Assignment 5

TI2206 Software Engineering Methods

Group 25
Pim Veldhuisen (4153448)
Jan-Gerrit Harms (4163400)
Amritpal Singh Gill (4419820)
Jeroen Vrijenhoef (1307037)

23 oktober 2015

Exercise 1

It was decided that game shall be extended to allow special jewels when longer jewel combinations are made by the player. When imploded, these special jewels will cover larger area and more jewels, which ultimately results in more points. The following user story and subsequent requirements describe this new functionality.
User stories

- As a user I want to gain special jewels when I make longer jewel combinations.

These user stories can be translated into following number of requirements:

Requirements

- Functional requirements
  - Must Have
    1. The game shall add a special jewel when player makes a combination of 4 or 5 jewels
    2. **Combination: 4 jewels**
       (a) The game shall add a 'Explosive jewel' when 4 jewel combination is made.
       (b) The game shall implode 'Explosive jewel' when it is involved in making combinations of 3 or more similar Jewels.
       (c) The game shall implode all adjacent jewels when 'Explosive Jewel' is imploded.
    3. **Combination: 5 jewels**
       (a) The game shall add a 'Hyper jewel' when 5 jewel combination is made.
       (b) The game shall implode 'Hyper jewel' when it is swapped with any normal jewel.
       (c) The game shall implode all jewels of 'x' color, present on the board, when the 'x' colored jewel is swapped with the 'Hyper jewel'.

- Should Have
  1. 'Flame Jewel' shall look like normal jewel with additional flames so as to make it easy for the player to find its similar jewel for making combinations with.
  2. 'Hyper Jewel' shall look distinctive from 'Flame Jewel' and from other normal jewels.
  3. The game shall add basic jewel points when a special jewel is imploded.
- Could Have
  1. The game shall give bonus points when special jewels are introduced into the game.
  2. The game shall give bonus points when special jewels are imploded.

- Non-functional requirements
  1. The documentation should be done before the 23rd of October, 23:55PM.
  2. The game shall be playable on Windows (7 or higher), Mac OS X (10.8 and higher), and Linux (Ubuntu 14.04 and higher).
  3. The code quality shall be verified using static analysis tools via the Maven framework.
  4. The functionality of the code shall be verified using test cases with at least 50% test coverage.
  5. The game shall be implementable in Java 8.
  6. The game shall be developed in an agile way, using (parts of) the SCRUM methodology.

Design

The Power Up feature was implemented using a decorator pattern. This way a jewel can be decorated to retain some of its properties, but also implement some new functionality. The class diagram for this Pattern can be seen in figure 1. The decorator pattern is used in several interactions. Figure 2 shows the sequence diagram for updating the sprites of a ExplosiveJewel; one of the PowerJewels. The ExplosiveJewel uses two sprites: the base sprite and an added one for the visualisation of the explosive effect.

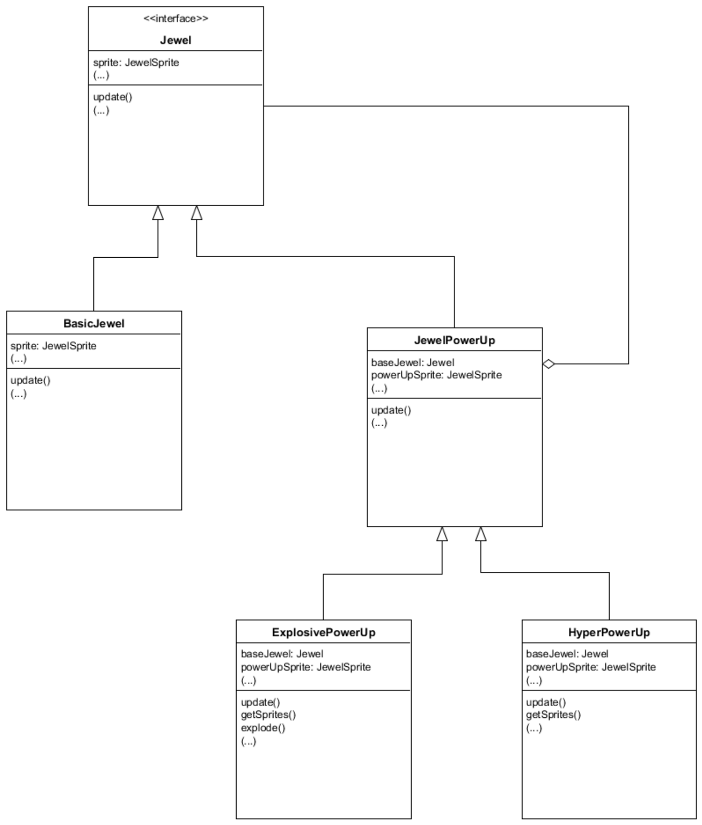
Exercise 2

Logger class (Singleton)

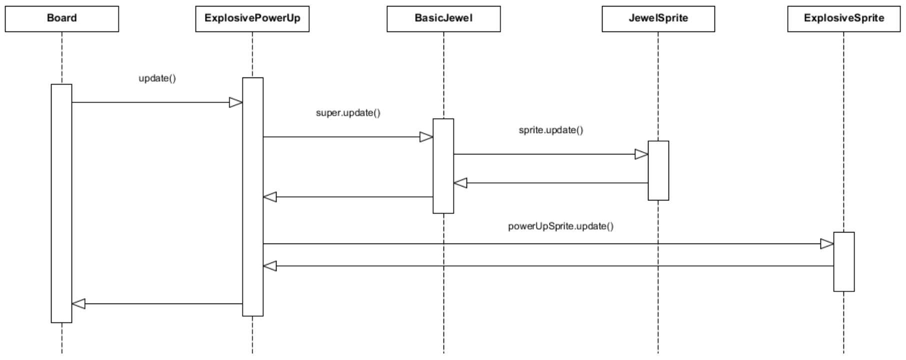
The singleton pattern is a design pattern that limits the instances of a class to one. That is throughout the lifetime of the main process only one instance of the class exists. This pattern is especially useful when one object is needed at many different levels of abstraction inside the project which would otherwise lead to a lot of dependencies. A good example is the Math class in Java which has static functions like $sin()$ and static attributes such as mathematical constants like π.

In our implementation of Bejeweled we used this pattern on the Logger class. The responsibility of the logger class is to print info, errors and warnings directly to a file for later review. It enables the developers to store debug information in a file and compare different runs of the program by comparing their log files. In this case it was useful to use the singleton pattern because printing can happen in any class at high and low abstraction level. Therefore a global instance can be used without introducing dependencies between the classes.

Figure 3 shows a class diagram of the Logger class. It consists of a set of static functions, one to enable logging, one to disable logging and a few for printing different types of messages. The class itself is part of it's own



Figuur 1: Class diagram of the decorator pattern used for PowerJewels

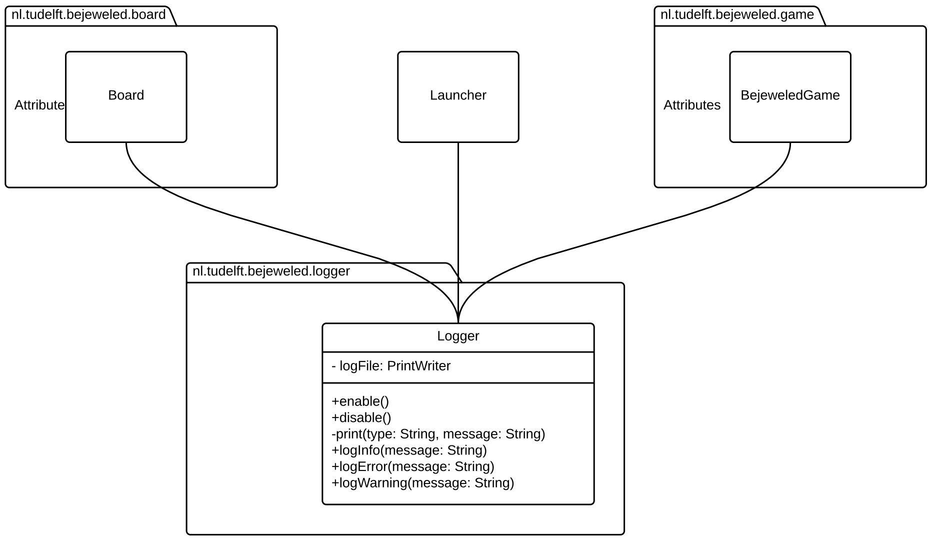


Figuur 2: Sequence diagram for updating the sprites of a ExplosiveJewel

package because it is not dependent on any other class and also no other class is dependent on it. Therefore it would not make sense to make it part of any existing package.

Other classes, for example Board, Launcher and BejeweledGame make use of the class. This is just an example, in the implementation any class could use the logger.

The sequence diagram in Figure 4 shows the the initialization and use of the Logger Singleton.


nl.tudelft.bejeweled.board
Attribute
Board
Launcher
nl.tudelft.bejeweled.game
Attributes
BejeweledGame
nl.tudelft.bejeweled.logger
Logger
- logFile: PrintWriter
+enable()
+disable()
-print(type: String, message: String)
+logInfo(message: String)
+logError(message: String)
+logWarning(message: String)

Figuur 3: Class diagram of the singleton Logger

Logging is enabled by passing the option $--logging\_enabled$ as commandline argument. This calls the function $Logger.enable()$ at the beginning of the main function of $Launcher$. This creates the logfile and sets the time stamp. This is all that is needed to initialize the logging.

The Logger can then be used by all classes for example Board, which logs the position of a combo that was made by the user or the BejeweledGui class which logs any errors that occur during the process.

When the window is closed, the $Logger.disable()$ function is called to close the logfile.

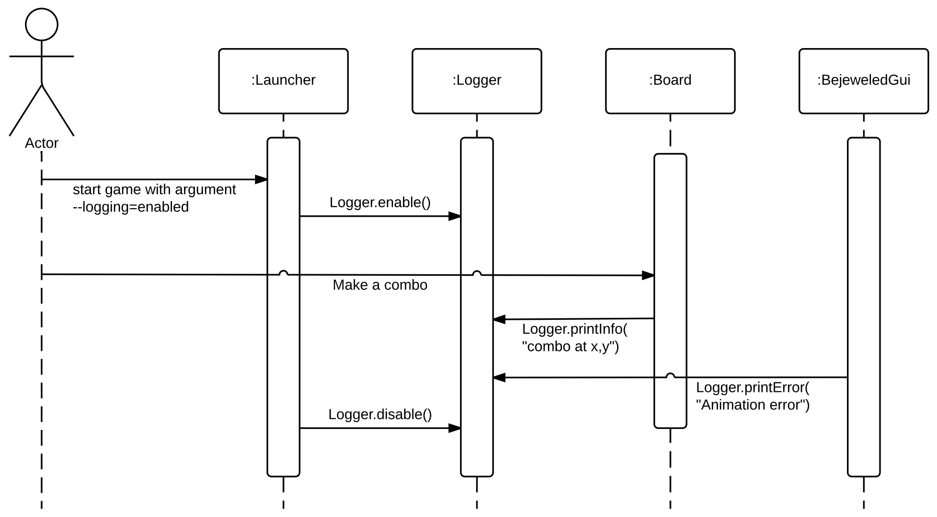
BoardMoveStrategy and BruteForceStrategy class (Strategy Pattern)

To enable us to dynamically change the algorithm that is used for finding possible combinations to be made on the board and simply because the Board class was getting really bloated we used the Strategy Pattern. The Board-MoveStrategy interface provides the interface for board to use the strategy and BruteForceStrategy is a concrete implementation of a strategy. The implementation as the name suggests consists of a brute force algorithm that checks every possible combination on the board for any valid moves.

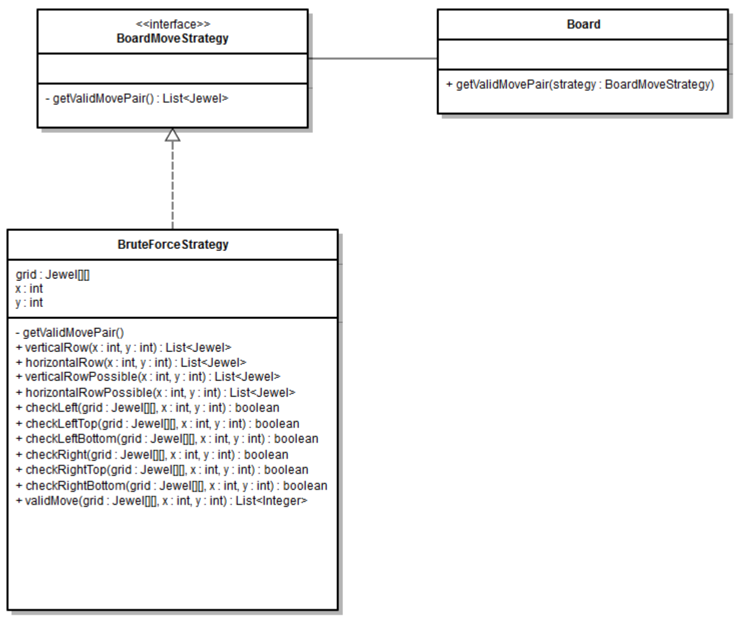
An UML overview of this implementation is shown in Figure (5) and an overview of how the code works dynamically is shown in Figure (6).

Exercise 3

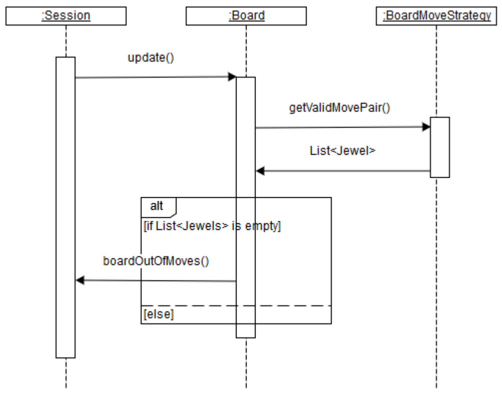
As a very good mixture of technical and management tasks, software engineering methods lab posed as an excellent learning experience. Our main goal in this lab was to design the Bejeweled game by making use of


Actor
:Launcher
:Logger
:Board
:BejeweledGui
start game with argument --logging=enabled
Logger.enable()
Make a combo
Logger.printInfo("combo at x,y")
Logger.printError("Animation error")
Logger.disable()

Figuur 4: Sequence diagram of a use case of the Singleton Logger


<<interface>>
BoardMoveStrategy
- getValidMovePair() : List<Jewel>
Board
+ getValidMovePair(strategy : BoardMoveStrategy)
BruteForceStrategy
grid : Jewel[][]
x : int
y : int
- getValidMovePair()
+ verticalRow(x : int, y : int) : List<Jewel>
+ horizontalRow(x : int, y : int) : List<Jewel>
+ verticalRowPossible(x : int, y : int) : List<Jewel>
+ horizontalRowPossible(x : int, y : int) : List<Jewel>
+ checkLeft(grid : Jewel[][], x : int, y : int) : boolean
+ checkLeftTop(grid : Jewel[][], x : int, y : int) : boolean
+ checkLeftBottom(grid : Jewel[][], x : int, y : int) : boolean
+ checkRight(grid : Jewel[][], x : int, y : int) : boolean
+ checkRightTop(grid : Jewel[][], x : int, y : int) : boolean
+ checkRightBottom(grid : Jewel[][], x : int, y : int) : boolean
+ validMove(grid : Jewel[][], x : int, y : int) : List<Integer>

Figuur 5: UML class diagram for the Strategy pattern.



Figuur 6: UML sequence diagram for the Strategy pattern.

effective and efficient software engineering methods. The whole design phase was divided into five assignments. We started with an idea of responsibility driven design and drafted our initial functional and non-functional requirements according to MoSCoW method. Of these, all must-have requirements were implemented into a very basic version of the game in first two weeks. Then as the lab progressed we added more features like tracking scores, displaying high scores, saving game state on exit, progressing levels and special jewels. Apart from these implementations, a study was also conducted related to the design of multiplayer feature with help of software engineering tools.
In order to maintain a control over increasing complexity as new features were added, the overall quality and structure of code was maintained by adopting design patterns, making use of code reviews and incode analysis. With aim to gain better understanding of the whole structure of the game, UML diagrams such as class diagrams and sequence diagrams were also used throughout the design phase.
Scrum methodologies constitute big part of our planning. It helped in enforcing better division of responsibilities. Scrum reflections were also very useful as they promoted learning from our prior mistakes. For instance, in assignment 1, two team members were assigned tasks that were dependent upon each other. This resulted in some problems very close to the deadline. Due to scrum reflection, special attention was given that week onwards while dividing such dependent tasks.

Lab taught us some good lessons on importance of team work, scrum, usage of github and many other parameters that could influence the implementation of a software system.
Effectiveness of scrum methodologies astonished us the most. The idea of steady heartbeat and product release in regular intervals, helped by making us aware of some bugs and design flaws in very early stages of design. For example, we realized it quite soon that two of our classes need refactoring that save us some efforts later when more features were implemented using these classes. We also witnessed the effectiveness of scrum plans while dividing responsibilities among team members.
Learning about something in theory is good but being able to actually apply it in real life promotes better understanding and knowledge. Through lectures, we were very certain about the importance of responsibility driven design, UML diagrams, design patterns, incode, code reviews and testing. But being able to actually apply them in lab strengthen our understandings.
Most of the group members had no good experience using github prior to this course. But since use of github was also graded during lab, we were forced to make a proper use of github, which eventually promoted better understanding. We learnt the importance of pull requests, making new branches and managing merge conflicts. We also learnt the usage and importance of travis testing and octopull along with our project.
Being from non computer science background, some team members also lack prior knowledge of eclipse and maven projects. Thus lab was the good platform for getting familiar with the usage of eclipse with github, maven,

testing, check styles, etc.

We started as a team of five with a goal to create the basic initial version of Bejeweled game during first two weeks. The task was quite tough for us as of five team members, only two were familiar with Java programming. Things got even worse when one of the team member dropped days before the first deadline. Despite all these difficulties we manage to gain descent grades for our initial version and in other weekly assignments.
Software engineering economics claim that an experienced team is a good factor and one of the major contributor to the success of software engineering projects. But after spending lot of time building our game, we now believe that though experienced team increase the chances of success, but it could be compensated with proper planning (scrum), motivated team members who could dedicate proper time towards their responsible tasks, and with proper communication and understanding among team mates.
We also believe that a motivating boss could bring the best out of a team of software engineers. Bastiaan, the assigned teaching assistant for our team, has been a motivating boss throughout the lab. He never failed to mention anything good that he manages to find in our submissions. This kind of positive behavior has inspired our team from the very beginning of this course. We think that this was one of the main factors behind our team being on right track for this long.
Working in a team requires overcoming conflict of interests and operating as one unit. We in a team of four have managed quite well in this aspect. There was no instance when a team member failed in completing his job, or any kind of conflict with preference of a particular task. All were patient and polite enough to give other member a chance to make the first choice. Everyone displayed good understanding and welcome new suggestions and ideas with open hearts. Overall we never had any kind of problem in any of these aspect.

We learnt many essential things during the period of this course, and without doubt they will prove very useful in future when we get the chance to design and implement new software systems. Now we very well know the importance of proper planning in software engineering. Thus our first approach towards any new task would be to make concrete and effective plans.
Another important aspect is team work. Our approach will be to make efforts in order to resolve any kind of conflict within a team at early stages of operation. We will try to promote better understanding regarding the whole project and responsibilities among all. We will make use of scrum methodologies, UML diagrams and code reviews for these purposes.
We also learnt that any unfamiliar task could be handled with proper planning and good team. Thus in future, unfamiliar tasks and new challenges will be welcomed with open hearts.