Asteroidsz Review

October 2016

Summary

This report contains a review of the source code of the Java based game Asteroidz. The game was programmed for for the course Software Engineering Methods at TU Delft in Q1. The goal of this report is to analyze the existing code, and advising on code evolvability and the use of design patterns. The goal of the analysis is to establish an overview of how well the game has been programmed. The adequacy of the programming will be determined by the following items:

Existing Design Patterns The state of the current implementations of the various design patterns found in the source code during analysis.

Software Engineering Principles An analysis of how Software Engineering Principles have (not) been satisfied.

Testing The amount, type, and quality of testing performed on the source code.

Code Readability The readability of the source code analysed by formatting, naming conventions, and documentation.

From this analysis and a further review of the code, a plan of action will be presented including suggestions to refactor parts of the code using design patterns, and suggestions to correct sections of code that could cause problems with evolvability.

Contents

Chapter 1

Existing Design Patterns

In the source code of Asteroidz, plenty of design patterns can be found, namely singleton, iterator, decorator, behavior, and MVC patterns. Each of these separate implementations will be discussed separately.

1.1 Singleton Pattern

The Singleton pattern is found in **AsteroidLog**. This is a good place to implement it and it is implemented correctly. However, eager instantiation might be a good alternative to the current implementation.

1.2 Iterator Pattern

The only implementation of the iterator pattern can be found in **controller.Controller : 497-501**. This is well done by creating the standard Java iterator implemented by the java class List. While this is a valid implementation of the design pattern, the actual implementation seems without value in comparison to more modern approaches of **for(e : E)** or **E.forEach(e)**. This is mostly due to the fact that the iterator pattern is not actually implemented, but simply called from the **List** class.

1.3 Decorator Pattern

Firstly, it has to be noted that while the source code exists for this patterns, including it's implementation, none of this code is actually touched by the actual execution of the game. In other words, all this code is dead code, and therefore should not be in a release.

The decorator pattern can be found spanning several classes, with as a base the abstract class **PowerupDecorator** and the interface **Pickup**. There seems

to be a quite a bit of duplicate, and wrongfully implemented code with the implementation of this class. In example: the class **IncreaseSize** overrides a lot of methods that are the duplicates of their super methods. Besides this, there seems to be a missing method, effect, in the abstract class **PowerupDecorator**, which causes a lot of redundant, and even straight up faulty code (i.e. empty methods in **Buff** and **Debuff**).

The pattern appears in a very senible spot in the code and is easily justifiable. The general set up of the pattern is done by the book, and is done reasonably well.

1.4 Behavior Pattern

Firstly, yet again the design pattern is implemented, but never used. The only behavior pattern found in the source code can be located in the interface **ShotBehavior** and it's implementations. The only time the shot behavior is referenced is in the class **Asteroid**, which supposedly has nothing to do with the actual shooting. This is due to the fact that **ShotBehavior** doesn't describe the behavior of a shot, but instead describes the effect on an asteroid when it is getting hit. The only reference, including internally, is a single setter.

As stated, the only behavior pattern found in the source code can be located in the interface **ShotBehavior** and it's implementations **ShotSplit** and **ShotVanish**. The implemtation is straightforward and straight from the book, though. On a last note, due to the skewed responsibilities in the project the interactions provided by **ShotBehavior** are currently executed in the **Controller**.

1.5 Model View Controller Pattern

This pattern has a global span on the project and divides the project into three distinct groups: the model classes, the controller classes and the view classes. In the project a strong attempt has been made at separating the implementation to support this. However, due to a number of factors, including skewed responsibilities and bad naming, these are still very much intertwined. In example, the controller handles data directly instead of giving the model instructions. Furthermore, updating the model by one step is confusingly called **render()**. There are also issues concerning the view class(es) which don't actually create a view, but instead directly draw the data stored in the model classes. With all these flaws, it can be concluded that, while a valiant attempt was made at visually separating the classes, a lot actual functionality in actually handled in the wrong places.

Chapter 2

Software Engineering Principles

This section describes whether the principles of software engineering have been upheld.

In the project, plenty of violations can be spotted, but as a whole the project tries to adhere to these principles. One of the very well upheld principles, Readability over Compactness, manifests in the easy readability of the code. This is done by, for the most part, clear naming, and organization of the order of instructions. The identification of different classes needed for the program also seems to be done very well, with little to no reasons to split out more without incentive due to new functionality. However, the project faces same problems. Among these problems is a very poor assignment of responsibilities. As it is now, some classes(especially the controller class), seem to have more responsibilities in the program that is reasonable. This causes a very tough read and neglects the principles of responsibility driven design. Lastly, the project seems to consistently attempt to avoid abstractions in favor of hard coded pieces of software that are not open for extension.

Chapter 3

Testing

This section describes the amount, type, and quality of testing performed on the source code.

The project contains a lot of test files for AC/DC tests for the most elemental physical game objects. It fails, however, to provide tests for the more high level structures in the project. Both the view and controller in the MVC structure of the program are completely untested. On the contrary, the model code is quite well tested (ignoring the dead code found in **1.2** and **1.3**). On a far more grave note, the tests that are implemented, are done using bad pactrice in the form of using the actual implementation of associated classes instead of mocks. Especially the use of the higher object **World** in the lower class objects can be very dangerous. With this information, it is even more surprising that the total line covarage of the tests doesn't exceed 30%.

Chapter 4

Code Readability

This section describes the readability of the source code analysed by formatting, naming conventions, and documentation.

Arguably, this is the best part of the project. The code is clear and easily readable due to the clean and constant use of clear naming, and a great organization of the order of instructions. However, some classes, like **ShotVanish**, can have counter intuitive implementations. For the most part, documentation is well done, yet suddenly absent in some places, like the **eventHandler** method in **Controller**. In some places, the formatting seems to be skewed as well, missing or containing extra tabs. This could be easily fixed using auto-refactoring the indentations.
All in all, the code is very easy to understand for an outsiders. The amount of dead code can cause large problems, though. On a final note, while the classes are split into model view and controller classes according to the MVC pattern, a lot of the classes could use a subpackage to help with the organizational structure of the project.

Chapter 5

Suggestions

This section is written to provide meaningful suggestion and corrections to improve and further sustain the source code.

5.1 Suggestions for New Patterns

The current code base grants the opportunity and necessity for a couple of design patterns. These include:

Factory pattern Mainly the asteroids and power-ups could benefit from a factory to support easy and flexible construction of different objects.

Visitor Pattern The collisions (that should be handled in the model itself) can be done using a visitor pattern to support the easy addition of different types of objects.

State Pattern The current code attempts to simulate a state pattern in it's view classes(due to the MVC structure of the project). It could benefit from creating an actual state pattern that does not depent on the intermediate concrete **View** class.

Singleton Pattern In the code, there are a lot of uses of comparisons with global boolean and string values. These could be implemented using a Singleton pattern. More specifically, it could be wise to use Enums to ensure that if statements are not accidentally missed or susceptible to spelling errors.

5.2 Revisiting Old Patterns

Some of the existing design patterns could use some improvements:

Iterator pattern The current implementation of the iterator ought to be replaced by a **E.forEach(e)**. It also has to be noted that the current implementation does not actually implement an iterator, but mere calls for an already existing implementation.

Decorator Pattern The abstract decorator needs to implement ALL methods obtained through it's interface. Duplicate code needs to be removed. Auto generation of methods can cause accidental bugs. Most importantly, the code should actually be used.

Behavior Pattern The code should actually be implemented.

Model View Controller Pattern The MVC should be better refactored out according to their intended responsibilities.

5.3 General Remarks

S.E. Principles Try to adhere to KISS : Keep It Simple Stupid. Refactor methods and classes that get too big using helper classes and methods.

S.E. Principles Try to better define the responsibilities per class and split them out if there are too many.

S.E. Principles Make sure your code is open for extension, while closed for modification. This can be done implementing patterns in the right spots and by better splitting out responsibilities.

Testing Try to be wary of you cyclomatic complexity. Currently this is very high, which makes testing a lot harder and time consuming.

Testing Try to test using mocks.

Testing Try to increase your overall test coverage.

Testing Try to create different types of tests (i.e. Think cucumber and run-time tests).

Testing Try to avoid adding untested code.

Code Readability Try to auto refactor the code to avoid possibly harmful indentation.

Code Readability Try to be wary of checkstyle issues. These stack up until you have very tough to read code.

Code Readability Try to organize your classes into subclasses so that it is easier to find related classes.

Code Sustainability when you '@Override' a method try using '/** @inheritDoc */' to inherit the JavaDoc of the original method.

Code Sustainability	Try to pay attention to possible bugs and bad practice using your already installed tools (FindBugs et alii).
Code Quality	Try using enums for identification, this simplifies code and makes it more consistent throughout the program.
Code Quality	Try to use switch case statements instead of a list of ifs.
Code Quality	Try to equals methods instead of ==.
Functionality	If a player hits an second asteroid when "dead", you lose another life. This is not supposed to happen.
Functionality	The logger doesn't seem to work correctly. The log file only contains '[19/48/2016 - 19:48] [className] text''s.
Functionality	The log file is created at a weird place, 'C:\', while the game ran from 'C:\...\...\'.
Functionality	The game doesn't actually terminate when clicking the close button.