Documentation

Assignment 3 - Software Engineering Methods

Game: Bubble Trouble
TA: Maiko Goudriaan

Exercise 1

1.

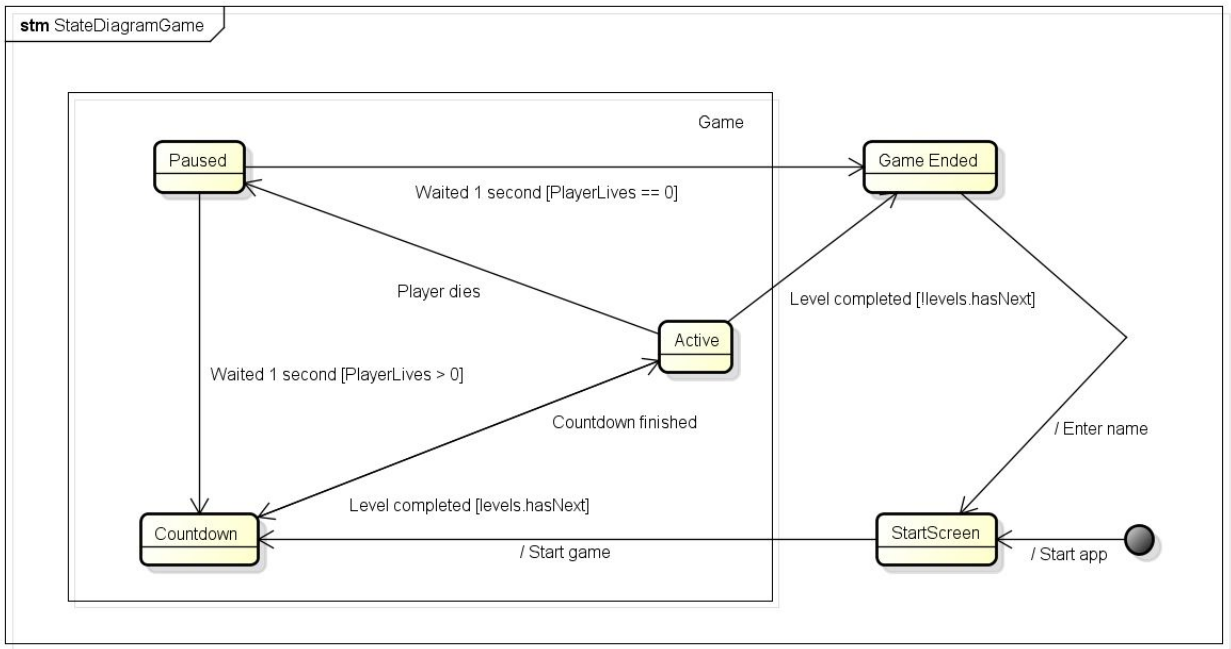
The following improvements were decided upon for this exercise.

- Implement a high scores feature
- Implement an options menu
- Implement transitions between levels.

These improvements all consist of new features that will be added to the game.
A requirements document for these features is supplied alonside this document.

2.

Transition

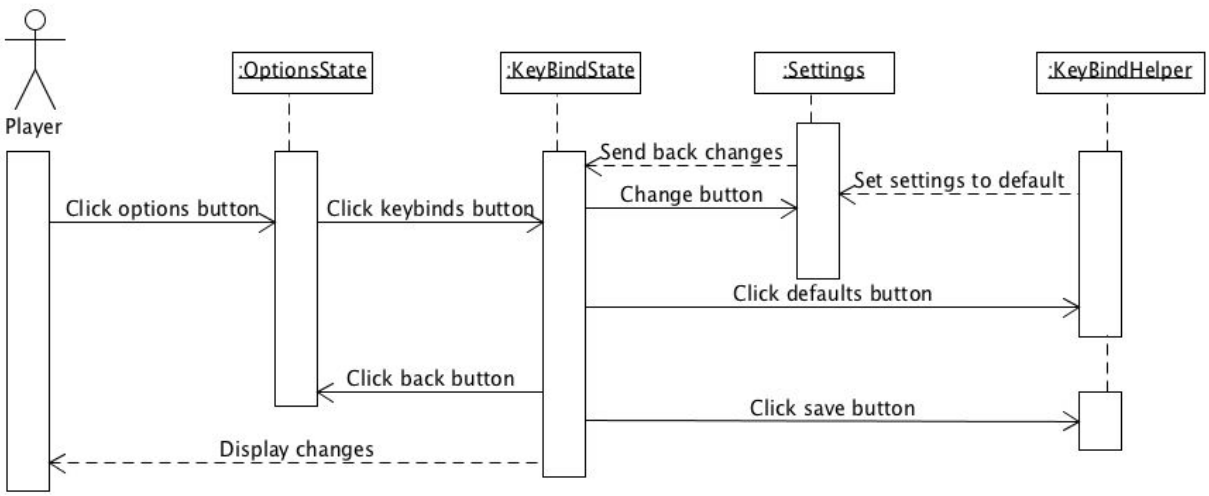


Above you can see the statediagram for Game, concerning the implementation of two new features. When the user starts a new game, a countdown is started from 3 to 1 (this countdown is regulated in the class Countdown, of which Game contains an instance). When the countdown is finished, the game is started.
When the player dies, the game is paused for 1 second (to be clear, the screen is ‘frozen’ for a second, but a pause menu doesn’t appear). This pause is regulated with a Timer, which is done at the moment the player collides with a bubble.
When the second is over, the application continues. If the player still has lives left, the level is reset and the countdown starts again, after which the cycle starts over. However, if the player has no lives left, the game is over.
When the player completes a level, which isn’t the last level, a new level is started with a new countdown (the shopscreen is left out for simplicity). However, if the player completes the last level, the game is finished.

To summarize, the new features that are covered here:

- A countdown before the start of each level (also when the player dies and the level is reset).
- Pausing the application for a second when the player dies.

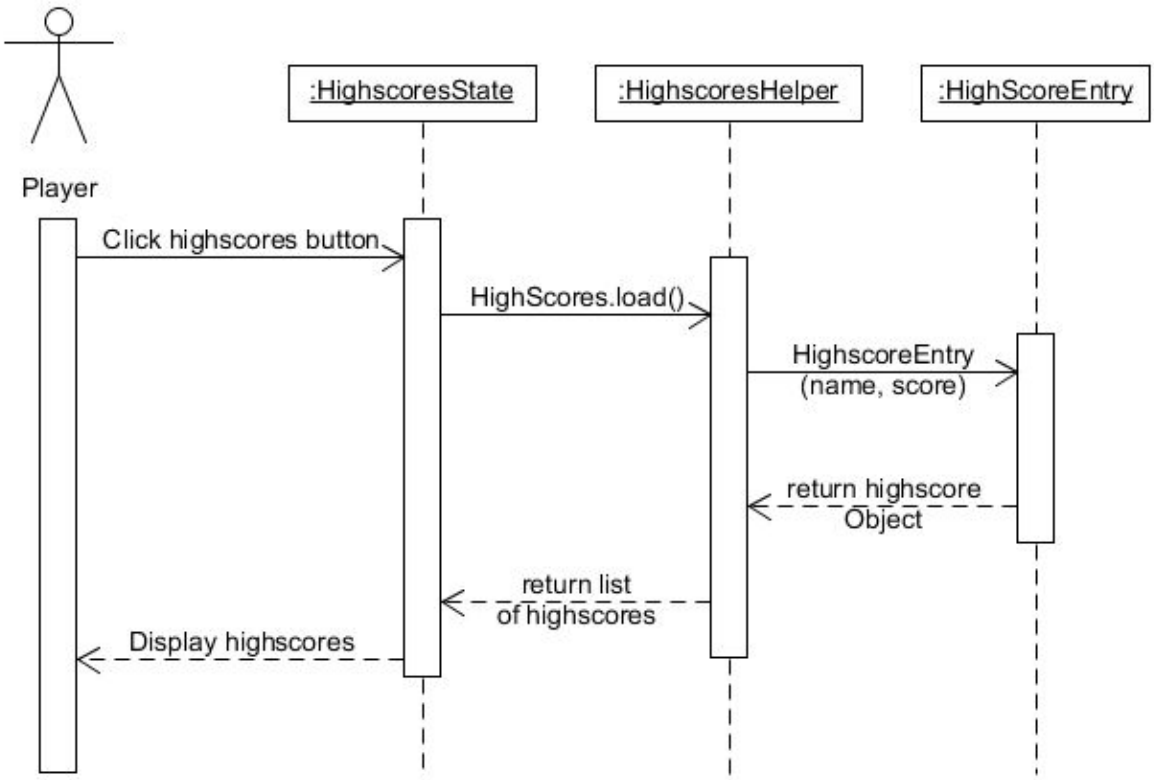
Options Menu



These sequence diagrams show the changes made to the options screen.
The first one displays the ability to go to a logger set screen. In this screen the player is able to press the logger severity they want to use. The player can cancel this by hitting the cancel button.

The second diagram shows the ability to change the keyboard layout. The player can go to the KeyBindState from the OptionsState. In this state they will be able to change the keys by hitting the new key on their keyboard, then clicking on the key they want to change. This will be saved to file by hitting the save button. This is handled by the KeyBindHelper. If the player does not save the new keyBinds, it is just saved in the Settings. This means the handlers for player movement will use the new keys while the game is running, but will be reset when the game is restarted.
The player can also click the defaults button in the KeyBindState. This loads the default settings from file. The loading is handled by KeyBindHelper.

High Score



The sequence diagram for loading and showing the highscores is shown above.
When the player has finished a game either by losing all of his lives or by completing all levels, he will reach the end screen, as described on the previous page. The name entered on this screen gets saved in a json file, together with the corresponding score. When there are 2 players, 2 entries will be added to the json file.

In order for the player to view his results and those of other players who have played the game on the same pc, he must click the highscores button in the start screen. When the button is pressed the HighScoresState calls the HighscoresHelper class which reads the json file containing the highscores. Each name and corresponding score are used to create

a HighScoreEntry which is returned to the HighScoreHelper class to be added to a list. As soon as all high scores are read this list is sorted in descending order (so the highest score will be on top). This list is then returned to the HighScoresState to be rendered onto the screen for the player to view.

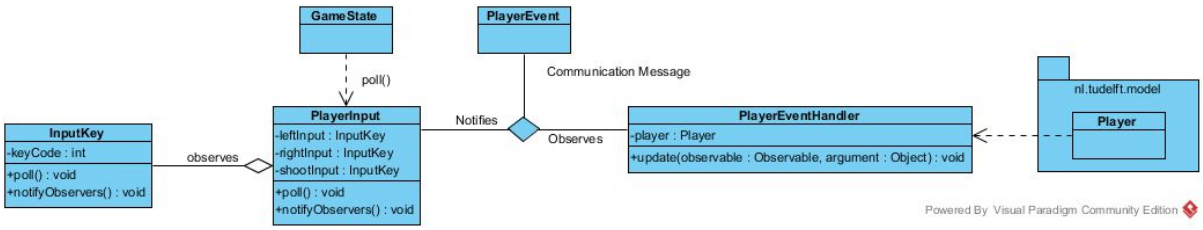
Exercise 2

Observer Pattern

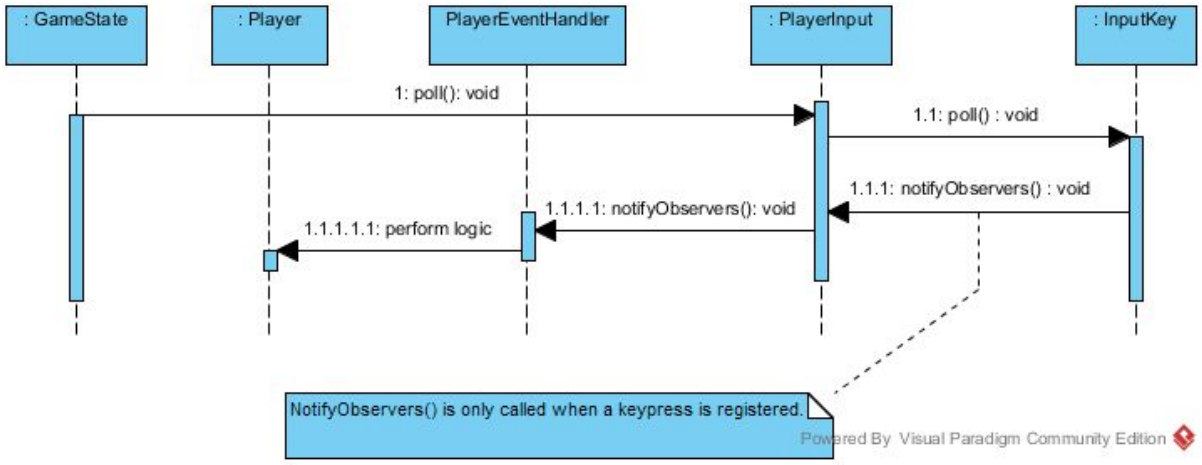
1

The Observer pattern is implemented to delegate the handling of Player Input to a separate class PlayerEventHandler. This class is able to receive events (it observes) that are fired by the PlayerInput class.
The PlayerInput class is constructed from the Settings and can therefore be easily changed. When a Player is created, there only needs to be a PlayerEventhandler that is created for that player, and that Handler must be set up to observe the correct input (either player1 or player2). The Player itself has no clue of what happens: he only provides the public methods that are required to perform the movements and events.

2.



3.



Abstract Factory Pattern

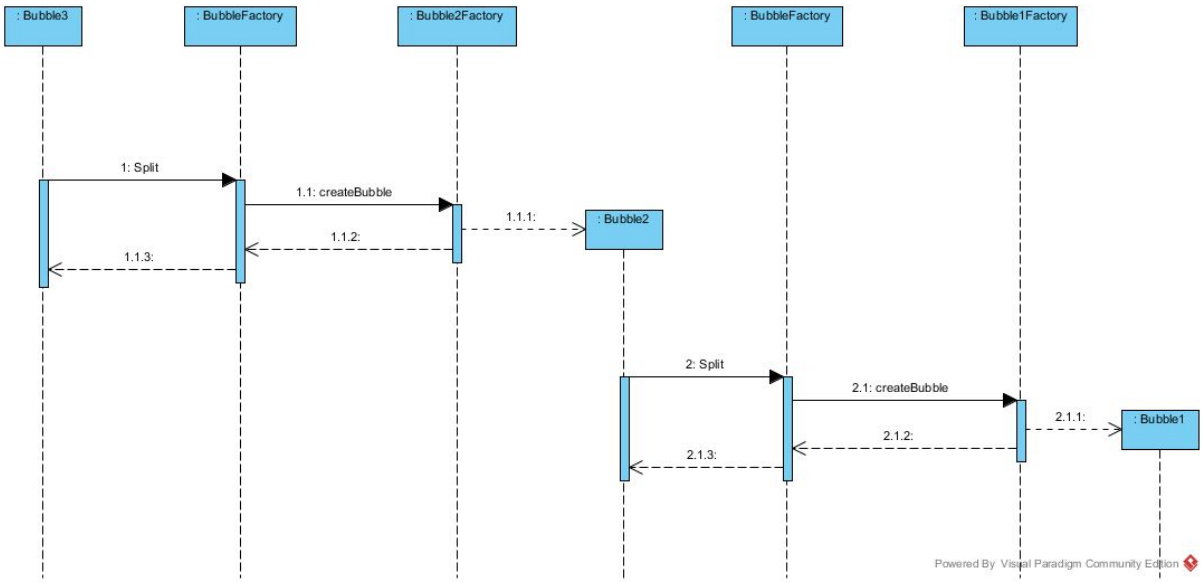
1.

The Abstract Factory pattern was chosen to ease the way in which the numerous objects in the game environment are created, especially on runtime. The existence of an AbstractFactory allows the objects that need to create new instances to be able to call to that interface. the benefit is that if the desired object must be changed, the object that wants to create it can remain intact (no constructor needs to be updated, or anything like that), since

the factory is simply implemented differently. This decouples the objects from each other, they no longer need knowledge of each other's detailed type and settings.

2.

3.



Exercise 3

1.

Good practice is recognized by the project being under the mean in both the Cost Matrix as well as the Duration Matrix. This means the project will have been completed faster and cheaper than the mean.
Bad practice is recognized by the project being above the mean in both the Cost Matrix as well as the Duration Matrix. This means the project will have been completed slower and more expensively than the mean.

2.

Visual Basic is most likely in the good practice because all the projects which used Visual Basic as its PPL were on average less complex than the rest.
Even if the study showed that Visual Basic projects always end up in the good practice quadrant it would be unreasonable for projects which were already in motion to change their PPL, as this would cost both time and money and is impossible to do over night.

3.

As this exercise needs us to go in depth into 3 different factors I will split them into sub exercises for the report.

3.1

In case the project used Scrum it would be interesting to take into account the length of the sprints. Having a sprint which isn't too long (2 months) but not too short (2 days) would probably make the project end up in the good practice quadrant. Otherwise it would probably end up in the bad practice quadrant. I believe this is the case because release based projects already have a higher chance of ending up in the good practice quadrant. Having a sprint of 2 days would result in "hacky" code, while having a sprint of 2 months would almost no longer be Scrum.

3.2

Amount of developers in the team. There is most likely an optimum number of developers which would make the project end up in the good practise quadrant. Any more or less would probably make it end up in the bad practise quadrant. Tracking how many developers worked on a project would give us an insight into how big the perfect development team should be.

3.3

Number of languages used in the project. In the studied paper they only took into account the primary programming language, not how many were used in total. There is most likely a correlation between the amount of languages used in a project and which quadrant it will end up in. We hypothesise less total languages will result in the project having a higher chance of ending up in the good practice quadrant, while more will result in having a higher chance of ending up in the bad practice quadrant.

Group 4

4.

We will describe the following 3 factors in depth:

1. Once-only projects
2. Technology driven
3. Dependencies with other systems

4.1

Once-only projects have a high tendency of ending up in the bad practice quadrant because everything has to be done for the first time, and will never be done again. This means there is no chance to learn while working on the project (which is the case for release based projects), which means that if you start with an inexperienced team you will also end with an inexperienced team. Also, because the project is once-only it will, on average, take longer than release based projects.

4.2

Technology driven projects are projects which mainly exist just to use certain (often new) technology. These projects have a high tendency of ending up in the bad practise quadrant because the developers are limited in what they can use. Something might be incredibly easy to fix by using some other technology, but because the project is centered around a certain type of technology it’s impossible to fix it that way. As a result the developers have to spend time figuring out how to work with the new technology, which costs time and money.

4.3

Projects that depend on other systems often end up in the bad practise quadrant because the systems the project depends on might be outdated or badly documented. A great example of this is working with legacy code. It might work, but no one who is still using it knows how or why. This results in time and money being wasted on figuring out how the system works.