SENG301 Assignment 4
Design Patterns

SENG301 Software Engineering II

Neville Churcher Morgan English Fabian Gilson

16th May 2023

Learning objectives

By completing this assignment, students will demonstrate their knowledge about:

- how to identify design patterns from a code base;
- how to justify the usage of particular design patterns to fulfil particular goals;
- how to retro-document a UML Class Diagram from a code base;
- how to implement new features by means of refactoring exisiting code to use design patterns;
- how to refactor and extend a code-base following the identified patterns.

To this end, students will use the following technologies:

- the *Java* programming language with `gradle` dependency and building tool;
- the *Cucumber* acceptance test framework;
- the *Mockito* stubing framework;
- the Apache *Log4j2* logging facility;
- UML class diagram, either using a tool like *Plantuml* or by hand.

Next to this handout, a `zip` archive is given where the code base used for *Labs 4* to *6* and for *Assignment 3* has been updated for the purpose of this assignment.

1 Domain, story and acceptance criteria

This Assignment builds upon the *Yu-Gi-Oh! Clone App* used during term 1 labs and the previous assignment. The code base from *Assignment 3* has been extended and modified to fit the purpose of this summative assignment. The user stories of current code base are listed in Section 1.1. A walkthrough of the code base is given in Section 1.2. Your tasks are described in Section 2 and the submission rules are stated in Section 3.

A simplified class diagram is reproduced in Figure 1. The diagram depicts the domain model and basic functionality of the application. **Note that several classes are missing or not fully described on purpose.**

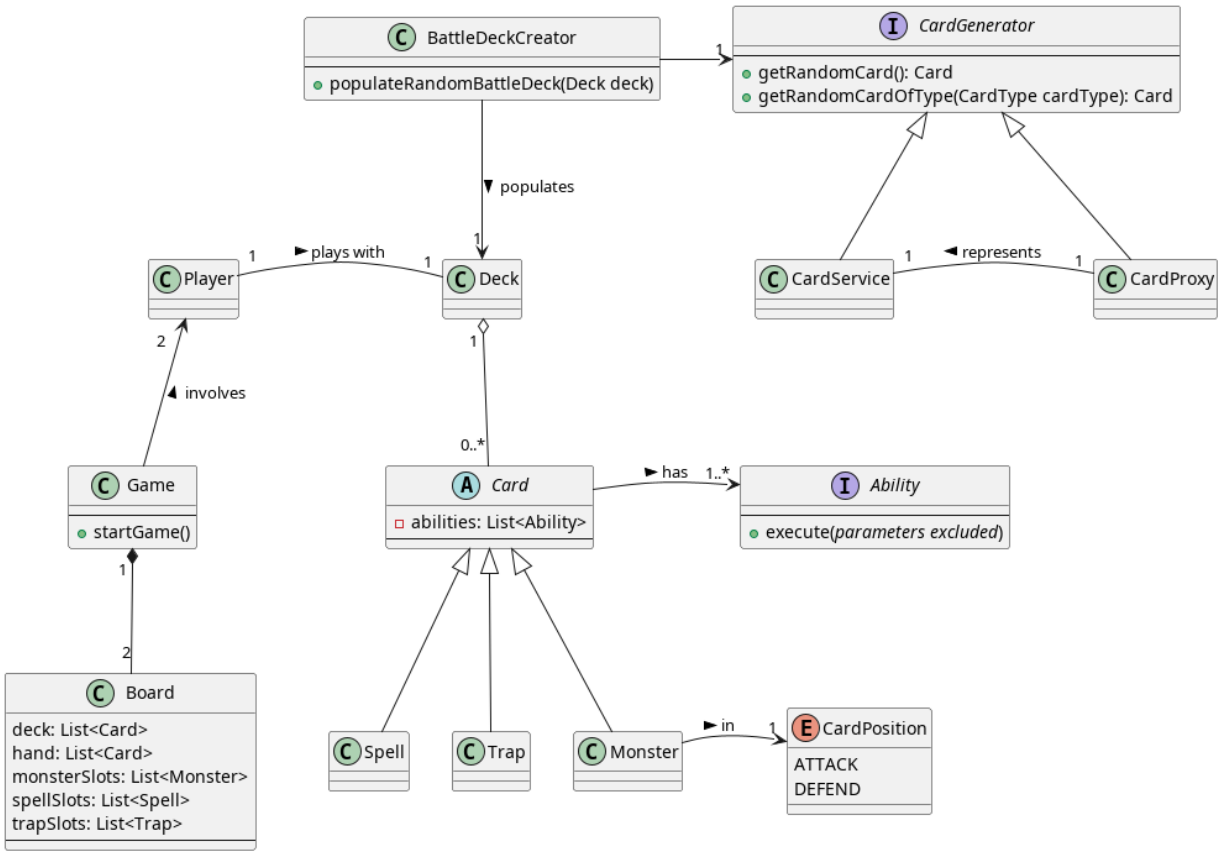

Figure 1: Partial class diagram of the *Yu-Gi-Oh! Clone App*.

1.1 User stories of current code base

This list expands on the user stories defined for the labs (with *Alex*, our only persona, refer to the lab handouts for a complete description of the persona). Note that some scenarios and their implementation have been slightly adapted to match the new code.

U1 **As *Alex*, I want to create a user within the application so that I can use other functionality.**
- AC.1 A player has a unique non-empty name and a specific AI play style
- AC.2 A player name cannot contain non-alphanumeric or numeric-only values
- AC.3 A player's play style must be selected from one of the existing types

U2 **As *Alex*, I want to create a battle deck so that I can fight opponents.**
- AC.1 I can create a random battle deck of 20 cards.
- AC.2 A valid battle deck is composed of at least 5 monsters, 3 traps and 3 spells.
- AC.3 I can create a battle deck with custom card type ratios.

U3 **As *Alex*, I want to be able to be able to battle against other players**
- AC.1 I can play a game against another player
- AC.2 Each player in the game must have a valid battle deck prepared to play
- AC.3 A game is concluded once all turns are up and the winner is decided as the player with the most total Monster health remaining

In the code base we give to you, **all three user stories have been implemented together with their acceptance tests**.

1.2 Walkthrough the code base

The code base used for *Labs 4* to *6 (and assignment 3)* has been updated removing some functionality to provide a more streamlined code base and some new features and acceptance tests added. You should already be familiar with most of the code included.

1.2.1 README

The `zip` archive with the code also contains a `README` and a `LICENSE` file that you need to consult prior going deeper into the code. This `README` describes the content of the archive and how to run the project. The code is also extensively documented so take some time to read the *Javadoc*.

1.2.2 Gradle configuration file

Take some time to review the `build.gradle` file. You should be familiar with its content as it is a complete version of what was needed to complete *Lab 6*. **You are required to keep this file untouched, failing to do so would prevent us from marking your assignment, and you will be awarded 0 marks for the assignment.**

1.2.3 Model layer

In this package, you will find the domain model presented in Figure 1. Extensive documentation is provided with pointers to external references, so you should take some time to understand how this works.

1.2.4 Accessor layer

Accessors have been implemented for `Deck` and `Player`. Note that these accessors no longer deal with Hibernate persistence.

1.2.5 Game package

This package contains many of the domain rules and functionality of playing a Yu-Gi-Oh!-like card game. Note that **The implementation is not perfect, with some liberty being taken to make the code simple enough for this assignment**.

1.2.6 Yu-Gi-Oh! card API

From *Lab 6*, you learned how to use an external API to retrieve random cards using a simple HTTP server and transparent JSON deserialization. These classes make use of the *Proxy* pattern, see `https://refactoring.guru/design-patterns/proxy` for more details.

1.2.7 Automated acceptance tests

You can take a look at the three `feature` files under `app/src/test/resources/uc/seng301/cardbattler/asg4/cucumber` to see the scenarios covering all acceptance criteria for U1, U2 and U3. The test code is placed under `app/src/test/java/uc/seng301/cardbattler/asg4/cucumber` (for U1, U2, and U3).

1.2.8 Command line interface (CLI)

After having taken some time to review the code base, you can run the project to get a deeper understanding of its existing features. Check the `README` file for explanations on how to run the CLI code (i.e. `$ ./gradlew --console=plain`

`--quiet run`). The `main` method is placed in the `App` class. For help running different commands, use the command *help* to see a list of all commands accepted by the CLI. When running the `App`, all commands are displayed.

2 Your tasks

2.1 Task 1 - Identify two patterns [30 MARKS]

The code base you received contains two design patterns from the famous *"Gang of Four"* [**?**] (GoF) book that you need to identify in this second task.

For each pattern, you will need to (awards **2x15 MARKS**):

- name the pattern you found and give a brief summary of its goal **in the current code[2 MARKS]**
- justify how this pattern is instantiated in the code, *i.e.* by mapping pattern components to Java classes and methods using a table (as done in class), as follows: **[8 MARKS]**
  - map *GoF* patterns **components** to their implementation **Java classes**
  - map the relevant **pattern methods** to their **corresponding implementation** in the code (note that some non-critical pattern methods may not be present in the code)
- draw the UML diagram of the **actual implementation** of the pattern (**with the relevant methods only**) by following a similar structure to the pattern, but using the actual class names in the code base; **note that** it is possible that an association in the *GoF* pattern is implemented in a slightly different way, but you need to draw the actual implementation **[5 MARKS]**

Answers to above questions must be provided into the `ANSWERS.md` file under *"Task 1"*. UML Class Diagrams [**?**, chap.11] will need to be referred to by specifying the name of the image file where asked in that `ANSWERS.md` file. These UML diagrams must be placed under the dedicated `diagrams` folder (where you will find a copy of Figure 1). An example is given in the `ANSWERS` file with the proxy pattern depicted in Figure 1.

2.2 Task 2 - Retro-document the design of the existing code base [5 MARKS]

You are asked to retro-document the overall design in the form of a UML Class Diagram [**?**, chap.11]. **No changes in the source code is required for this task.** To this end, you can build on:

- the domain model given in Figure 1;
- the walkthrough you conducted, helped by Section 1.2;
- the patterns you identified in task 1.

The full diagram must be put in the `diagrams` folder that you used already in *Task 1*. You need to add its name in the `ANSWERS.md` file under *"Task 1"*.

Note that **generated diagrams** (e.g., from automatic extraction from *Intellij*) **will not be accepted** for this task. When marking your diagram, we will look at:

- the syntactic validity, i.e. is it a valid UML 2.5 class diagram;
- the semantic validity, i.e. are all classes and associations semantically valid (e.g., no wrong types of associations or non-existent ones), are **all associations' multiplicities** present and valid, are all associations named where needed (i.e. not necessary for aggregation, composition, and inheritance);
- the completeness, i.e. are all classes of the code base present (but you may consider to hide/shorten some methods for readability reasons, as we did);
- the readability, e.g., **meaningful names on associations**, readable layout.

2.3 Task 3 - Refactor two areas of code suboptimal design to use GoF patterns [30 MARKS]

Two parts of the code are annotated with `TODO` comments. You are expected to refactor these two places with a design pattern **discussed in class**.

You are expected to update the following areas of code with an appropriate design pattern:

- **Task 3.1** Creating cards from the `CardResponse` Jackson (JSON-library) annotated class.
- **Task 3.2** Handling different AI types for the game found in `PlayStyle`.

In a similar fashion to *Task 1*, you are expected to provide the following details under *"Task 3"* in the `ANSWERS.md` file. We give the alotted marks per sub-question, for a total of **2 times 5 marks**:

- name the pattern you implemented and give a brief summary of its goal in this code, including how the original implementation was improved with this pattern **[2 MARKS]**
- justify how this pattern is instantiated in the code, *i.e.* using a table (as done in class and for *Task 2*) **[3 MARKS]**
  - map *GoF* patterns components to their implementation Java classes
  - map the relevant pattern methods to their corresponding implementation in your new code

When assessing your tasks 3.1. and 3.2., we will verify that **2 times [10 MARKS]**:

- the expected refactoring is implemented correctly, i.e. **all existing ACs pass without modification [5 MARKS]**;
- you **respected the pattern** you selected to the letter **[5 MARKS]**;
- your code runs (**unexpected crashes in the code would award 0 marks**, i.e. 0 out of 10 marks, regardless you implemented the pattern properly or not).

2.4 Task 4 (BONUS) - Implement randomized card abilities acceptance[5 MARKS]

U4 **As *Alex*, I want to have cards with a diverse range of abilities.**
AC.1 A card must have at least 1 ability and at most 3
AC.2 A card's abilities are assigned at random

As a bonus, you can implement acceptance tests for above story U4 (awards **[2.5 MARKS]** for each AC implemented in code with appropriate **passing** cucumber tests). The name of the files (`feature` and Java class) must be specified in the `ANSWERS.md` file under *"Task 4"*.

- Extend your implementation of the design pattern you chose for card creation to randomly assign card abilities following AC4.
- create a new `feature` file for that story (remember to respect the naming convention);
- translate the two acceptance criteria of U4 in *Gherkin* syntax (i.e. *Cucumber* scenario);
- implement the acceptance test for each of the scenario into a new test class;

When assessing this task, we will verify that:

- you have adequately translated the acceptance criteria, i.e. the **behavioural semantic** of the *Gherkin* scenario **is identical to the English version** given in Section 1.1;
- the automated acceptance tests effectively **checks the expected behaviour expressed in the AC**[1];
- the tests are self-contained, readable and follow the design practices taught in the course (e.g., Lecture on testing and additional material);
- the tests pass (**a failing test or a test that would not exactly translate the AC into *Cucumber* award 0 marks for that AC**).

[1] This may include the use of *Scenario Outlines*

3 Submission

You are expected to submit a `.zip` archive of your project named `seng301_asg4_lastname-firstname.zip` on Learn by **Friday 2 June 6PM**[2]. **No other format than `.zip` will be accepted**. Your archive must contain:

1. the updated source code (please remove the `.gradle`, `bin`, `build` and `log` folders, but keep the `gradle` - with no dots - folder); **do not remove the `build.gradle` file or we will not assess your assignment and you will be awarded 0 marks**;
2. your answers to the questions in the given `ANSWERS.md` file (you are **required to keep the file format intact**);
3. the UML Class diagrams in `.png`, `.pdf` or `.jpg` format under the `diagrams` folder. All diagrams must be **images** (e.g., `jpg`, `png`, `pdf`), but you can include the `.puml` if you like). If you use pen & paper photos, please ensure we can read them, it is **your responsibility** to make sure they are legible.

Your code:

1. **may not** import other libraries / dependencies than the ones currently in the `build.gradle` file;
2. **will not be evaluated** if it does not build straight away or fail to comply to 1. (e.g., no or wrong `build.gradle` file supplied);
3. **will be** passed through an advanced clone detection tools (*i.e.* Txl/NiCad) that proved to be performing well on sophistically plagiarised code earlier.

The marking rubric is specified next to each task (overall **65 marks**), but is summarised as follow:

Task 1 name **2x2 marks**, map **2x8 marks** and draw the patterns **2x5 marks**
Task 2 draw the full UML diagram **5 marks**
Task 3 name **2x0.5 mark**, explain **2x1.5 mark**, map **2x3 marks** and refactor the indicated areas of code using a GoF pattern **2x10 marks**
Task 4 correct and functional implementation of U4 ACs **2x2.5 mark** (BONUS)

4 Tools

You only need the following tools to run and develop your program if you work on your own computer:

- an IDE, *e.g., IntelliJ IDEA* `https://www.jetbrains.com/idea/download/`
- *OpenJDK Java SDK* `https://openjdk.java.net/install/`
- for *Windows* users, setting up your environment variables for Java to be recognised: `https://stackoverflow.com/a/52531093/5463498`

The code you receive already contains the minimal binaries for `gradle` that will manage the dependencies for you. Please refer to the `README` shipped with the code for more details. You should be familiar with the process as it is the same as for term 1 labs. If the `gradlew` command has issues (unix), you can recreate the wrapper by running `$ gradle wrapper` (from your local install), as we showed you in *Lab 3*.

We suggest you use an embedded markdown editor (in *Intellij IDEA* or *VSCode*) to edit your `ANSWERS.md` file, or another mark down editor of your choice.

As UML drawing tools, we recommend the following web- and textual-based drawing *PlantUML* (as used in labs during term 2), `https://www.planttext.com/`. Plugins exist for *IntelliJ IDEA*: `https://plugins.jetbrains.com/plugin/7017-plantuml-integration` and *VSCode*: `https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml`.

[2]There is a 4 day grace period (i.e. extension with no penalty by default). No further extension will be granted, unless special consideration.