Assignment 3

SEM Group Project, sprint 3, week 1.6.

TU Delft Software Engineering Methods 2015-2016 (TI2206)
Supervisor: Dr. A. Bacchelli
Teaching Assistant: A.W.Z. Ang

Group 12

David Alderliesten	4368703	J.W.D.Alderliesten@student.tudelft.nl
Jesse Tilro	4368142	J.Tilro@student.tudelft.nl
Jeroen Meijer	4382498	J.Meijer-5@student.tudelft.nl
Floris Doolaard	4362748	F.P.Doolaard@student.tudelft.nl
Niels Warnars	4372069	N.Warnars@student.tudelft.nl

Table of contents

1 20-Time: Refactoring the User Interface

Motivation

The old graphical user interface (henceforth GUI) was outdated and inflexible. It was decided within the team that a main menu and a toggle-able multiplayer feature was needed. However, attempting to implement this in the old GUI would have been impossible. As such, an entire re-do of the graphical user interface aimed at improving the visual design, code quality, and flexibility of the GUI.

CRC session

During the CRC session, the cards were aligned in quite a few ways, and there were numerous additional cards which were utilized. Some examples of these other cards are "Menu Controller," "Settings Handler," and "Extendable Window Handler." The cards above were the result after many extraneous cards were removed from the session. In the interest of time, project scope, and capability, the five following cards were chosen as the required classes for the new interface:

- Main Menu
- User Interface
- Settings Menu
- Game View
- Game

The game class itself is a reference to the game class, which acts as the center point for the user interface's data. The User Interface is the instantiation and general interface responsible for the generation and implementation of the interface. The Main Menu is called from the User Interface, and the user interface's window is passed to the main menu for handling and display. The user can then make a call to the settings menu or to the game view. The settings menu is a simple implementable pseudo-singleton that allows any other GUI interface or class to call it without parameters or requirements. The Game View can be opened to render information from the game.

After the session, this implementation seemed to be the most simple yet extendable one. As such, this implementation was agreed upon by the team to implement as the "new graphical user interface". The CRC cards describing the purposes and collaborations of the classes of interest are visualized in the diagram below.

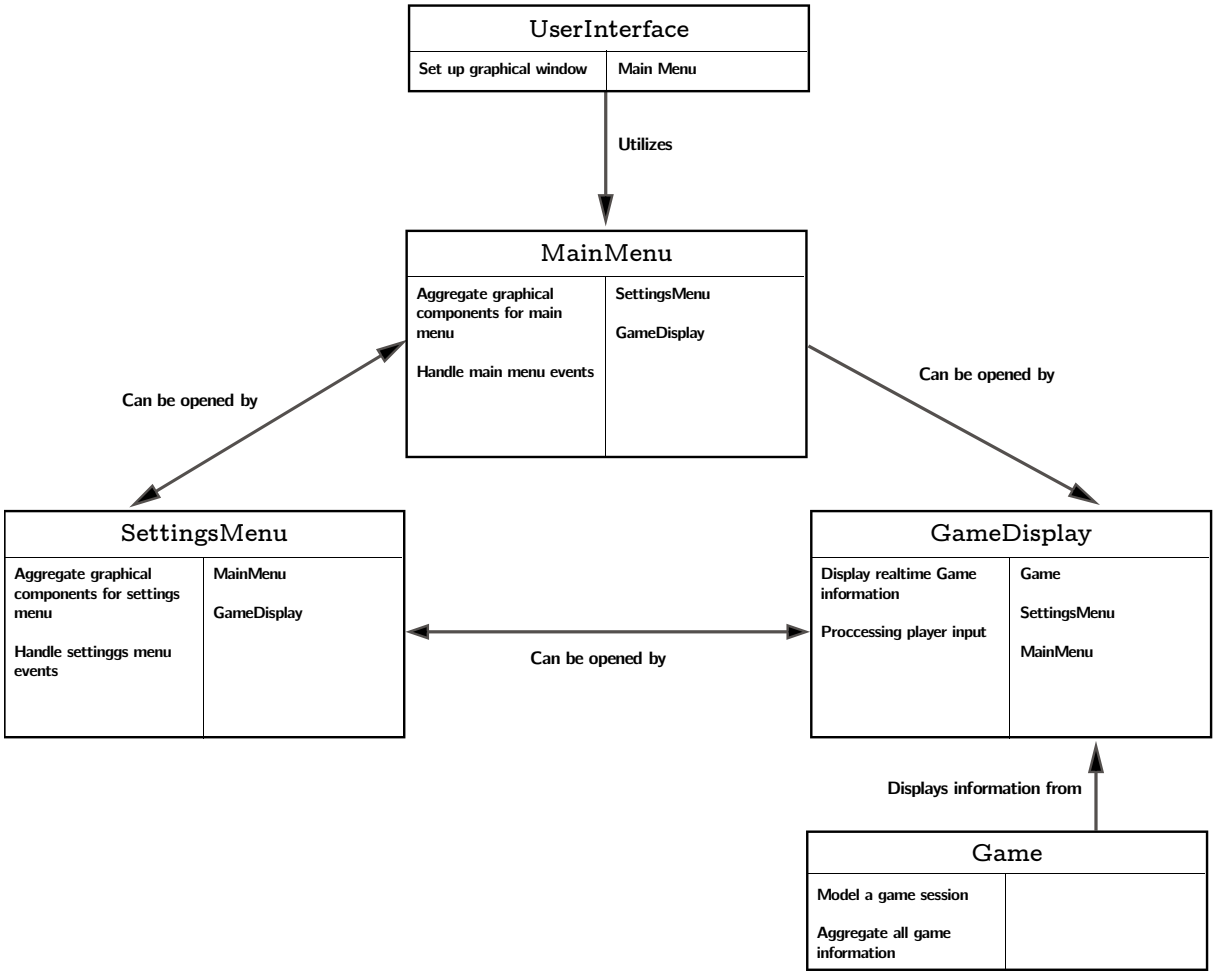

Figure 1: A diagram of the CRC Cards derived through Responsibility Driven Design for the implementation of this sprint's updated Graphical User Interface.

UML State Diagram

During the planning, it seemed like a good idea to think of each GUI component as a "state." In other words, we had a settings state, a main menu state, and a game state, which in turn could be split into singleplayer and multiplayer cases. A state diagram of the final design can be found below.

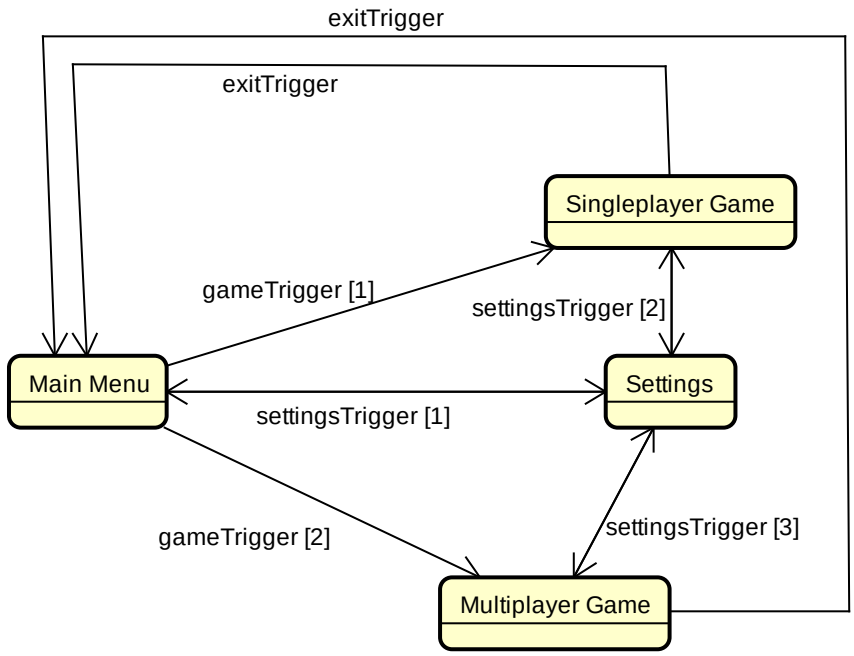

Figure 2: The UML state diagram describing the refactored Graphical User Interface implementation as a State Machine.

2 Design Patterns

2.1 Singleton Design Pattern

For the first design pattern we implemented the Singleton Design Pattern. This section discusses why and how this design pattern was implemented and includes the UML Class and Sequence diagrams visualizing this implementation.

2.1.1 Why the Singleton DP was implemented

When we first implemented the Logger feature during the previous sprint, we figured that this should be implemented by a single central component. There would be only one log file per session, to which only one line can be written at a time. Furthermore this component should be accessible by any class, since the hooks for writing a line to the log occur at many different points in the code.

Therefore, we initially implemented this functionality using class scoped methods and attributes, in other words, we implemented a Utility class. The problem with this implementation however, is that Utility classes generally are a bad practice, for they may violate principles of good object oriented design:

- The class may easily implement many different class scoped methods, in which case the Single Responsibility principle would be violated. This would leave the class to have no clear purpose and place in the system.
- The class loses the advantage of Object Oriented design, since it is used as a Class rather than an Object.
- The class is not designed to be extended upon, not really conforming to the Open Closed principle.
- The class cannot derive from an interface, nor implement a base class, so it violates the Dependency Inversion principle.

As a solution, we refactored our Logger class to implement the Singleton design pattern. This class has a private constructor, and a class scoped method allowing clients to construct and get an instance of the Logger. This way, the Logger class can make sure only one instance of itself will ever be instantiated, and all clients refer to the same instance. We implemented a thread safe instantiation of the unique instance using double-checked locking (presented as solution 3 to the concurrent instantiation problem during the lectures). We kept the ability to call the Logger at any place in the code and the prevention of writing to the same file concurrently, but we are now able to use it in an instance scope.

2.1.2 Class Diagram of Singleton DP

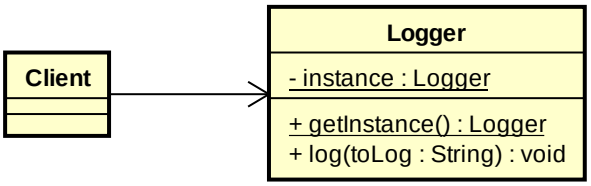

Figure 3: The UML class diagram visualizing the implementation of the Singleton design pattern on for the Logger class. The Client class represents any possible class that needs to log something.

2.1.3 Sequence Diagram of Singleton DP

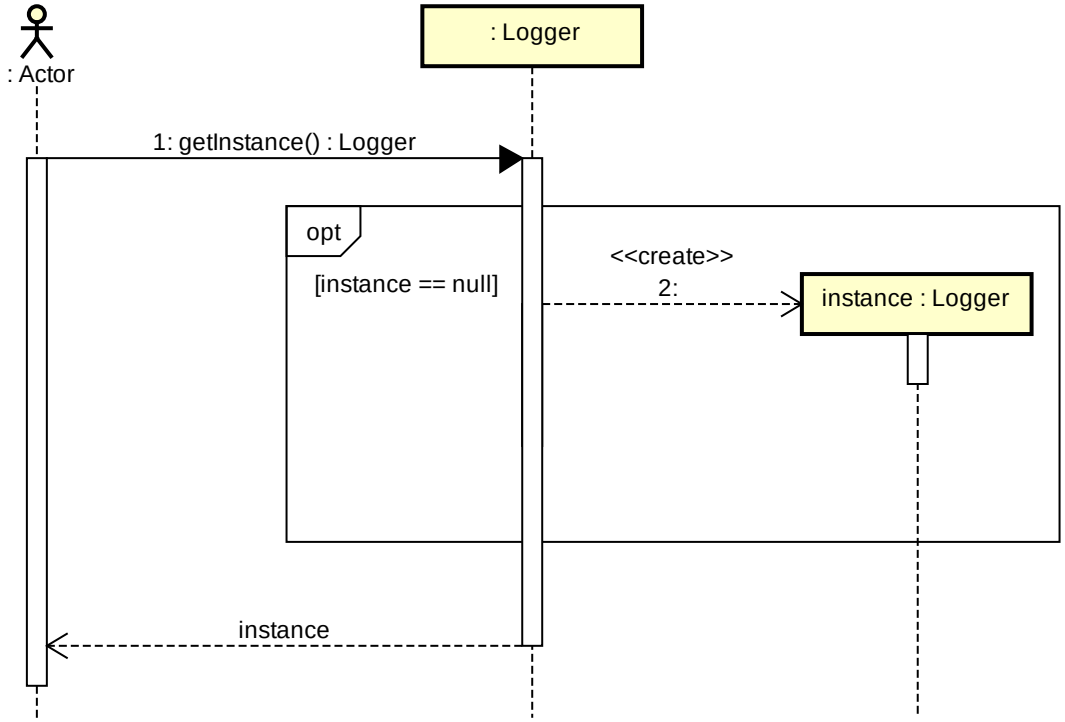

Figure 4: The UML sequence diagram visualizing how the implementation of the Singleton design pattern for the Logger class can be used. The actor can be any client in the system.

2.2 Observer Design Pattern

For the second design pattern we implemented the Observer Design Pattern. This section discusses why and how this design pattern was implemented and includes the UML Class and Sequence diagrams visualizing this implementation.

2.2.1 Why the Observer DP was implemented

The Observer design pattern was implemented in an earlier phase of the project, while we were unaware of it being the Observer design Pattern. We implemented a construction that allowed the Game class to notify a bunch of dynamically added Level Modifiers on every cycle that is propagated by the clock. (Coincidentally, the Game class actually observes the Clock, but since this is quite trivial, we decided to focus on our Level Modifier implementation for this design pattern.)

Whenever the Game (Observable) performs a step method, the Level Modifiers (Observers) registered to the Game are subsequently notified by having their modify (the notifyObservers equivalent) method called. As parameters to this call, the current level of the game to be modified and the delta (exact number of steps passed since last execution) are passed to the observers. In order to effectuate subtyping polymorphism for the different Level Modifier classes we created an Interface *LevelModifier*. This Interface serves as the Interface for the Observers, so that the Observable may treat its Observers uniformly.

Implementing the Observer design pattern this way allowed us to distribute the intelligence concerning updating the state of the system's models over different classes, each with a clear responsibility. However, we concluded that there are disadvantages to this approach as well, which will be briefly discussed in section 2.2.4.

2.2.2 Class Diagram of Observer DP

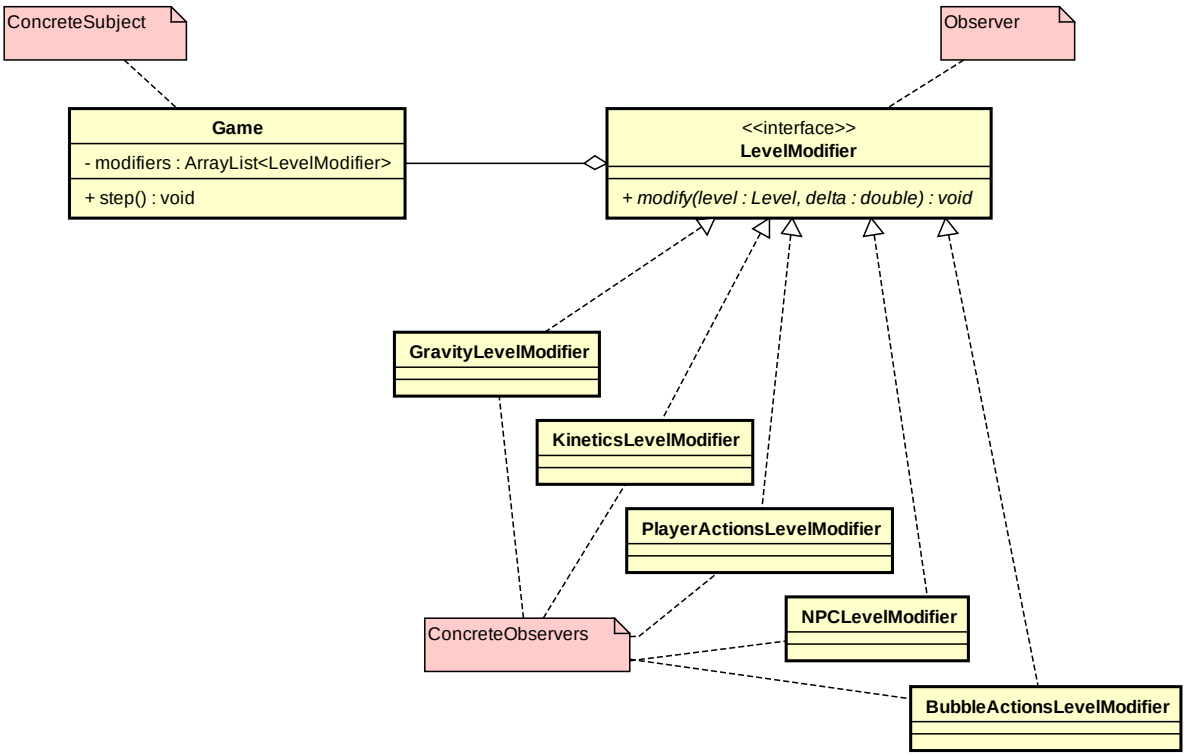

Figure 5: The UML class diagram visualizing our implementation of the Observer design pattern between the Game and Level Modifier classes.

2.2.3 Sequence Diagram of Observer DP

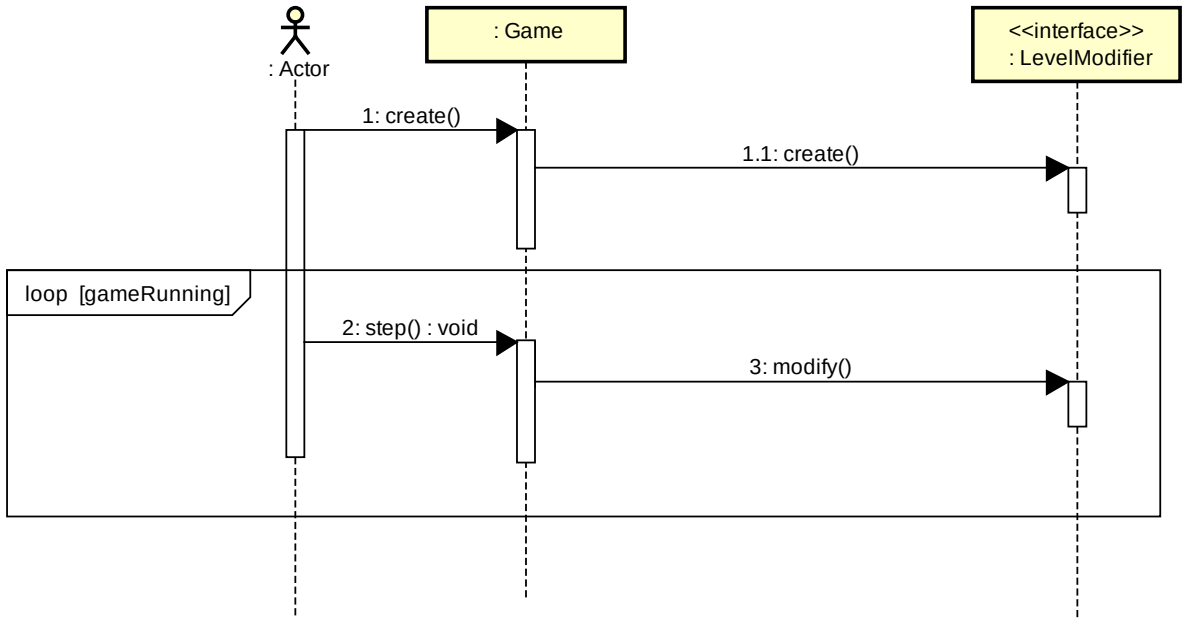

Figure 6: The UML sequence diagram visualizing how our implementation of the Observer design pattern between the Game and Level Modifier classes would be used. The Actor would probably be an instance of the StepTimer class.

2.2.4 Additional remarks on the Observer DP implementation

We noticed our implementation of the Observer design pattern has led to some design problems. Now that all the behavior is implemented in higher level components, the Level Modifiers, the model classes actually representing the elements in the game, the Level Elements, have become data classes. They do nothing other than store data about the element and provide a lot of simple accessor methods to this data. The Level Modifier classes have become quite large, use these simple Level Element classes extensively and do this via a lot of non-communicative behavior (accessing and updating data). Therefore these classes can be classified as god classes.

For this reason, we have decided that we would like to refactor this implementation in a future sprint. The Level Elements themselves should gain more intelligence and do more operations on their own data (encapsulation), without them getting too bloated and while allowing them to be more uniformly treated. We were thinking of applying for example Strategy design pattern for assigning behavior to elements. Since the behavior does not necessarily have to be interchangeable at runtime it is not a perfect fit solution, but it does help to distribute system intelligence in a better way. Another design pattern that could be used to solve this problem is the factory design pattern, in order to make different Level Elements implement different behavior while being uniformly treated by clients.

3 Software Engineering Economics

3.1 How to recognize good and bad practice

The authors of the paper "How to Build a Good Practice Software Project Portfolio?" took a number of steps to distinguish good and bad practices.

The first step to take involves comparing the performance of the project under investigation with an (industry) average. If a project performs better than average on cost and duration then the project is seen as a good practice, if a projects performs significantly less than average on cost and duration then it is considered a bad practice.

Besides only classifying a project as a Good or a Bad Practice one could also look at the key characteristics that make out a project. In the conducted research 56 project factors were drafted that could have an impact on the overall performance of a software project. If a certain project factor is present in more than 50% of the Good Practice projects that this is a factor that might help in creating a Good Practice project. The same thing can be said about Bad Practice projects and their project factors. If a project factor is present in more than 50% of the Bad Practice projects then this is likely one of the causes of that.

3.2 Why visual basic was considered a good practice

In the paper "How to Build a Good Practice Software Project Portfolio?" using the Visual Basic programming language turned out to be a good practice. However, this good practice factor is not representative for the entire collection of projects. In total only 6 projects that were analyzed were written in Visual Basic in which 5 came up as Good Practice and one as Cost over Time whereas the entire collection consisted of 352 projects, meaning that only 1.7% of the projects was written in Visual Basic. This number is too low to say something definitive about the entire collection.

In the end however it is still hard to tell why exactly Visual Basic projects end up more often in the Good Practice quadrant than other projects. The paper suggests that this might be due to the lower than average complexity of these projects, but no concrete cause has been pointed out in the paper.

3.3 Other factors of interest for study

1. **Test driven design**
   This factor can likely be considered as a good practice factor as it enables a development team to kill bugs in an early stage of development and prevent delays and additional costs due to unforeseen problems.

2. **Team consisting of more than eight developers**
   Whenever a development team consists of a large number of people problems might start to occur. The team starts to be unmanageable as in a large team communication efforts start to take up too much time and become too complicated, therefore this factor might be considered as a bad practice factor.

3. **Continuous Integration**
   CI allows for a uniform way of testing making sure that no local dependencies can cause false test results and ensures more consistent test results. Therefore this factor could end-up as a good practice factor.

3.4 Three bad practices explained

1. **Once-only project**
   When dealing with a once-only project there are a number of things that can be said about the likelihood of failure. First of all there is the fact that it takes time before developers joining in on a project are familiar with the code base. This process will only be of use when a developer works with a code base for a longer period of time, something that is likely not the case in a once-only project.

2. **Dependencies with other systems**
   Whenever a project starts heavily relying on other systems problems might occur. Whenever changes are made to the project or to the system(s) that a project relies on one has to make sure that project is still compatible with its dependencies. Not only can maintaining these systems be a complicated task, but changes may also result in failure of either of the projects. This is a classic example of 'building complexity on complexity' in which it is taking much more effort to support an existing system. Resulting in extra time needed and accompanied additional costs.

3. **Security**
   Security is considered one of the nine bad practice factors, why is this the case? Whenever an application is designed for being used in critical infrastructure or in the public domain there is the risk of vulnerabilities being exploited by malicious actors. To minimize the chance of this happening a company could choose to make security a vital part of the development process and this is where problems (delays and costs) might occur. Whenever security is important in a software project this means that additional measures have to be taken to ensure the safety of a system like the performance of multiple source code audits or application penetration tests conducted by external parties. This testing and the resolving of found vulnerabilities not only costs a large amount of time, it also severely impacts the budget of a software project. Besides the costs of external audits, internal trainings regarding secure software development take up a large amount of resources all contributing to an increase of time and costs.