Software engineering methods assignment 5

by

Irene van der Blij, Mayke Kloppenburg, Samuel Sital,
Kiran Kaur, Takang Kajikaw Etta Tabe

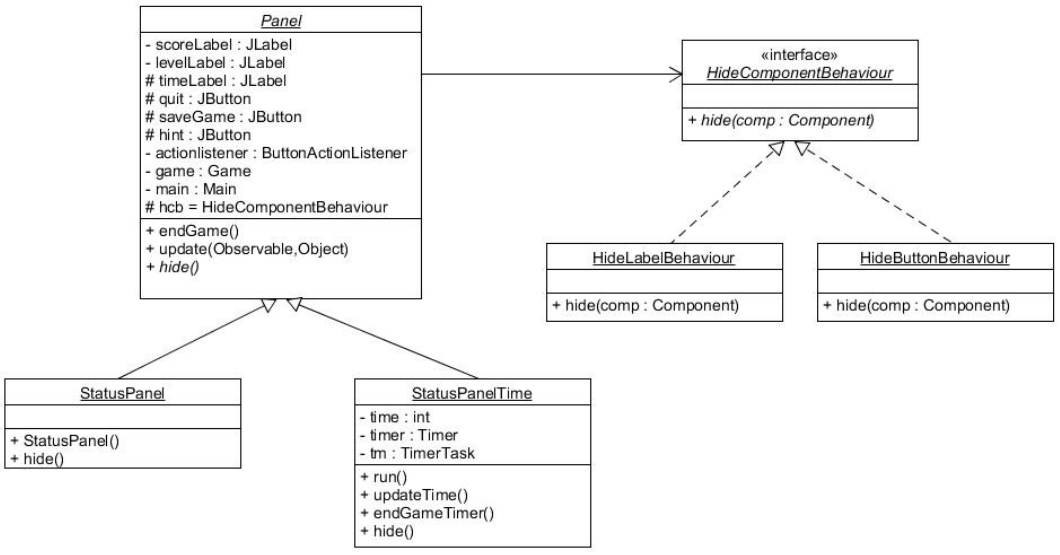
TI2206 Software Engineering Methods

Group 37

Exercise 1 – 20-time, revolutions

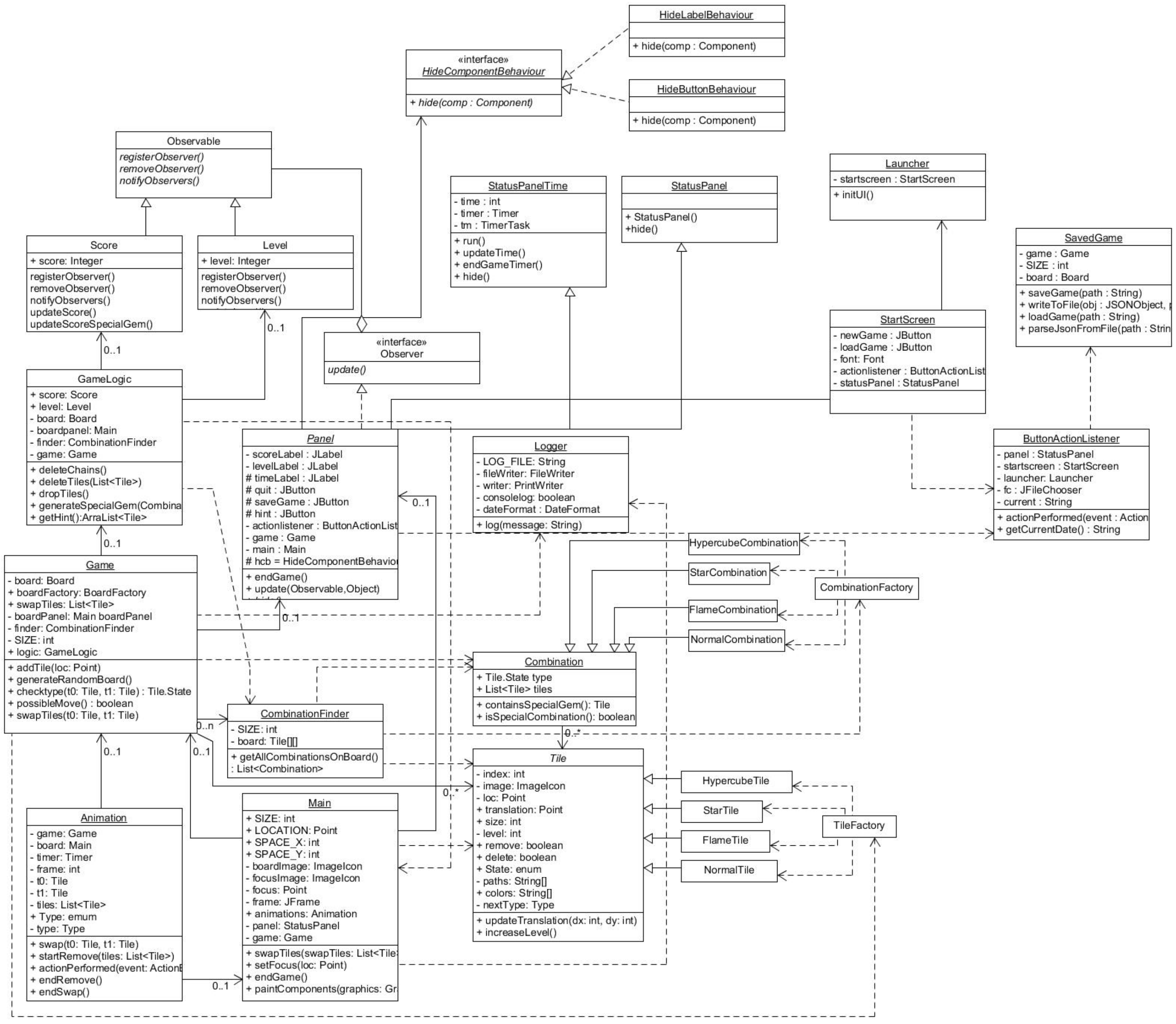
During this sprint we managed to implement two extra novel features, the first one being able to get a hint. In order to manage to get a hint when playing the game, for the logic behind this we only needed to write a method getHint(), which we implemented in the class GameLogic. To integrate this logic into the user interface was also not a great effort, since we built our system with evolvability in mind. We already had a focus image on the board, which would appear around a gem when this gem was clicked. For visualizing the hint we only had to create a new focus image, and place this around the two gems that could be switched.
Since this feature turned out to be very easy to implement, due to the evolvability of the system and similar methods already existing, we were not able to really show off design patterns with this particular feature. Therefore we decided to implement another novel feature which would allow us to do so.

The second novel feature that we implemented during this sprint is the Time Mode. The idea of the time mode is that you start the game with a predefined amount of time (60 seconds in our game). During these 60 seconds you can make as many valid combination as you can, and earn points. When you are out of time, the game is ended and you are not able to do any more moves.
The first thing we did to realize this feature, is to create a separate statusPanel class for the time mode. We now have the classes statusPanel and StatusPanelTime, which extend the abstract superclass Panel. We did this, because this was the easiest way to distinguish between a normal game and a time mode game. The superclass Panel is responsible for putting all the labels and buttons on the screen, but we did not want all the labels and all the buttons in both game modes. The normal game mode should not have the timer label, and the time mode should not have the save game button. In order to solve this issue we used the strategy pattern. We made an interface HideComponentBehaviour, which is implemented by HideButtonBehaviour and HideLabelBehaviour. In HideComponentBehaviour we have a method hide; in HideButtonBehaviour the hide method causes a button to be invisible, in the HideLabelBehaviour class the hide method causes a label to be invisible. If a new StatusPanel or StatusPanelTime object is made, the right hide behaviour object is instantiated. This makes sure that the right hide method is called.
To clarify how this works, we have the following class diagram:



If we like to make new game modes or screens in the future, this strategy pattern can also be very useful, since we can easily extend or reuse it. If we would like to make new screens with different buttons or labels but the same layout, this feature will be very useful and easy to use. This, again, supports the evolvability of our system.

Our full UML class diagram now is as follows:



Exercise 2 – Design patterns

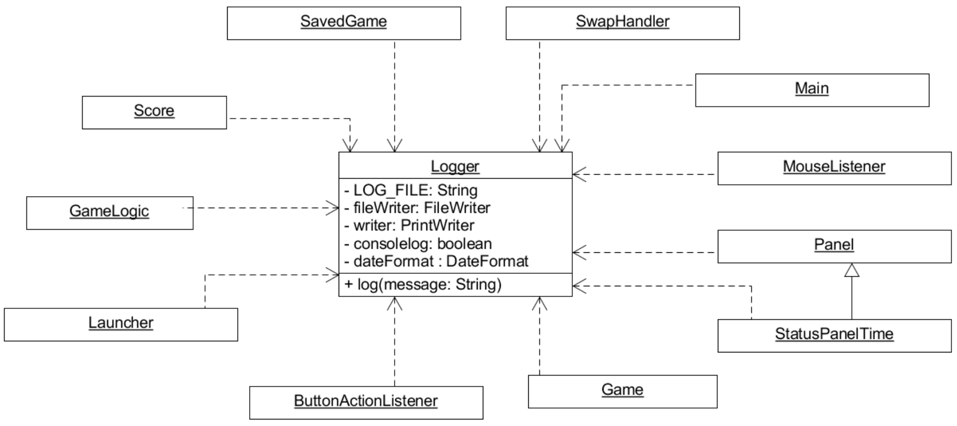
The first design pattern: Singleton

1. For this exercise we needed to implement another design pattern. This was a tough task for us because for the previous assignments we had already implemented some design patterns. Also, for the current assignment, for the first part a timer mode was implemented and this was done with the help of the Strategy pattern. Since this pattern was already explained and visualized in a class diagram in the previous exercise, we decided we would not do this here again. Considering the fact that we already have so many design patterns implemented, we decided to consider one of the already implemented designed patterns for this question.

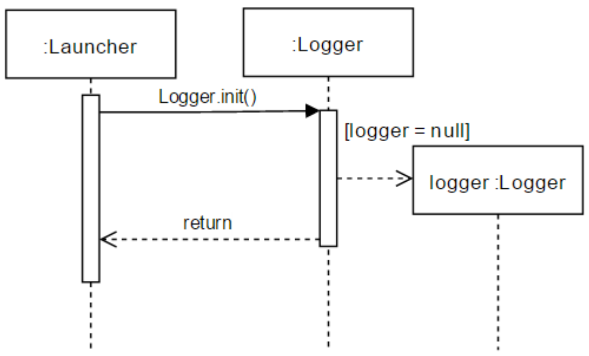
   From all the newly implemented patterns, we chose the singleton pattern to talk about. Looking at our code, we have a lot of classes that are accessing the logger.java class. Since all these classes need to access the logger.java class, we created just one instance of the logger object and decided to implement the singleton pattern. This logger.java class therefore provides a global logging access point in all the application component without being necessary to create an object each time a logging operation is performed. A few of the aforementioned classes which perform logger operations are SwapHandler, SavedGame, Score etc.

   To realize this design pattern in our code, we have an init() method in our logger.java class. This init method makes a new instance of the Logger only if the logger object is null; otherwise it uses the already instantiated logger object. Since the method is also synchronized, it is not possible to have more than one instance of the Logger class. The constructor of the Logger class is made private so that no other class can make an instance of the Logger. When the other classes need to make use of the logger object, they just call this one instance of the logger.

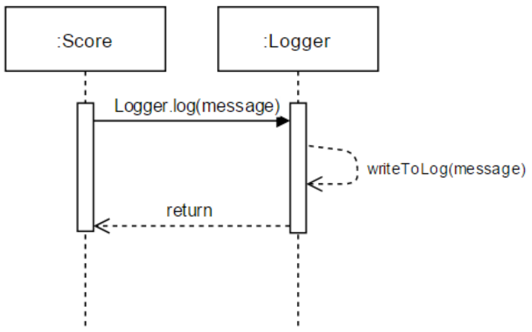
2. To visualize how the singleton pattern form the logger is statically used in our code, we made the following class diagram. The diagram clearly shows that the Logger is used by many different classes, since many classes contain actions that need to be logged when executed.



3. To visualize how the singleton pattern works dynamically in our code, we make the following sequence diagram:



When the program is started by the client a new Launcher is made, which calls the Logger.init() method in its constructor. When the logger is not yet instantiated, and thus null a new logger object is made. When the logger object is already instantiated, this instance is used.
This instance is used by many classes, as is shown in the class diagram. To show how these classes use the Logger, we made the following sequence diagram. This diagram uses the Score object, but the diagram is the same for every class that uses the Logger to log a message.

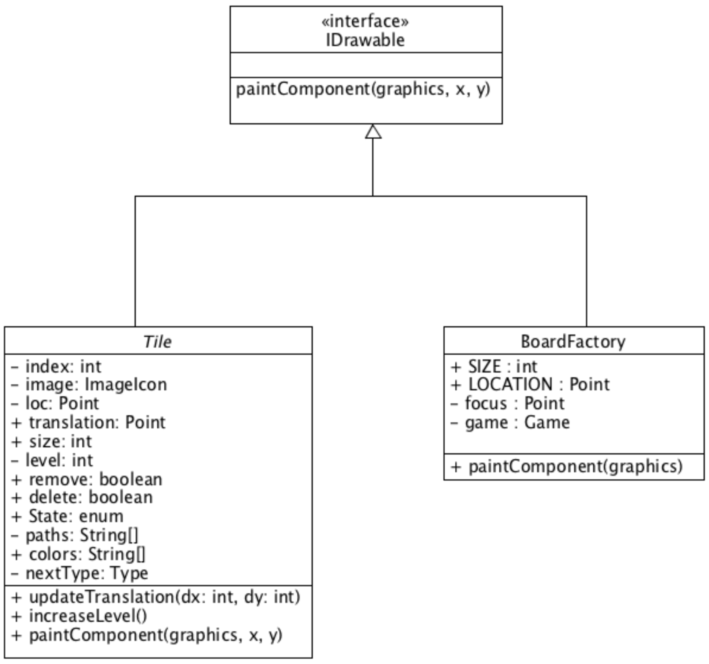


The second design pattern: Composite

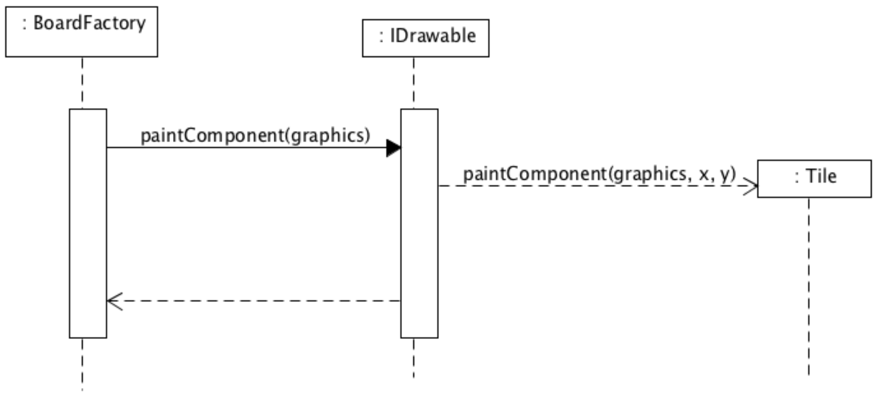
1. At first we couldn't think of any pattern that would make our game code more efficient and comprehensible. It took us a while to decide which pattern to implement next since we had already implemented the Singleton, Observer and Factory patterns. Eventually, we decided to go for the composite design pattern. Since our data is tree structured, this makes our code a little bit more complex especially when trying to differentiate between a leaf node and a branch. The solution to this was to implement an interface called IDrawable which allows us to treat the boardfactory and tile classes uniformly. With the composite pattern, our boardfactory.java and tile.java classes now have a relationship between themselves and that is that of painting the tiles on our board. This pattern was also implemented because it was the only one which could be implemented in a relatively small amount of time.

   In our code, we created an interface called IDrawable which has a paintComponent method. This paintComponent method is both used by our tile.java and boardfactory.java classes. In our BoardFactory.java class we have the paintComponent method, this paintComponent method uses the paintComponent method implemented in the Tile.java class by making use of abstract IDrawable interface.

2. This pattern led to the following class diagram:



3. This pattern led to the following sequence diagram:



Exercise 3 – Wrap up - reflection

It is known that many hands make work lighter, which means that working as a team on a particular project is always a fun experience compared to working on a project alone. Hearing about software engineering methods, we would think about methods or ways which can be used to make the quality of software better, how we can solve regular problems faced in the process of software development, how we can create good software and in general, how we as developers can become good or better software engineers. In this paper, we analyse the progression of our project from the start of the project until the present and how we worked as a team to be able to achieve common goals.

According to Wikipedia, software engineering is defined as the application of engineering to the design, development and maintenance of software. This is also seen as the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines. This definition when looked at in details, matches the thoughts we all had at the beginning of this course. Considering how far we have come from the start date of the project until now, it is very obvious that this course has improved our skills in the field of software development and has taught us a lot on how we can improve our software quality and solve problems which frequently occur during the process of software development.

In the beginning of this course and practicum session, we were expected to have a working version of a game at the end of the second practicum week and at that same point in time, we were introduced to scrum, which is considered to be an important aspect in the field of software development. This meant hard work right from the word “GO” and this was exactly the kind of kick or boost we needed to be able to get to work. Figuring out how to start was one of the greatest challenges we faced during the first two weeks of the project, but once the foundation on which we were going to work was laid, development of our software progressed at an increasing rate.
However, since we had to finish this working version so fast, and without much knowledge about good software engineering practices, our code for the working version was not written very neatly. This led to some problems later in the project, since we had to extent and change certain aspects of our game which were not written in a evolvable way. During the weeks after finishing the working version we had to put a decent amount of time into organizing the code in a logic and evolvable way. Now that we are coming close to the ending of the project, our code is a lot more organized than after the working version, and this can also be seen in the easiness of implementing new features such as hints and time mode.

We made use of the *scrum methodology* during our software development project. At the beginning of every week, the various tasks which were to be done for that week were divided and each group member was assigned a task. In some or most cases, two or more people were assigned on a particular task depending on the weight of that assignment and the required effort; this division was included in our *sprint plan*. Also, we had a *reflection* every week, where we wrote down how we can improve in the following week and the problems or challenges we faced the previous week while working on the assignments. These problems were mostly discussed during the meetings on Tuesday with the student assistant, and solutions were always found to those particular problems with the help of our student assistant.

Working in groups on one project may often lead to conflicts within the group and just like other groups, we had problems within the group while working on this project. With the help of the student assistant, this was peacefully resolved. This taught all of us in the group that working together is better than working separately and is more productive because we came up with more productive ideas while working together than when working separately. We also noticed that due to the task division, work was done as it was supposed to be and it was also done on time. Looking at all these points, working as a group is *good practice* but when the time is limited, it is more advantageous if the workload is equally divided between all the group members, in this way, everyone can be a contributor to the project.

During this project, we learned a lot on how to develop better software using an iterative working approach, how to work as a group to achieve a common goal, how to get work done within a particular time interval, how to peacefully deal with problems within the group and most importantly, how to deliver working and correct software of high quality. All of the aspects learned from this course have been very useful and will help us in the future. Now we know what approach to use when asked to develop a particular software and also how to solve the various problems which one frequently comes across in software development.

Looking at the working version we submitted after two weeks into the course, we can say that it was working but full of bugs and had little or no extra features, no design patterns and the code structure was not organised and poorly structured and almost not understandable for any external parties. As time went on, extra features were added to our game (such as the special gems), our code structure was reorganised, bugs were fixed and the software quality was improved. The software at this point is not perfect but it is better than the first delivered working version and is still undergoing refinement. Most of the skills acquired to make this project a success were acquired during lectures and looking at the progress made with time, this is a clear indication of the fact that some more values were acquired and that with more time, we can build a better working software.

The building of software is a continuous and gradual process which never ends and is also subject to changes. Becoming a good software developer entails being able to work in a group, put in some relevant input and your goal should not just be delivering working software, but a working software of high quality.

Now that the Software Engineering Methods course is coming to an end we can all say that we have learned useful things from this course. We have continued our learning process of working in groups and communication with each other, even if you do not agree with each other. We have learned that communication is one of the most important factors of a project, and that internal group problems need to be openly discussed. Besides the group process we of course also learned a lot about good software engineering practices. As mentioned before, our working version was not neatly coded. When, in the future, we have to start a new project, we will definitely take design patterns into account, especially when setting up the project. The learned design patterns and other good software engineering practices will surely help us to write better code in the future!