Assignment 3

TI2206 Software Engineering Methods

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

9 oktober 2015

EXERCISE 1

1. We decided to add different levels as an extension of the game. The following user story and subsequent requirements describe the new functionality.

   User stories

   - As a user I want to be able to progress to a new level when I perform well.

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

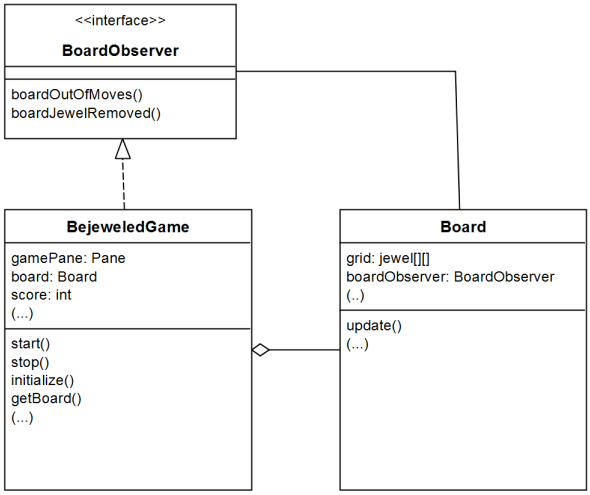
   Requirements

   - Functional requirements
     (a) The game shall progress to the next level when the player's score is equal to or higher than 500 times the current level squared.
     (b) The game shall multiply the amount of points scored when removing Jewels by the current level.
     (c) The game shall create a new board when the next level is reached.
     (d) The game shall display the current level.
     (e) The game shall include the current level when saving and loading.
   - Non-functional requirements
     (a) The implementation should be done before the 9th of October, 23:55PM.
     (b) The game shall be playable on Windows (7 or higher), Mac OS X (10.8 and higher), and Linux (Ubuntu 14.04 and higher).
     (c) The game shall be implemented in Java 8.
     (d) The code quality shall be verified using static analysis tools via the Maven framework.

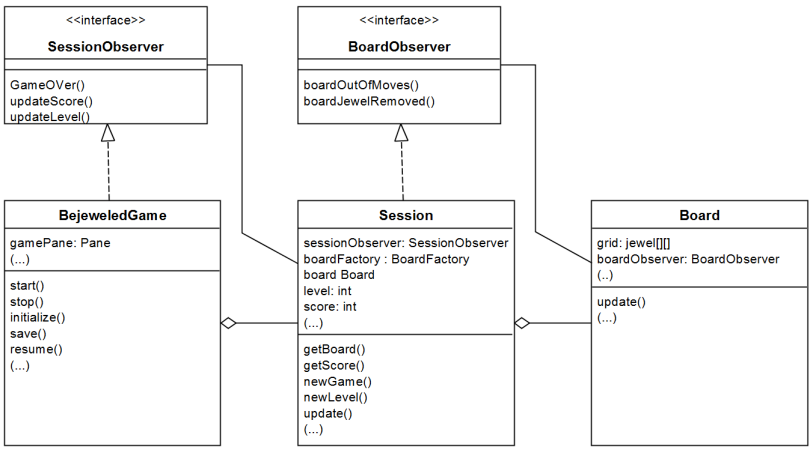
(e) The functionality of the code shall be verified using test cases with at least 50% test coverage.

(f) The game shall be developed in an agile way, using (parts of) the SCRUM methodology.

2. During the process, responsibility driven design was used. The new feature of levels created a new distinct responsibility; progressing through levels. For this responsibility a separate class was created; the Session class. In the hierarchy, this class is inserted between the board and the BejeweledGame. The Session class also takes over some responsibilities from the BejeweledGame. The Session creates the board at the start of the game. This makes sense because the Session should also create a new board when the player reaches a new level. The Session also keeps track of score, as this is also closely related to level progress. The UML of relevant classes before the addition of the Session class is shown in figure 1, the UML after in figure 2. As shown, the interaction between the classes is modeled using Observers.



Figuur 1: The UML of relevant classes before the addition of the Session class.

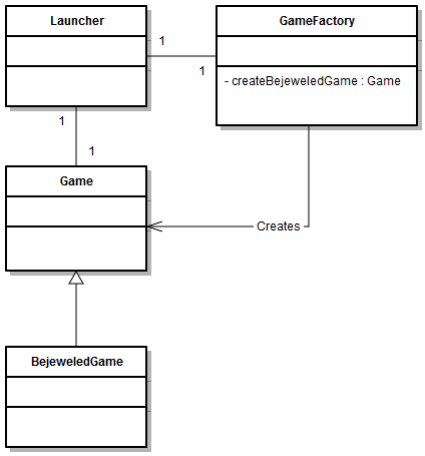


Figuur 2: The UML of relevant classes after the addition of the Session class.

EXERCISE 2

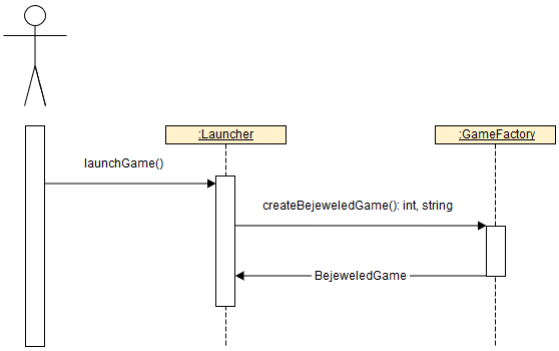
GameFactory (Factory Method, Creational Pattern)

1. Because the Launcher should be able to launch any Game and not just our implementation BejeweledGame, there is a factory for the Game class. The factory can be used by the Launcher to create a Game object without specifying the concrete class (in this case BejeweledGame).This way the factory is responsible for creating the correct Game object. This has been particularly useful creating the CustomBoardLauncher that launches a game with a previously defined board configuration (used in testing).
2. UML diagram showing the implementation of the Factory Method pattern is shown in Figure (3).



Figuur 3: UML class diagram for the Factory Method pattern.

3. UML sequence diagram showing the sequence of events that use the factory is shown in Figure (4). First the user launches the game, then the launcher calls createBejeweledGame() in the GameFactory to create the game object to launch.



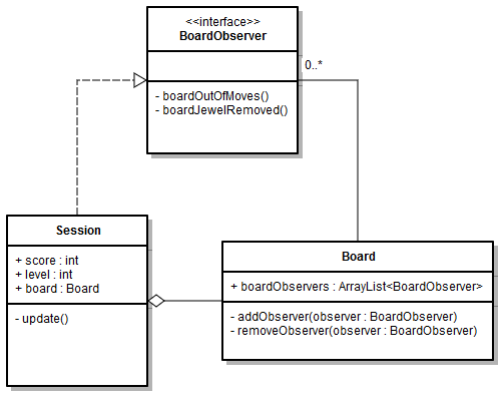
Figuur 4: UML sequence diagram for the Factory Method pattern.

BoardObserver-Session-Board (Observer, Behavioral Pattern)

1. This pattern allows our board class which is responsible for maintaining the board to communicate useful information to the Session class, which in turn is responsible for keeping game information like the score

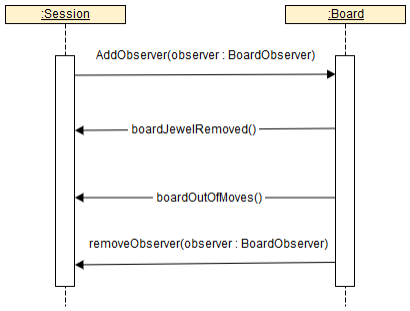
and current level. By using only a common interface called BoardObserver this can be achieved without requiring the two classes to know anything about each others implementation. This ensures a loose coupling between Board and Session. This could also become useful for integration testing purposes of the Board and Session classes. Although we did not implement this because of time constraints.

2. UML diagram showing the implementation of the Observer pattern is shown in Figure (5).



Figuur 5: UML class diagram for the Observer pattern.

3. UML sequence diagram showing how the observer pattern is used is shown in Figure (6). After the board object is created the Session object will register itself as an observer for the Board. The board in turn will give updates about Jewels that have been removed (for keeping score) and when there are no more moves left (game over). After that the Session object can remove itself from the board as an observer.



Figuur 6: UML sequence diagram for the Observer pattern.

EXERCISE 3

1. The paper analysed 352 finalized software engineering projects within two large banks and a telecom provider, that were collected in measurement repository. The overall average performance with regards to their size, cost and duration was computed. Corrected for the applicable project size, the project was accessed as good practice or bad practice when its performance on both cost and duration was respectively better or worse than the average cost and duration.

   Based upon these, It enumerated factors that were strongly strongly related to a high percentage of good

practices (10 factors: Steady Heartbeat; Fixed, experienced team; Agile (Scrum); Business Domain Data Warehouse BI; Business Domain Organization; Programming Language Visual Basic; Release-based (one application); Programming Language and programming language SQL) and bad practices (13 factors: Security Many team changes; inexperienced team; Business domain Mortgages; Bad relation with external supplier; New technology, Framework solution; Rules Regulations driven; Dependencies with other systems; Legacy; Migration; Technology driven; Client Account Management; Once-only project and Pilot, Proof of Concept). Thus software engineering projects with more number of good practice factors are expected to belong t0 good practice group, and likewise the projects with more number of bad practice factors are more likely to fall in bad practice group.

2. The paper assigned visual basic to good practice group with mentioned associated probable reason to be the less complex environment of visual basic. But the exact reason for this case is no where specified. It is because the project data was collected prior to the start of the research, making it difficult to dig this exact reason.
   Also, this conclusion was made based on the analysis of 6 projects out of which 5 score good practice and one as Cost over Time. Thus the source data used is quite limited in amount and might need more data to be analysed before concrete conclusions could be made.
   Even though statistical tests proves visual basic to be a good practice group, yet exact reason for this is unknown, making the information bit less intriguing.

3. Following factors could also have been studied in the paper:

   - **Automatic testing**: Test automation involves the use of special software that is applied on software under development and is responsible for executing tests, and comparing predicted outputs with the actual outputs. It is believed to be a good practice because automatic testing could point out bug in the very initial stages of development. Thus preventing loss of time and money in later stages when this bug amplifies in amplitude because of the building complexity, with time, of system under development.
   - **Multicultural team**: Multicultural team consists of team members from different cultural and/or national backgrounds. It is considered as good practice given the cultural clashes are sorted out in the initial stages of team operation. Team members from different background bring along different prospective of looking at things. This could help in coming up with creative solutions to many complex problems. This different point of view could result in more discussions on aspect of conflicts, that could further result in better analysis of these aspects, which finally promote better understanding.
   - **Team strength per functional point**: The team strength plays a significant role on its overall performance. Philosophically, the team should be big enough but no bigger than that. More than required team members will result in more conflicts at every stage of team operation, will result in difference in understandings of goal, which would ultimately results in poor performance. On the other hand, less than required team members will result in more stress and work load per member, which will result in the degradation of the quality of design. Thus a team with appropriate number of team members belongs to a good practice group. The paper could have incorporated this into study and might have given some conclusion on the number of team members, of which a team should consist of.

4. The bad practice factors:

   - **Rules & Regulations driven**: It refers to the rules and regulation to which the member of team working on software project are bound to. These could be related to the the regulations on style of coding, working hours, hard deadlines, etc. This is considered as bad practice as it might result in the team members not being able to exercise their skills to its fullest. For instance, regulations on working hours might be bad for a developer who has developed a habit of working late at night.
   - **Dependencies with other systems**: It refers to the dependencies on systems that are developed and

maintained by some other team or firm. This is considered as bad practice as any change in the depender system might induce a bug in the dependee requiring further revisions, which ultimately results in extra cost and time.

- **Technology Driven**: It refers to the management philosophy that pushes the development of current system based on the technical abilities of a firm. It is considered as bad practice because technology is in ever changing phase to which software development should adopt to. For instance, a firm dependent on technologies such as floppy disks could not survive in the era of solid state drives.