Assignment 3

Group 20 Bejeweled

Exercise 1

Exercise 1.1

Animations elaboration

In sprint plan 3 we described that we were going to make the jewel swaps more visible by using animations. There was also a could have in our requirements regarding the smooth animation (meaning frame per frame movement). We did not implement this because the structure of our code was not compatible with this kind of implementation. The only way to solve that was to refactor our code, however it was not possible in this timeframe from our estimation so we decided to implement aminations differently as described below.

Before sprint 3
There was no visible way to see that a jewel actually moved from one place to another during the swap and also when there was no match (for example you have {red, red, blue} next to each other and blue was swapped for green, nothing would happen). When there was a valid match of jewels, the matching jewels were removed instantly from the board and the empty spots that appeared were instantly filled by new jewels. The user had no idea what happened at that instance, new jewels suddenly appeared without delay on the board.

After sprint 3
Animations have been implemented and they have added more feel to the game. When there is an invalid match, the jewels that were being swapped change their positions with each other, so you can visibly see that they swap. However, since the swap is invalid, the jewels are changing back to their original position. Whenever there is a valid match, the matching jewels first show themselves after the swap and then disappear from the screen, leaving (visible) empty spaces. After a short delay (smaller than 1 second), the jewels above these empty spots drop down to fill the gaps. Because of the filling of the gaps, new gaps are introduced in the top rows. These gaps are then filled with new jewels. When a match is made as a result of dropping down jewels and/or adding jewels, these are also visibly processed in the same manner as described above. This process of clearing jewels, filling gaps and adding new jewels, makes it clear to the player what is actually going on.

Testing

We can clearly see the changes above and sprint plan 3 mentioned the term testing, however there was no explanation of it. The changes of the animations have to be tested and we thought of Junit testing and manual testing. However, after the implementation of animations we thought there was no easy or feasible way to use Junit to test the changes of this sprint plan. The problem is that we work with delays during swaps and during the filling process and it is not feasible to test the changes that happen exactly during those delays, because for example you would have to test that there is in fact a delay occurring and that during that specific delay an invalid swap animation is made and swapped back after another delay. Besides, even if those things are easily testable, the animations have to be visible by the user which is displayed with our gui class, so you would have to somehow test that a user can actually see what is happening during those delays. As you can see the problems are piling up, and we decided to test it as a user by using manual testing to have a more effective way of testing the changes since those changes are easily noticed by a user.

Manual testing

For manual testing we thought of some scenarios and user stories to test our animations.

User making a move in the game

As a user, I want to be able to see that I made a swap so that I know that I made a valid or invalid move.

Scenario 1: invalid swap

Given that a user has selected two adjacent jewels to swap and there is no match. When the jewels are selected, the jewels swap positions for a small delay (less than 2 seconds) and swap back afterwards to their original location.

Scenario 2: valid swap

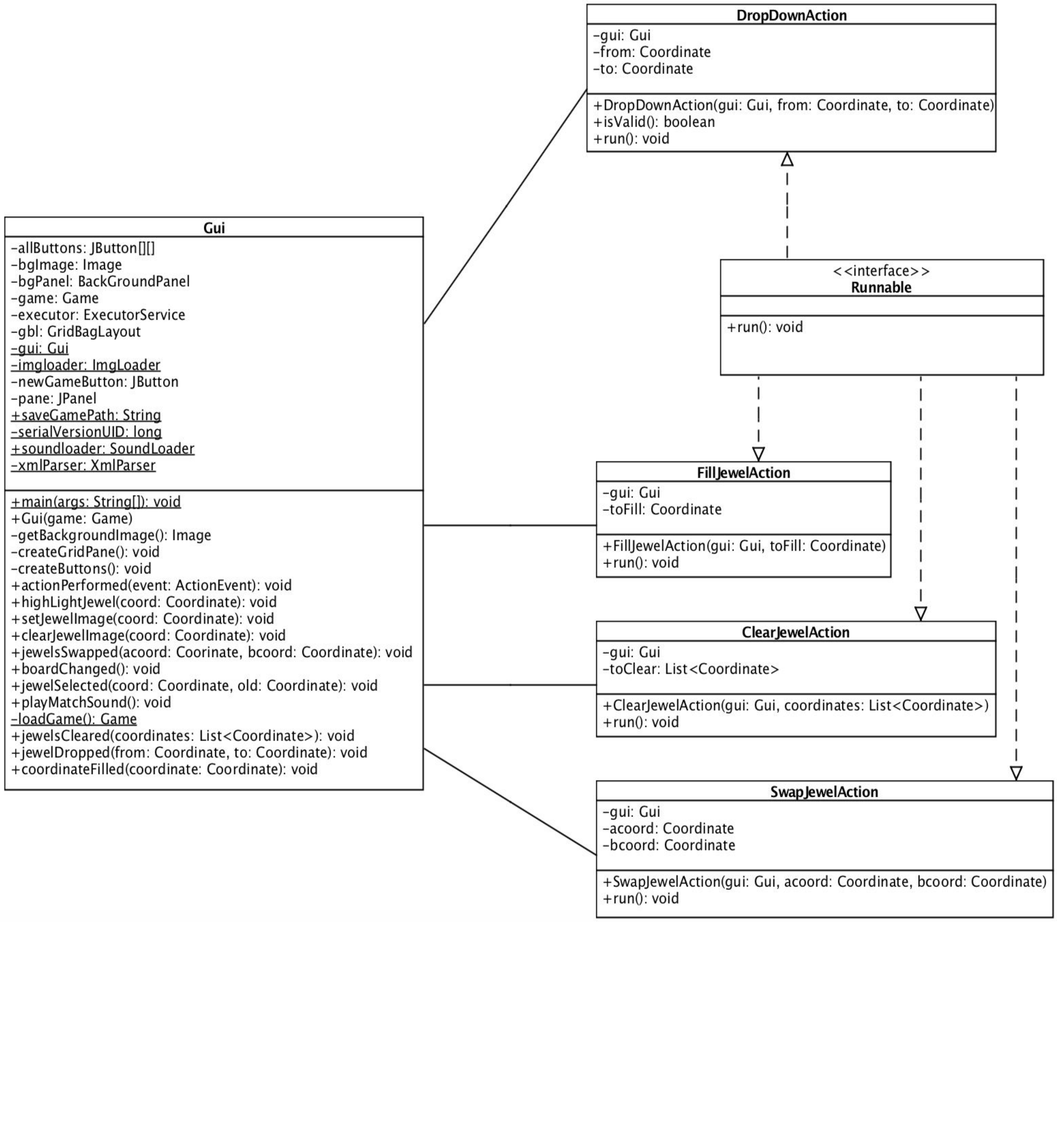
Given that a user has selected two adjacent jewels to swap and there is a match. When the jewels are selected, the jewels swap positions for a small delay (less than 2 seconds) and afterwards the matching jewels disappear from the screen.

Scenario 3: fall animation

Given that a user has made a valid swap. When the matching jewels have disappeared from the screen, then the jewels that are above the match should drop down.

Exercise 1.2

Animations UML Class Diagram


DropDownAction
-gui: Gui
-from: Coordinate
-to: Coordinate
+DropDownAction(gui: Gui, from: Coordinate, to: Coordinate)
+isValid(): boolean
+run(): void
Gui
-allButtons: JButton[][]
-bgImage: Image
-bgPanel: BackGroundPanel
-game: Game
-executor: ExecutorService
-gbl: GridBagLayout
-gui: Gui
-imgloader: ImgLoader
-newGameButton: JButton
-pane: JPanel
+saveGamePath: String
-serialVersionUID: long
+soundloader: SoundLoader
-xmlParser: XmlParser
+main(args: String[]): void
+Gui(game: Game)
-getBackgroundImage(): Image
-createGridPane(): void
-createButtons(): void
+actionPerformed(event: ActionEvent): void
+highLightJewel(coord: Coordinate): void
+setJewelImage(coord: Coordinate): void
+clearJewelImage(coord: Coordinate): void
+jewelsSwapped(acoord: Coorinate, bcoord: Coordinate): void
+boardChanged(): void
+jewelSelected(coord: Coordinate, old: Coordinate): void
+playMatchSound(): void
-loadGame(): Game
+jewelsCleared(coordinates: List<Coordinate>): void
+jewelDropped(from: Coordinate, to: Coordinate): void
+coordinateFilled(coordinate: Coordinate): void
<<interface>>
Runnable
+run(): void
FillJewelAction
-gui: Gui
-toFill: Coordinate
+FillJewelAction(gui: Gui, toFill: Coordinate)
+run(): void
ClearJewelAction
-gui: Gui
-toClear: List<Coordinate>
+ClearJewelAction(gui: Gui, coordinates: List<Coordinate>)
+run(): void
SwapJewelAction
-gui: Gui
-acoord: Coordinate
-bcoord: Coordinate
+SwapJewelAction(gui: Gui, acoord: Coordinate, bcoord: Coordinate)
+run(): void

Animations requirements

Functional requirements

1.1 Must Haves

1. Whenever the player swaps two jewels, the jewels must be (visibly) swapped on the screen.
2. Whenever the player has swapped two jewels, but no match is made, the jewels must be swapped back on the screen after a short delay. (The length of the delay will be determined by trying different values until we all agree on the looks of the on-screen swaps).
3. Whenever the player has swapped two jewels and one or more matches are made, the jewels that form a match should be replaced by empty spaces, so that the matches seem to disappear from the board.
4. After matches are cleared from the board, the jewels above the cleared spaces should drop down after a short delay. (The length of the delay will be determined by trying different values until we all agree on the looks).
5. After all the jewels above the cleared spaces have dropped down, the newly created empty spaces must be filled after a short delay with newly created jewels, until no empty spaces remain. (The length of the delay will be determined by trying different values until we all agree on the looks).

1.2 Could Haves

1. The on-screen swaps and moving of jewels could be smoothly animated during the delays instead of instantly after the delays (meaning, the jewels are actually moving on-screen).

Nonfunctional requirements

1. A fully functional version of the animations will be delivered on 09-10-2015.
2. The animation functionality will be developed using the Scrum software development methodology. (The development will take one Scrum iteration)
3. The time/level function will be developed using the Responsibility Driven Design technique.

Exercise 2

The Composite Design Pattern

The composite pattern is a design pattern which builds up structures of an object in a tree style way. This pattern should be used when a set (composition) of objects has to be treated the same way as one individual object. In our project, one Match can consist of three or more Jewels. However, it could also consist of three or more Jewels and another match. Both should be treated as one Match. For example, an L-shaped match, with three jewels vertically and three jewels horizontally, needs to be treated as a single Match, although it consists of two matches. The UML diagram given in Exercise 2b. describes how the pattern is implemented in our project.

The Decorator Design Pattern

For this specific situation (the match representation in our project), we have also considered the Decorator Pattern, but we have concluded that it isn't suitable. This is because the Decorator Pattern is used to add new functionalities to an object. However, we are not adding new functionalities to Match objects. We only need both a single Match and a composition of Matches to be treated equally. For this reason, we have used the Composition pattern, which does exactly that.

Exercise 2.1 (1)

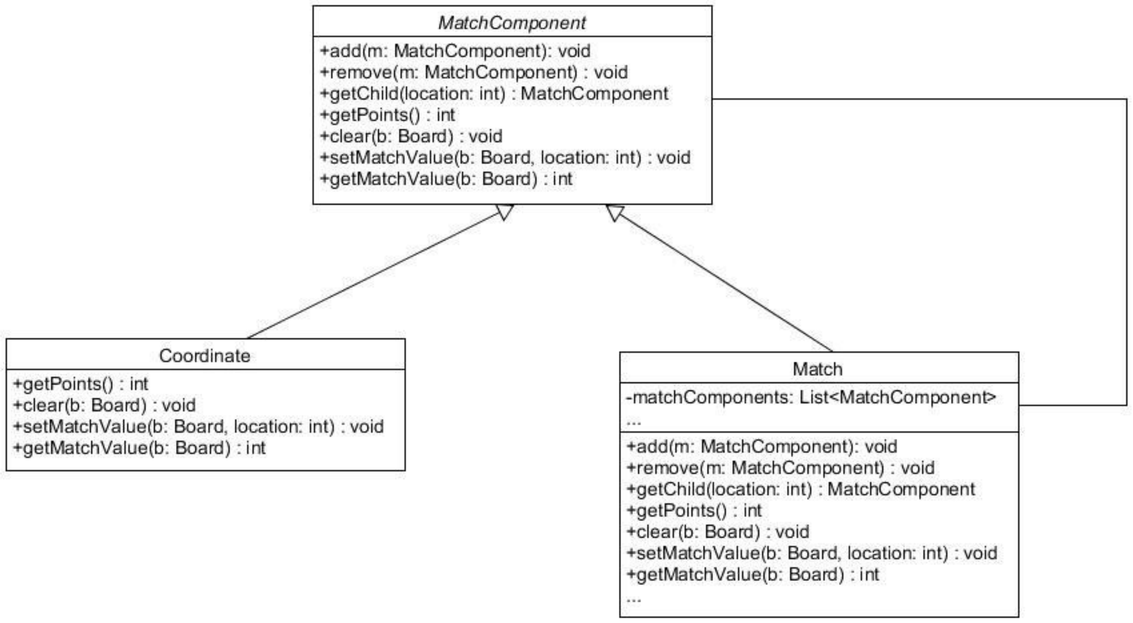
Composite Design Pattern:

We decided to implement the composite design pattern as one of the design patterns. We use it for the Match class: We want that a Match can contain Coordinates and/or other sub-Matches. We then want to calculate the score for the entire Match (with an additional bonus when having a Match in a Match) For this situation, the composite pattern is best: An Object which contain a few other Objects, which can also be of the same instance of your main-Object. You can then calculate the Score of the entire Match by just calling the getPoints() method on the main Match.

We implemented the design pattern in the following way: We created an abstract class MatchComponent. This describes the Objects that a Match can hold. We then extend this MatchComponent to two of our already implemented classes: Match and Coordinate. In the Match class, we change the arrayList<Coordinate> to arrayList<MatchComponent>, so that a Match can hold either. This also required a change in the Board class, because clearMatch(Match) only worked for Coordinate objects. Finally, we also made a check while obtaining the Matches from the getMatches method to see if a Coordinate is already in another Match.

Exercise 2.2 (1)

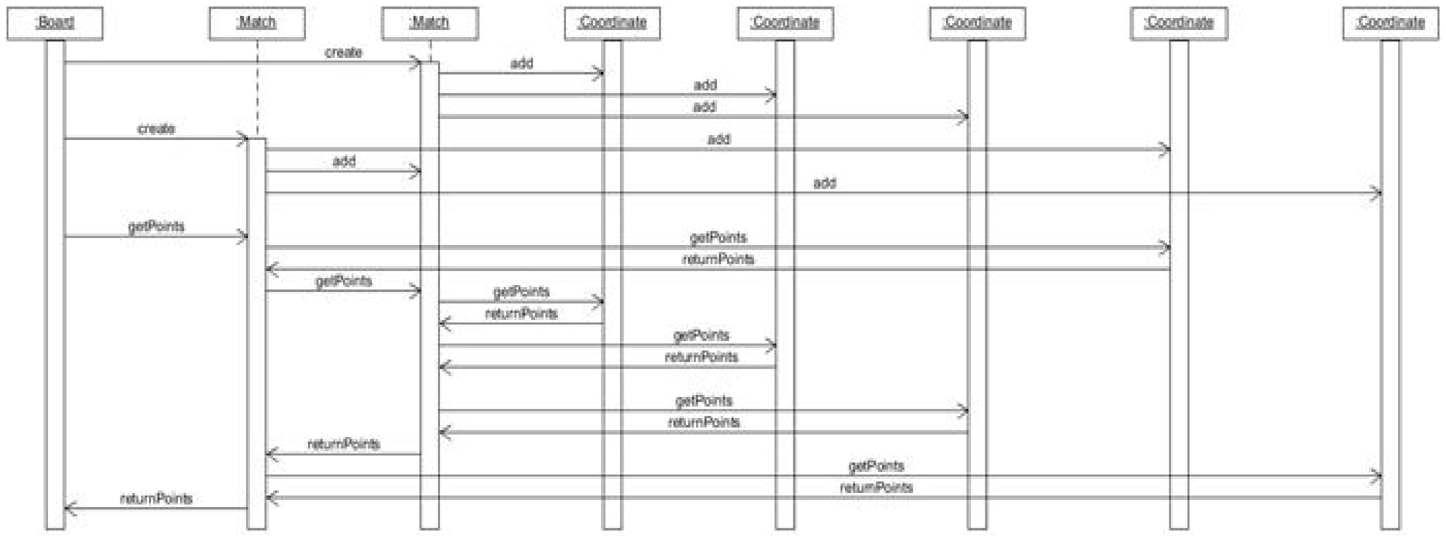
Composite Design Pattern Class Diagram:



MatchComponent
+add(m: MatchComponent): void
+remove(m: MatchComponent) : void
+getChild(location: int) : MatchComponent
+getPoints() : int
+clear(b: Board) : void
+setMatchValue(b: Board, location: int) : void
+getMatchValue(b: Board) : int
Coordinate
+getPoints() : int
+clear(b: Board) : void
+setMatchValue(b: Board, location: int) : void
+getMatchValue(b: Board) : int
Match
-matchComponents: List<MatchComponent>
...
+add(m: MatchComponent): void
+remove(m: MatchComponent) : void
+getChild(location: int) : MatchComponent
+getPoints() : int
+clear(b: Board) : void
+setMatchValue(b: Board, location: int) : void
+getMatchValue(b: Board) : int
...

Exercise 2.3 (1)

Composite Design Pattern Sequence Diagram:



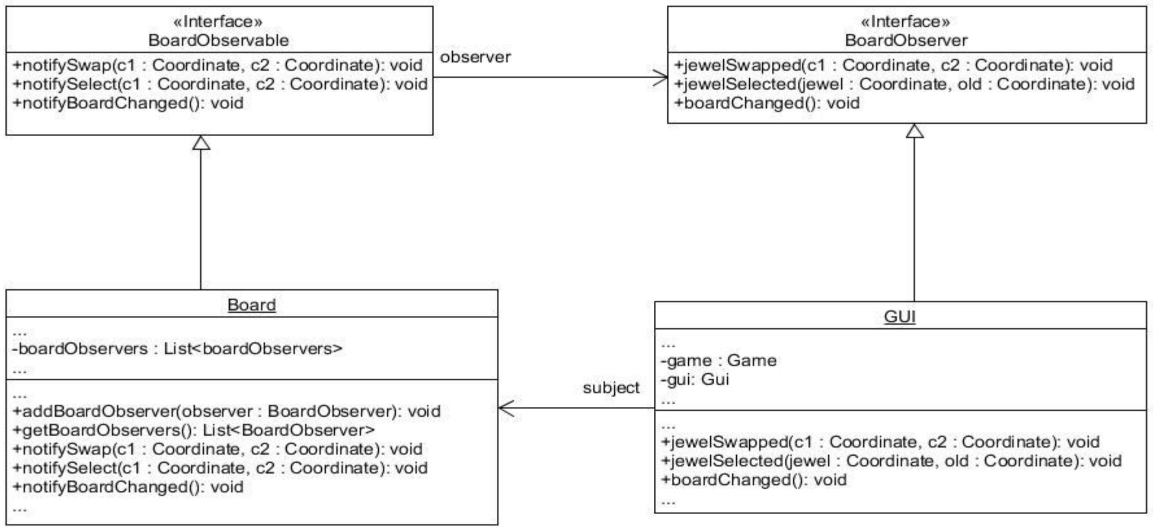
Exercise 2.1 (2)

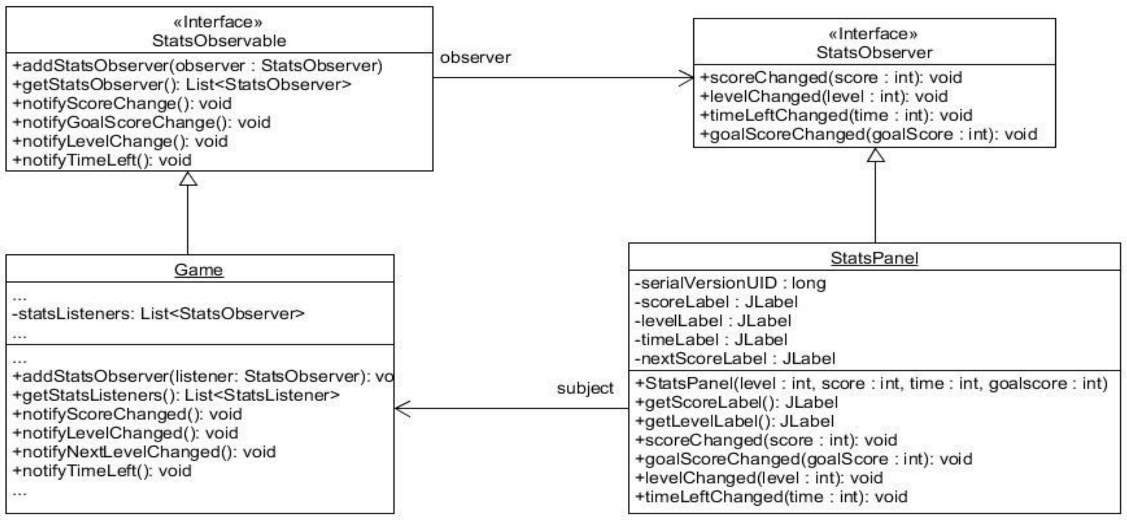
Observer Design Pattern:
We chose to use the observer design pattern because we found the fact that it is used to manage the relationship between one class and its dependants useful. Specifically because it can be used to notify and update the dependants of a class automatically. For example, the Board class is a pretty big class; and as such, it is very important that it's relationship with its dependant is managed properly. Therefore it is useful for us to have an observer that can help us monitor that relationship.

We implemented the observer design pattern by utilizing the Listeners that were already in our project, and creating/updating the interfaces that would be used by the observers and the subject. By updating their interfaces of the listeners, creating the observable interface for the subject of the observer, and making sure that they were consistent with the observer design pattern, we were able to implement the observer design pattern.

Exercise 2.2 (2)

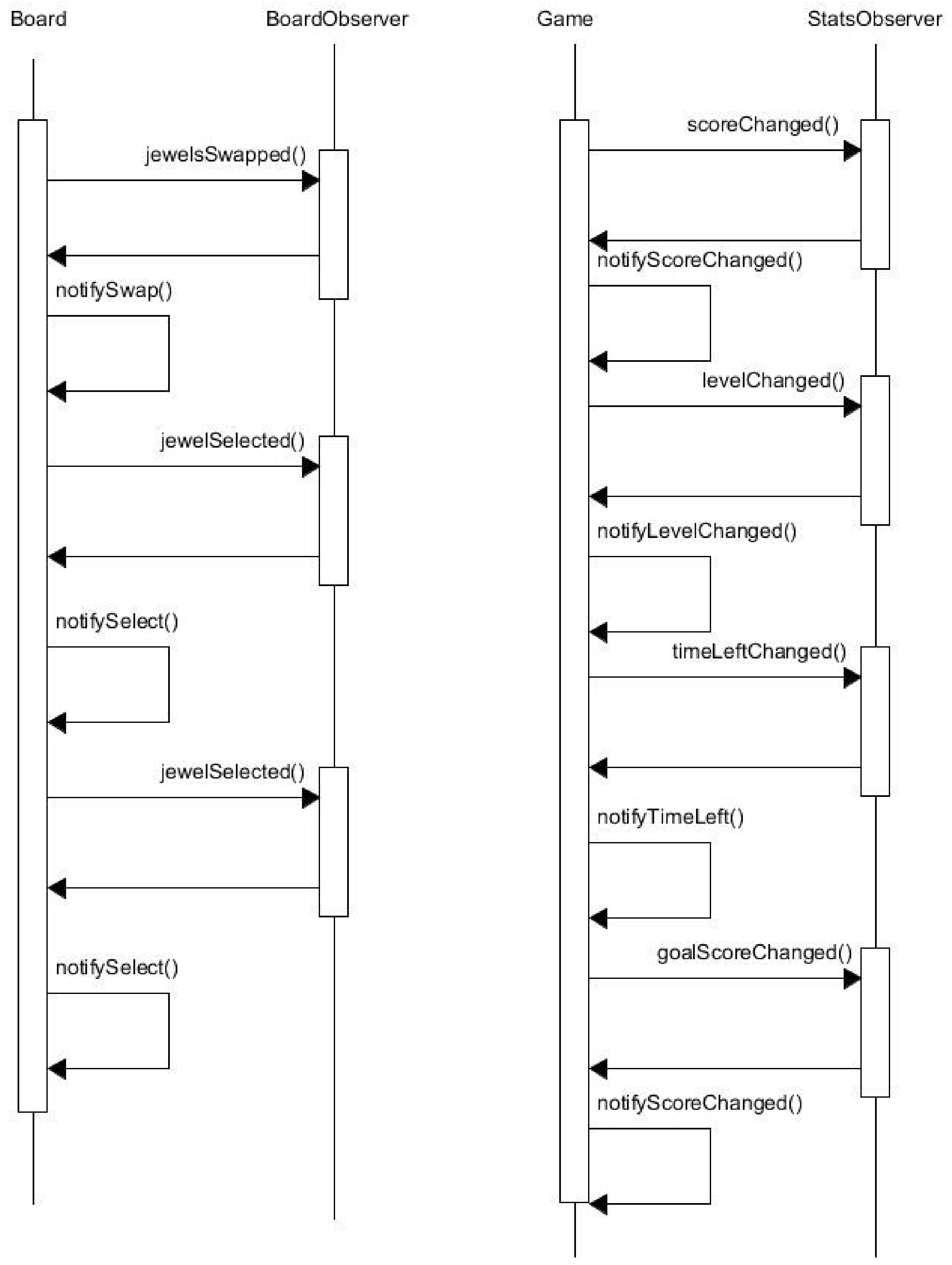
Observer Design Pattern Class Diagram:


«Interface»
BoardObservable
+notifySwap(c1 : Coordinate, c2 : Coordinate): void
+notifySelect(c1 : Coordinate, c2 : Coordinate): void
+notifyBoardChanged(): void
observer
«Interface»
BoardObserver
+jewelSwapped(c1 : Coordinate, c2 : Coordinate): void
+jewelSelected(jewel : Coordinate, old : Coordinate): void
+boardChanged(): void
Board
...
-boardObservers : List<boardObservers>
...
...
+addBoardObserver(observer : BoardObserver): void
+getBoardObservers(): List<BoardObserver>
+notifySwap(c1 : Coordinate, c2 : Coordinate): void
+notifySelect(c1 : Coordinate, c2 : Coordinate): void
+notifyBoardChanged(): void
...
subject
GUI
...
-game : Game
-gui: Gui
...
...
+jewelSwapped(c1 : Coordinate, c2 : Coordinate): void
+jewelSelected(jewel : Coordinate, old : Coordinate): void
+boardChanged(): void
...


«Interface»
StatsObservable
+addStatsObserver(observer : StatsObserver)
+getStatsObserver(): List<StatsObserver>
+notifyScoreChange(): void
+notifyGoalScoreChange(): void
+notifyLevelChange(): void
+notifyTimeLeft(): void
observer
«Interface»
StatsObserver
+scoreChanged(score : int): void
+levelChanged(level : int): void
+timeLeftChanged(time : int): void
+goalScoreChanged(goalScore : int): void
Game
...
-statsListeners: List<StatsObserver>
...
...
+addStatsObserver(listener: StatsObserver): vo
+getStatsListeners(): List<StatsListener>
+notifyScoreChanged(): void
+notifyLevelChanged(): void
+notifyNextLevelChanged(): void
+notifyTimeLeft(): void
...
subject
StatsPanel
-serialVersionUID : long
-scoreLabel : JLabel
-levelLabel : JLabel
-timeLabel : JLabel
-nextScoreLabel : JLabel
+StatsPanel(level : int, score : int, time : int, goalscore : int)
+getScoreLabel(): JLabel
+getLevelLabel(): JLabel
+scoreChanged(score : int): void
+goalScoreChanged(goalScore : int): void
+levelChanged(level : int): void
+timeLeftChanged(time : int): void

Exercise 2.3 (2)

Observer Design Pattern Sequence Diagram:



Exercise 3

Exercise 3.1

Good practice projects were recognized by having particular success factors. Below are the four strongly significant success factors for software projects:

1) Steady Heartbeat;
2) Fixed, experienced Team;
3) Agile (Scrum);
4) Release-based(one application).

Bad practice projects were recognized by having particular failure factors. Below are the four strongly significant failure factors for software projects:

1) Rules & Regulations driven;
2) Dependencies with other systems;
3) Technology driven;
4) Once-only project.

The order of the good practice and bad practice factors are based on the probability to end up as good or bad practice. We can see for example that having an agile or Scrum way of working seems to be good practice, however in their projects portfolio there were also projects that had those good practice factors without actually using an agile development. The four major success factors are also easy to implement. The article also mentions about an interesting relation between the 4th significant factor of the good and bad practice. They assume that release-based working creates a high probability to end up as a good practice, whilst going for a once-only project and having to do things for the first time has a high probability to end up as a bad practice project.

Exercise 3.2

The article mentions that they were not able to identify the exact cause that Visual Basic ended up in the good practice group. They assume that the Visual Basic projects were on average less complex than the others, thus having a bigger probability to end up as good practice. Besides, only 6 projects used Visual Basic and of which 5 scored as a good practice. This indicates that using Visual Basic does guarantee a project to end up in a good practice group. According to the analysis of the paper's authors, it was a mere coincidence and not so much an interesting finding of the study.

Exercise 3.3

I. Extensive and correct usage of widely recognized software development tools.

One could investigate if the extensive usage of many different software development tools is a good practice. I would expect this to turn out to be a Good Practice, whilst not sticking to this factor to obviously be a bad practice, for reasons I will name as follows.
Setting up tools at the beginning of a project and getting them all to work together with server integration can be quite a hassle, however I expect the initial time investment will most likely be worth it and pay off in the long run. Especially when a project team has done so multiple times, the initial required setup time will be negligible, both for small projects and for large projects. Furthermore software development tools can greatly improve the efficiency of software development, such as providing aid with early detection of bugs, organized and complete documentation, advice on how to improve a system's architecture, saving time by taking repetitive work off an employee's hands, provide a clear overview of the current status of the project and all steps taken to get to a certain point etc.

II. Team composition based on employee background check.

I would expect this to be a Good Practice (if perhaps morally questionable, but necessary in some cases). Before setting up a team, there should be a thorough investigation into each employee. It should be determined together with which people each employee would work best with, thus ensuring an effective team composition. Especially for projects where human lives are on the line (such as creating software for medical purposes) and larger projects, where a team will have to work together for extended periods of time, you will want to have reliable members from the start. Besides team composition, impeding factors for each individual employee should be taken into consideration. These factors could include a person's mental state (mental stability), physical state (healthy workers are generally more efficient), resistance to stress, stance towards superstitious beliefs (certain degrees of religious conditioning could reduce efficiency), passion for the subject of the project in question etc. One could apply each aspect of a background check with different degrees of necessity, sometimes perhaps leaving out a factor entirely, depending on the importance of the project.

III. Development of certain projects during a certain time of year.

Perhaps as a more esoteric variant of the “Heartbeat” factor, this factor would take into account the cycles in human’s lives.
During winter, the days are shorter and this has a certain impact on humans. The perception of (especially blue) light activates the brain, whilst darkness tends to lessen its activity. In other words, a project team may find it more in accordance with their biorhythm to do short, small projects during the winter, whilst working on longer, larger projects during summer.
Personally I think, if we only take this factor into perspective, it would be a Good Practice, because it synchronizes temporal human conditions with their work conditions. However, taking into account the fact that a company may lose customers if it say’s it won’t start on a certain project until it is either summer or winter, it could be disadvantageous to implement initially. Then again, if a certain company has gained a reputation of producing quality products by taking the currently discussed factor into account, it may be of positive influence. And perhaps this method will develop into a trend and become a signature method of said company increasing its fame. It is a matter of initial investment.
Thus I am of the opinion that this will be a Good Practice.

Exercise 3.4

I. Once-only

When a project team gets assigned a “Once-only project”, it generally already means that the team has little to no experience with said project, since they work on it only once (unless one or more team members has previous experience with said project, however given the vast amount of different project types, let us neglect this detail). This lack of experience will lead to the project costing more resources and requiring more time to finish. Usually, whilst a project was finished behind schedule and with higher costs, at least the team has gained experience with said project. However, “Once-only” means that this project team is not planned to be tasked with the same type of project again. This means the gained experience has far less value than when the team *would* be tasked with such a project in the future. Assuming this project team would use Agile Development, one of the most valuable advantages of this method is the feedback. However, like with experience, the positive effects of feedback would also be vastly reduced.

II. Rules & Regulations driven

With terms like “agile” and “flexible” flying around nowadays and experts pointing out the ability to change as one of the most important factors for software development, the term “Rules & Regulations” will immediately raise questions among those who have been following recent developments. When one initiates a project where both the project team and the final product are bound by a set of rules, the ability to adapt to change is promptly reduced, if one does not consider breaking the rules. However, if the rules are there to be broken once change is needed, why have rules in the first place? Nowadays virtually every software development team is bound to face changing requirements, a changing final product, changing conditions etc. While the concept of Rules & Regulations is meant to bring stability and organization, perhaps it would be better not to stick to them as if they represent all overriding directives, but rather as handy guidelines that may or may not be chosen to be followed, depending on the situation.

III. Dependencies with other systems

There are several problems a project team will imminently face upon starting a new project if the fulfillment of the product’s ultimate function depends on the workings of other systems not part of the project itself. For starters, the more systems the project depends on, the broader the range of knowledge required by the project team, because this dependence will no doubt play a role in the product’s implementation. More systems also means more sources for bugs and conflicts to arise. Next, the project team will have to trust that the other systems are reliable and well designed, since the implementation may have to be based on this design. Should these other systems be so complex, communication and cooperation with experts would be required, which would mean a higher cost and more time needed (also trusting that these experts are reliable). The above mentioned problems and the dependency on other systems furthermore limit the freedom of the project team, because it will have to adapt its methods to suit cooperation with the other systems.
All in all, one would want to avoid these dependencies. Unless the members of the project group created these systems themselves and fully understand them and the systems were verified to work flawlessly, I see no reason why one would proceed with such a project. It may then be better to consider creating a standalone product.