SENG301 Assignment 4
Design Patterns

SENG301 Software Engineering II

Neville Churcher Morgan English Fabian Gilson

8th May 2024

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 *Super Auto Pets 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.

An incomplete 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, some multiplicities are missing, and some classes are not fully described on purpose.** We expect you to augment this diagram in you Task 2.2.

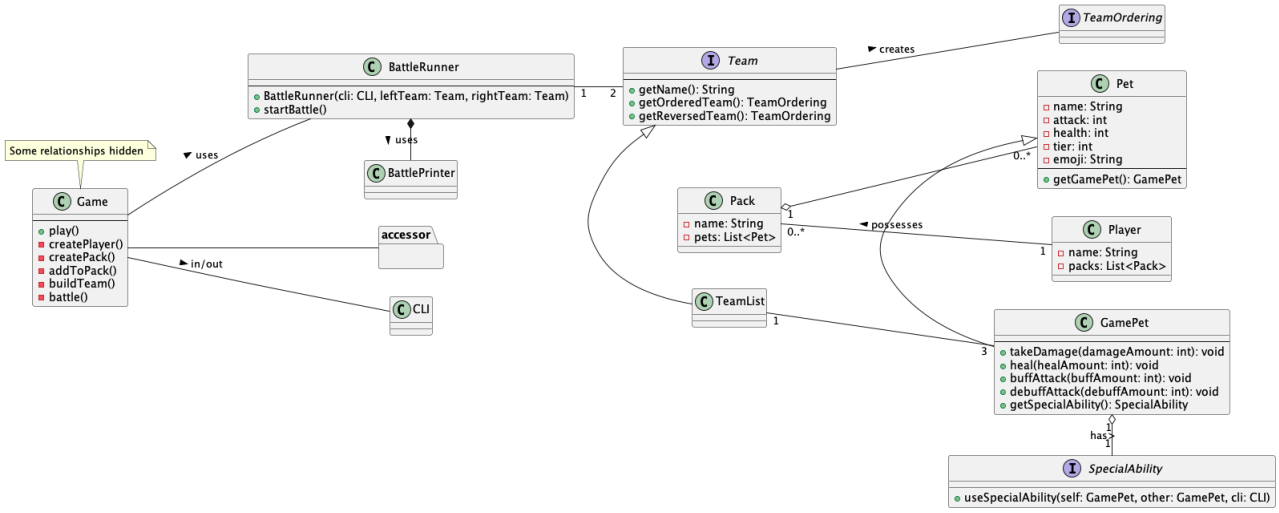

Figure 1: Incomplete class diagram of the *Super Auto Pets Clone App*.

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 pet so that I can use it in a pack.
- AC.1 A pet has a unique non-empty name, a strictly positive attack and health stats.
- AC.2 A pet name can contain alphabetic characters only.
- AC.3 A pet cannot have a negative or zero values for the attack or life stats.

U2 As *Alex* I want to create a pack so that I can build a set of pets to use in battle.
- AC.1 A pack has a unique, non-empty alphanumeric name.
- AC.2 A pack cannot have a numeric-only name, or special characters in its name.
- AC.3 A pack must be able to store zero to many pets.

U3 As Alex, I want to draw random pets from an external API so that I can build a pack from them.
- AC.1 I can draw a random pet that has valid name, tier, and attack and health stats.
- AC.2 If I find a suitable pet, I can add the pet to my pack.
- AC.3 I can decide to ignore the pet and not add it to my pack.
- AC.4 A pack cannot contain the same pet twice.

U4 As *Alex*, I want to create a team from the pets in one of my packs so that I can use it in a battle.
- AC.1 A pack must contain at least one pet to build a team with.
- AC.2 When building a team there will be 5 options randomly selected from my pack.
- AC.3 When building a team I must select 3 options.
- AC.4 When building a team I must select 3 unique options.
- AC.5 When building a team and I select 3 unique options then my team is ordered the same way I entered my options.
- AC.6 When building a team and I select 3 options of all the same pet type they are different copies from those in my pack.

U5 As *Alex*, I want to be able to simulate a battle between two teams so that I can have my pets fighting.
- AC.1 I can start a battle between two distinct teams.
- AC.2 A battle can not last longer than 20 turns.
- AC.3 The team who loses all their pets first loses the battle.

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

U1-U4.

1.2 Walkthrough the code base

The code base used for *Labs 4* to *6 (and assignment 3)* has been updated. We changed some functionality to provide a more streamlined code base and adding some new features and acceptance tests to comply with this assignment. You should already be familiar with most of the code base 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 from the labs for `Player`, `Pack`, and `Pet` are included with few if any changes.

1.2.5 Battle package

This package contains many of the domain rules and functionality for implementing a SAP-like battling mechanic. Note that **the implementation is not perfect, with some liberty being taken to make the code simple enough for this assignment**.

1.2.6 Pet 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. **This implementation of the *Proxy* pattern will not be accepted as an answer for Task 1** and has been documented as an example answer in the `ANSWERS.md` file.

1.2.7 Automated acceptance tests

You can take a look at the `feature` files under `app/src/test/resources/uc/seng301/petbattler/asg4/cucumber` to see the scenarios covering all acceptance criteria for U1, U2, U3, and U4. The test code is placed under `app/src/test/java/uc/seng301/petbattler/asg4/cucumber`.

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 three patterns [45 MARKS]

The code base you received contains three design patterns from the famous *"Gang of Four"* [1] (GoF) book that you need to identify in this second task. They all mostly follow the styles described in the GoF book, the Design Patterns Reference Card[1] and the Refactoring Guru[2] website, but some deviations may have been made to fit the code base. When documenting your patterns, you can use any of the stereotypes used in the Reference Card or Refactoring Guru website.

For each pattern, you will need to (awards **15 MARKS** per pattern):

- 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 and adding the pattern **stereotypes** (see the `Proxy` example in given archive); **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 [2, 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.md` file with the proxy pattern (left out from Figure 1). Take the time to read the additional note about the UML diagram in that example answer.

2.2 Task 2 - Retro-document the design of the existing code base [10 MARKS]

You are asked to retro-document the overall design (including all patterns) in the form of a UML Class Diagram [2, 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 pattern we identified for you as an example for task 1;
- 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 2"*.

[1] See `http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf`
[2] See `https://refactoring.guru`

Note that **generated diagrams** (e.g., from automatic extraction from *IntelliJ*) **will not be accepted** for this task and will award 0 marks 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, but all simple associations must have a name and multiplicities;
- the presence of all stereotypes for all identified patterns from task 1;
- 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 - Implement a new feature with one of the GoF patterns [25 MARKS]

Part of the battling implementation given is annotated with `TODO` comments, see `BattleRunner` class. You are expected to implement a *undo / redo* feature for battle rounds **using a design pattern discussed in class**.

In a similar fashion to *Task 1*, you are expected to provide the following details under *"Task 3"* in the `ANSWERS.md` file:

- name the pattern you implemented **[1 MARK]** and give a brief summary of its goal in this code, including how the pattern fits the expected implementation **[3 MARKS]**
- justify how this pattern is instantiated in the code, *i.e.* using a table (as done in class and for *Task 2*) **[6 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 the implementation of task 3, we will verify that:

- the expected feature is implemented correctly, i.e. **all existing ACs pass without modification [10 MARKS]**;
- you **respected the pattern** you selected to the letter, i.e. as you described it **[5 MARKS]**.

Note that your code must execute with no problems, i.e. **unexpected crashes or uncaught exceptions would award 0 marks** out of 15 marks (for the implementation part of the task), regardless you implemented the pattern properly or not.

2.4 Task 4 (BONUS) - Implement U5 acceptance [7.5 MARKS]

U5 As *Alex*, I want to be able to simulate a battle between two teams so that I can have my pets fighting.
AC.1 I can start a battle between two distinct teams.
AC.2 A battle can not last longer than 20 turns.
AC.3 The team who loses all their pets first loses the battle.

As a bonus, you can implement acceptance tests for above story U5 (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 three acceptance criteria of U5 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**[3];
- 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* awards 0 marks for that AC**).

3 Submission

You are expected to submit a `.zip` archive of your project named `seng301_asg4_lastname-firstname.zip` on Learn by **Friday 31 May 6PM**. There is a 3 day grace period (i.e. extension with no penalty by default). No further extension will be granted, unless special consideration. **Submission later than the 3 June 6PM will not be accepted. 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` folder** - with no dots -);
2. make sure you keep the **`build.gradle`** file or we will not assess your assignment, and you will be awarded 0 marks;
3. your answers to the questions in the given `ANSWERS.md` file (you are **required to keep the file format intact**);
4. 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 **75 marks**), but is summarised as follow:

Task 1 name **3x2 marks**, map **3x8 marks** and draw the patterns **3x5 marks**; total **45 marks**
Task 2 draw the full UML diagram; total **10 marks**
Task 3 name **1 mark**, explain **3 marks**, map **6 marks** and implement the new feature using a GoF pattern **15 marks**; total **25 marks**
Task 4 correct and functional implementation of U5 ACs **3x2.5 mark** (BONUS); total **7.5 marks**
Overall: **80 marks** maximum

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

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

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 1*.

We suggest you use an embedded markdown editor (in *Intellij IDEA* or *VSCode*) to edit your `ANSWERS.md` file, or another markdown 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://plantuml-editor.kkeisuke.com/`. Plugins exist for *IntelliJ IDEA*: `https://plugins.jetbrains.com/plugin/7017-plantuml-integration` and *VSCode*: `https://marketplace.visualstudio.com/items?itemName=jebbs.plantuml`.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] Object Management Group, "OMG unified modeling language (OMG UML), version 2.5.1," https://www.omg.org/spec/UML/2.5.1/PDF, 2017.