constant flow of feedback and suggestions.

III. Design choices

After analyzing 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:** Start the program.
2. **UserInterface:** Interaction with the players using java.util.Scanner via case specific methods.
3. **Game:** Initialization and control center of the game. Holds and organizes instances of all other classes, except Main.
4. **Player:** 2nd most comprehensive class of the project. Organizes a player's hand and keeper cards and a prominent amount of user interaction.
5. **Card:** Abstract class that is parent to CardKeeper, CardRule and CardGoal. Makes possible to incorporate the different card types in the same structure allowing usage of intuitive structures like *deck* and *discardPile* and polymorphism.
6. **CardKeeper:** Rudimentary functionality, defines the keeper cards.
7. **CardRule:** Defines all 4 categories of rule cards. Category is defined by a String variable. Each instance of CardRule provides a limit in form of an integer. Possible values:
   * Card Rule→ Keeper limit (2, 3, 4)
   * Card Rule→ Play limit (2, 3, 4, 0 representing "play all")
   * Card Rule→ Hand limit (0, 1, 2)
   * Card Rule→ Draw limit (2, 3, 4, 5)
8. **CardGoal:** (15 unique pairs of keepers) Each instance is composed of 2 instances of CardKeeper. This provides excellent scalability by the means of changing an integer variable. The keeper combinations are randomized with uniqueness ensured, giving the game a personal and unpredictable touch. This also allows the straight forward

implementation of a check win method by calling contains() on a player's List<CardKeeper>.

CardGoal further implements special goals by using a deviant constructor that initializes an integer representing the necessary quantity of keepers to win.

9. **RuleArea:** Holds the 4 categories of rule cards by using a hash map. Rule cards of the same category will replace each other.

IV. Special elements

As mentioned before the design includes the possibility scale card goals by updating only one variable. Further, special goals were implemented. Detailed measures were taken to ensure functionality identical to the real Fluxx game (such as hand and keeper limits going into effect immediately except for the player whose turn it is and the immediate adaptation of new play and draw rules, taking into account the number of cards already drawn/played).

V. Steps to play FLUXX

1. **Choose the number of players → between 2 and 6.**
2. **Provide the nick name of each player→** It is not allowed to have duplicate 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).

4. **Game on:**
   In 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.

   At all times, the current rules are enforced. Discarding hand cards and keepers is performed through an interface similar to choosing a card. By choosing a card, the routine continues with its normal dynamic until the goal is accomplished by a player (exclusively), in which case the routine is stopped and the following message is printed:

   GAME OVER
   Player "Nickname" wins!!!