Assignment 1

TI2206 Software Engineering Methods

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

18 september 2015

Exercise 1

1. To derive classes, responsibilities and collaborations from our requirements document we first individually wrote down all nouns from the requirements document and selected candidates from the lists. For these candidates we looked for verbs in the requirements document expressing the kinds of functionality these candidates should provide for the game. Finally we asserted for all these classes what kind of information is needed from other classes to perform these responsibilities. The classes selected from the requirements document are shown in the table below. The classes we found by using the RDD methods match the

Class	Responsibilities	Collaborations
Board	Generate clear combos	Jewel Game
Jewel	Visualize the jewels	
Game	Visualize the board	Board
Menu	Provide an entry point to slect game mode, highscore or exit	Game HighScore
HighScore	save, load and insert highscore	Game
ClassicGame	calculate the classic game logic	
TimedGame	calculate the timed game logic	
SoundEffect	play sounds and background music	Board Game

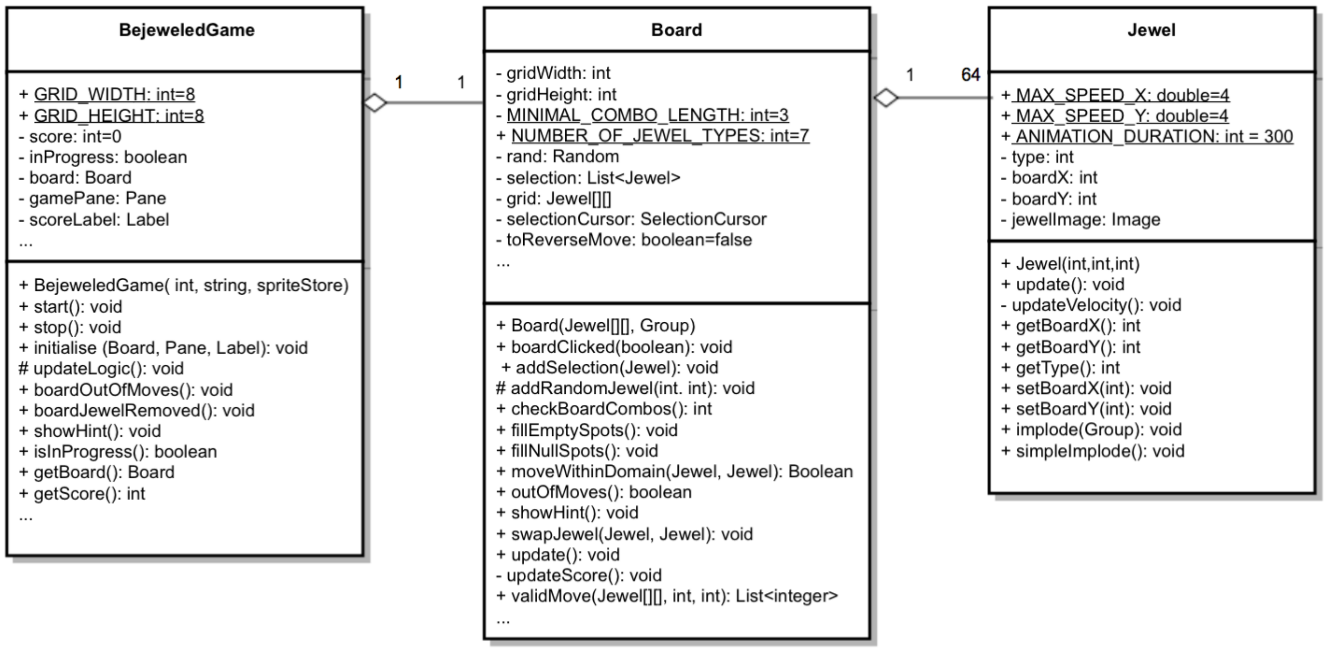
implemented classes pretty well. One thing that differs is the fact that we took all requirements into consideration for the RDD, while we only implemented those with the highest priority. There are also some additional classes implemented that have to do with the inner workings of the software, such as the BoardObserver, BejeweledGui and BoardFactory classes. Because these classes operate behind the scenes, they are not immediately apparent in RDD.

2. We consider BejeweldGame, Board and Jewel the main classes, as they take care of the central game mechanics. The jewels are the atomic elements of the game that are manipulated. They are relatively simple objects, and thus have a limited amount of methods. The class is primarily responsible for the

display and animation of the jewels. The jewels are instantiated by the board class. The board is where the game happens by swapping jewels and calculating the effects of these moves. This includes among other things checking for combos, letting jewels fall down by gravity, and generating new jewels in empty spots. The board class also collaborates with jewels when they disappear; the board starts the fade-out animation and checks when it is finished. The board is instantiated in the BejweledGame class. The game class is responsible for running the actual game, i.e. checking for starting the game, continuously updating the logic and rendering the game. For this it collaborates with the other main classes.

3. While the other classes do not touch the core game functions, they each serve a useful function that is distinct from the other classes' functionality. For example, the SelectionCursor and the Logger each add some functionality; visually indicating the selection and logging events respectively. These are separate tasks that can be best done by separate classes to divide the responsibilities and to reduce the interdependencies of different functions. Other classes, such as the Launcher and the BejeweledGui cover implementation details that are best kept seperate from the main logic.

4. Figure (1) shows the class diagram of three main classes: BejeweledGame, Board, Jewel

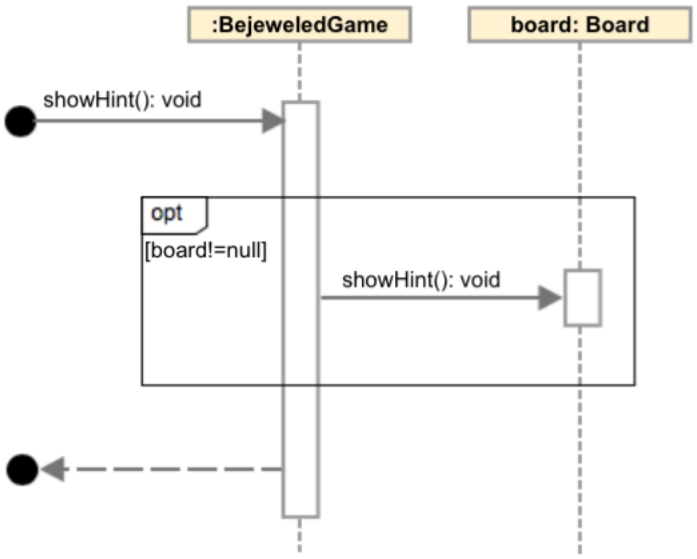


Figuur 1: UML class diagram for main classes in the Bejeweled game.

5. Figure (2) shows the sequence diagram when a hint is to be displayed. Figure (3) shows the sequence diagram for swapping two jewels.

EXERCISE 2

1. The difference between aggregation and composition is that in the case of composition the object can not live without it's owner class. So when the owner is destroyed the object also gets destroyed (in Java language you can implement this using the final keyword when declaring a class variable). This is in contrast with aggregation where the object can live on without it's owner class.



Figuur 2: UML sequence diagram for providing hint.

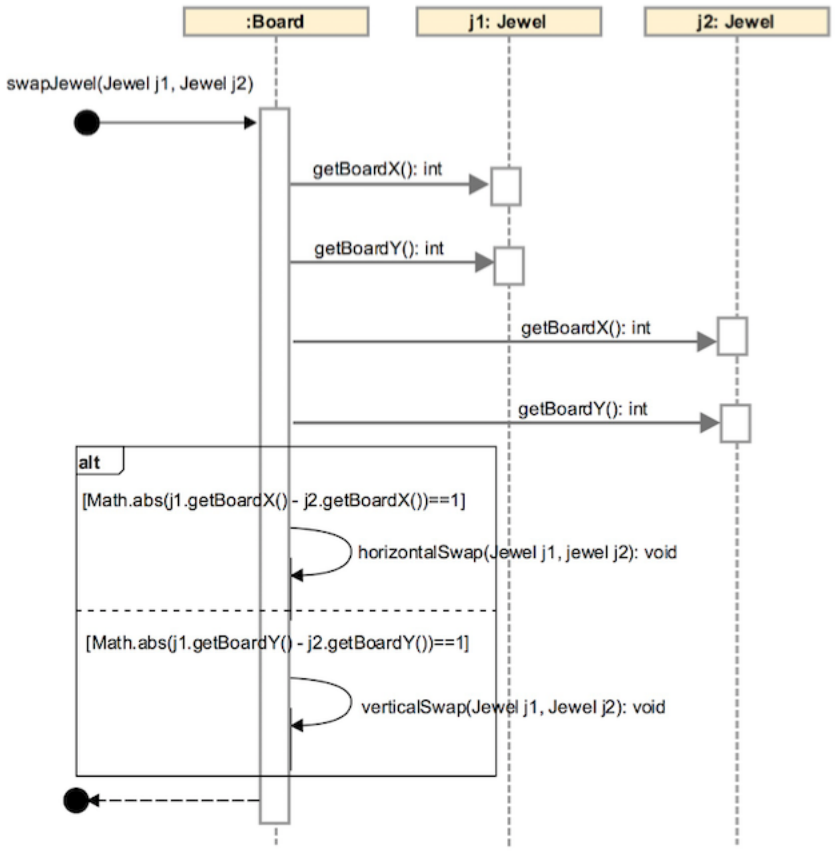
Aggregation is used in the following classes:

- Board - The board class aggregates sprite objects like the Jewels and the SelectionCursor so it can perform operations like swapping jewels, checking for combo's and importantly letting the sprites know when they are moved. The animations themself are taken care of in the Sprite class itself.
- BejeweledGame - The BejeweledGame class aggregates the Board class. It's the responsibility of the BejeweledGame class to convey any user input to the board class (that has all the game logic) and in return the BejeweledGame class receives notifications about the status of the game from the Board class and conveys these back to the user.

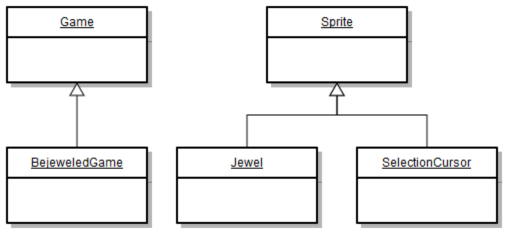
2. Parametrized classes are not used in our project. Parametrized classes can be used when a class performs the same behaviour for a set of other classes. If that is the case you can make a generic type declaration of that class so you can use a type variable (that can be anything) in your class.
3. Figure (4) shows where inheritance is used in the game (they are 'is-a' relationships).

BejeweledGame is a Game and both SelectionCursor and Jewels are Sprites. The abstraction for the Game class was chosen so that all code needed to make a general game (setting up gameloop etc.) is not specific for our Bejeweled game implementation. The abstraction of the Sprite class is chosen because all sprites in the game have similar properties and functionality (x/y positions, they can be moved etc.). Figure (5) shows where polymorphism is used in the game.

The CustomBoardLauncher class has exactly the same functionality as a normal Launcher except it overrides one method (makeBoard()) so it can make the board from a predefined file instead of randomly generated boards like the normal launcher.


:Board
j1: Jewel
j2: Jewel
swapJewel(Jewel j1, Jewel j2)
getBoardX(): int
getBoardY(): int
getBoardX(): int
getBoardY(): int
alt
[Math.abs(j1.getBoardX() - j2.getBoardX())==1]
horizontalSwap(Jewel j1, jewel j2): void
[Math.abs(j1.getBoardY() - j2.getBoardY())==1]
verticalSwap(Jewel j1, Jewel j2): void

Figuur 3: UML sequence diagram for swapping two jewels.



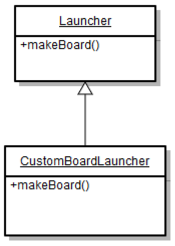
Figuur 4: UML diagram of inheritance used in the Bejeweled game.

EXERCISE 3

A feature to be added to the game is the logging of the game actions. The following user stories relate to this feature:

User stories

- As a user I want to be able to log the game events for testing and debugging purposes.



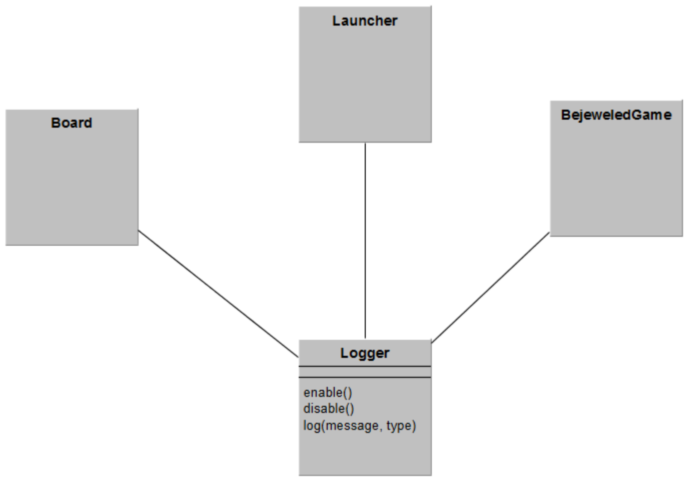
Figuur 5: UML diagram of polymorphism used in the Bejeweled game.

- As a user I want to be able to save the log to a file so I can read it later.

These user stories can be translated into an number of requirements:
Logger requirements

- The logger shall be able to be enabled and disabled when necessary.
- The logger shall be able to store messages.
- The logger shall be able to store the type associated with each message.
- The logger shall store the logs for each session in a different file.
- The logger shall name each file such that the name includes the date and time of the session.

A simplified UML diagram for the logger is shown in figure 6. The logger has been implemented and tested successfully.



Figuur 6: Simplified UML diagram for the Logger class