Project report - Mystic tarot

Aurelien Robineau

December 20, 2020

Preamble

This is my report for the JAVA *Mystic tarot* project. This project was realized for the professional bachelor's degree *Web and Mobile Project* of *Sorbonne University*.

The intent of this report is to explain the architectural choices I made.

1 Analysis and modelling

1.1 The `Card` class

There are two types of card in the game of tarot: Minor Arcanas and Major Arcanas. Most of the time, the mystic tarot only uses the Major Arcanas, so I chose to implement only one `Card` class for the Major Arcanas. It would be easy though to add Minor Arcanas just by making the current class abstract and having a new class for each type of card.

Major Arcanas always have a number and a name. They must also have an image to represent the physical card. So cards will have the following properties:

- a `number` that must be unique. It is the ID of the card
- a `name`
- a `imagePath` to the file of the card's image

First I also put a `description` property but I realized later it was unused so I removed it.

1.2 Action classes

I chose to add a class per action on cards so actions are all independant and it is easy to add new ones.

These classes - that I will call *action classes* - will implement the `CardAction` interface. This interface should have two methods: one that actually do the action and another one that asks in the console informations about the action to the user. However, I only added the `openInConsole()` that opens a console interface because action classes were too differents (actions with a different number of arguments, some have sereval actions, etc...). So I could not have a unique `performAction(Card card)` method as I first intended.

1.3 The `CardManager` class

Any action on cards will go through another class, the `CardManager`, wich calls itself methods from the action classes. This way action classes are hidden and if a change has to be made, one would only have to adapt the `CardManager` and

not change anything in the rest of the code. The class contains a method that displays a menu in the console so the user can choose an action. It is responsible for the cards so it contains the list of cards.

As there is only one card deck in the game because there is a unique player, the `CardManager` is a singleton. This prevents to instanciate several `CardManager` for nothing and makes it usable globally via the `getInstance()` method, so we do not have to pass it as an argument to each method. One could argue that the game may have several players. In this case, we could remove the singleton pattern and rename the `CardManager` class to `Deck` for exemple. The `Deck` class could extend the `ArrayList` class for more convenience. Then each player would have a `deck` property and its own `Deck` instance. Since I only need one deck I will use the `CardManager` solution.

1.4 The `UserInput` class

I added the `UserInput` class later in the project because I realized I was doing the same verifications on the user input sereval times. For exemple I must check that the user entered an integer or an existing card number in different action classes.

The `UserInput` class provides methods to ask the user to enter a value while the value does not pass the method validation.

For exemple the `getInteger()` method asks the user to enter a value while it is not a integer.

2 Basic card system

It is requested to be able to add and delete cards, so I added to action classes:

- The `CardCreator` class
- The `CardDeletor` class

The `CardCreator` has a method that asks the user the values for the new card and another one that creates the card from these values.

The `CardDeletor` has a method that asks the user wich card to delete and another one that actually delete the card.

All action classes are protected so can only be used in the `card` package, so the only entry point for other packages is the `CardManager`.

3 Extension and research

3.1 Editing cards

To edit cards, I just need to add a new action class named `CardEditor` wich has a method that asks the user wich card to edit and then the new card values. It has another method that updates the card values and saves it.

3.2 Displaying cards

To display the card list there is no need for an action class since the `CardManager` constains the list of cards. I overwrote the `toString()` method in the `Card` class and added a method in the `CardManager` that simply loops through the cards and displays them. However for more advanced display we could create a `CardDisplayer` action class.

3.3 Searching cards

For the research feature I added another action class named `CardSearcher`. It has a list of cards to search in and sereval search methods:

- one for searching a card by its `number`
- one for searching a card by its `name`
- one for searching cards by there `number` and `name` in the same time

3.4 Adding a image

For the image I already had the `imagePath` property on cards so I just added a new field to be asked to the user in the `CardCreator` and `CardEditor` classes. The user must give the path to the image. I do not yet have an `image` property containing the image file because the image will not be rendered for now.

4 Saving cards

4.1 Classic serialization

I added a new class named `CardSerializer` to serialize and save cards to files. The constructor takes a `Card` as an argument that will be the card to serialize. The `CardSerializer` has two methods to manage this card. One to create the card file, the other one to delete it. It also has a static method to load all cards from serial files.

Files are saved in the `data/cards/serialized` directory. They are named `<card_number>.serial`. Each file in this directory will be loaded when starting the application. Invalid or old files found when leaving the app will be deleted.

4.2 JSON serialization

To save cards as JSON I added the exact same methods but with the JSON logic using the Gson API.

JSON files are saved in the same directory than the binary files. They are named `<card_number>.json`.
Both binary and JSON serialization are still available. It is possible to switch method by using the `SERIALIZATION_METHOD` property of the `CardSerializer` class.

5 Graphical User Interface

5.1 `gui` package

The card management files are all in the `card` package. To seperate clearly the GUI from the card management I added a new package `gui`. Since each action class is `protected`, the GUI can only rely on the `CardManager` singleton to interact with cards. Indeed the GUI does not have to know the logic behing the scenes, but must only call simple methods to manage cards.

5.2 The `GuiManager` class

For the same reasons I used a `CardManager` class, I created a `GuiMananger` class. Any action on the GUI will use this class wich contains the list of cards to display. So no action will be done directly into the action listeners. Again, this separation makes maintenance more easy and the application more scalable.

Since there is only one list of cards to display, the `GuiManager` is also a singleton and can easily be used from anywhere in the GUI.

5.3 Package architecture

We will need frames to interact with the user. Theses frames will contain components (buttons, inputs, file inputs, etc. . .) wich will trigger actions via action listeners.
So the GUI package will have three sub-packages:

- `gui/frames`
- `gui/components`
- `gui/listeners`

5.4 The main frame

I decided that the main frame will have:

- an area to display the cards
- a button to create a new card
- a text input to search cards

The input will update the shown cards everytime the user types in it and will show all cards if empty.

Each card will be displayed in its own panel with buttons to delete and edit the card.

The create and edit card forms will be displayed in there own frames.

5.5 Card images

When adding a image to a card, the image is copied to the `data/cards/images` directory. The card `imagePath` is set to the relative path of the new image so cards can be shared on different devices. When editing or deleting a card, the old image is removed from the directory.

5.6 Switching to console interface

The console interface is still available. It is easy to switch from a interface to the other with the `INTERFACE` property of the `MysticTarot` class.