

Project Report Team 3

Koushik Kannepally s000000

Oebele Lijzenga s1954385

Tommy Lin s1840932

Jordy van der Poel s2692937

May 12, 2022

Skills to be reviewed: S1, S2, S3, S4, S6

Contents

1	Project documentation	2
2	Agile team work in Jira	4
3	Continuous integration pipeline	5
4	Testing	6
5	Design patterns	7
6	Maintainability	8
7	Requirements change	9
8	Refactoring	10
9	Debugging	11
10	Functioning of the product	12

1 Project documentation

For this project, all the separate game objects will be made in separate classes. These are the Patch, QuiltBoard and the TimeBoard classes. To implement the time board, the individual spaces are implemented in the SpaceElement class. A SpaceElement has a type, indicating whether it is a normal or a special space, and a Player, to determine where each player is on the TimeBoard.

The Patch class is the implementation for a single patch. A patch has spaces, which is a list containing lists of booleans. This way, the patch can be made as a matrix of booleans, to indicate whether that space is used for the patch. Think of for example `[[true,true,true]]` to implement a patch that is simply 3 horizontal connected spaces. The `nrButtons` and `nrTimeTokens` are the costs for buying this patch, as indicated by the game rules. The `buttonScore` is the amount of buttons this patch earns when getting income. The Patch class also has methods to rotate and flip the patch.

To create all the specific patches for the game, a factory class named Patch-Factory is used. These created patches are used in the PatchList class, which also has a `neutralTokenPosition` to keep track of which patches can be bought during the game.

Similar to the Patch, a QuiltBoard has Spaces, which is a list containing lists of booleans, to implement the 9x9 matrix on which you can put patches. Furthermore, a list of patches that are on the quiltboard are used, as the `buttonScore` attribute of all patches is used for paying income to the player. For methods, this class has `patchToCoords` and `canPlace` as helper methods to translate the patch to coordinates that can be used to determine where a patch can be placed. A patch can then be placed with the `placePatch` method. `hasSevenBySeven` and `countEmpty` are used to implement the game rules, i.e. the first person with a full 7x7 square gets additional points, and any empty spaces at the end of the game result in a penalty.

The Player class is the implementation of a single player within the game, who has their own name, quiltBoard and their number of buttons. Methods for increasing or decreasing the button count are implemented, with `addPatchButtons` being the method for paying button income. `placePatch` is also implemented here, as when placing a patch you have to pay a cost, and then it calls the method in QuiltBoard.

All of these classes are then used in the Game class, which is the controller of the overall game. For showcasing the current game, a TextUI class is used, implementing the observer pattern, as these should always be updated to keep in sync with each other.

To actually interact with the game, the GameInteraction class will be used, following the Strategy pattern, as multiple ways of interacting with the game should be possible, as the game should be playable on a single computer. Think of for example, interacting with the game with WASD or the arrow keys.

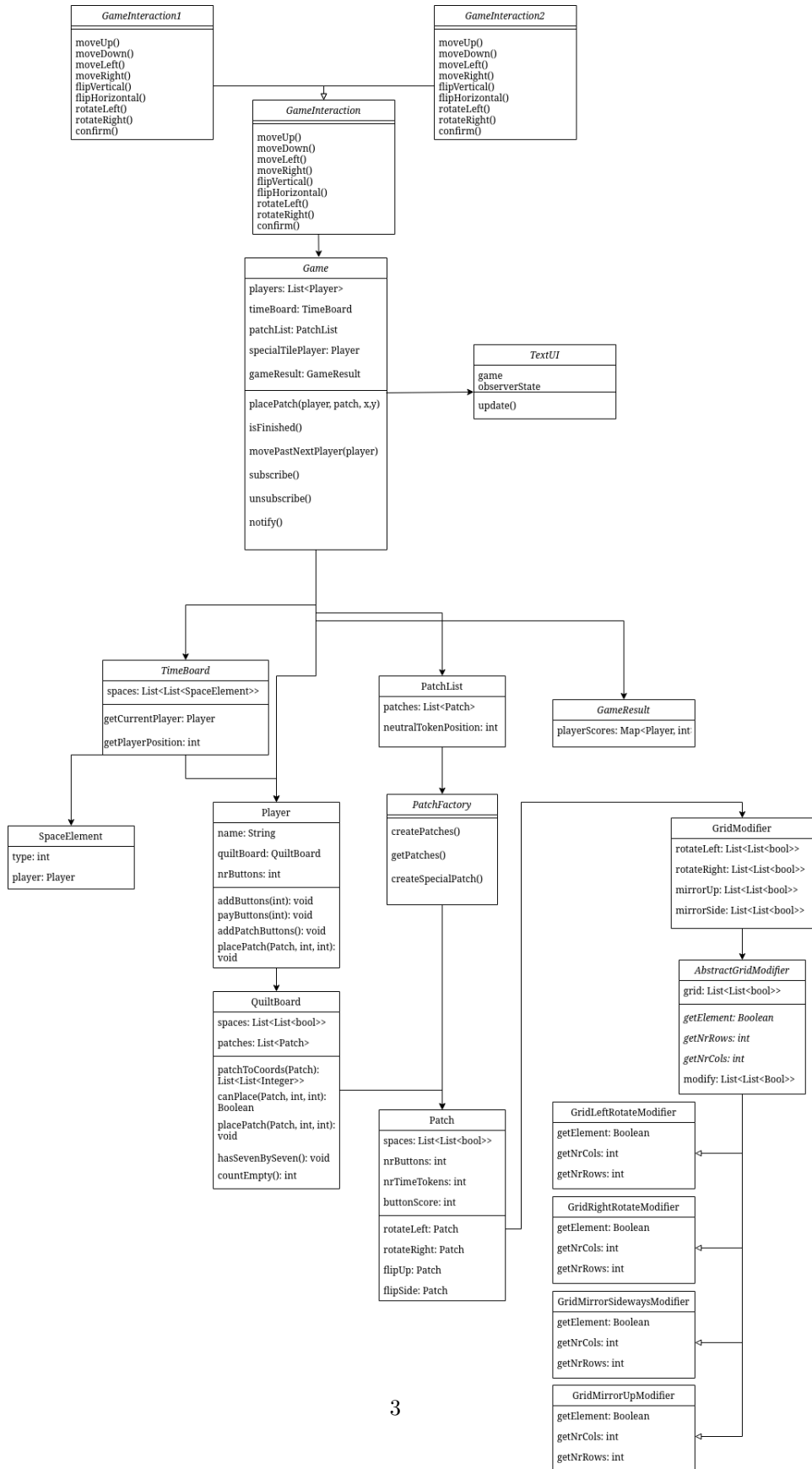


Figure 1: Current class diagram during sprint 1

2 Agile team work in Jira

In the Jira environment, individual issues for each task were made. For the first sprint, 4 main issues were made regarding the implementation of the game, each estimated to be roughly 3 hours of work, so the work was divided roughly equally. This was discussed and divided during the practical of week 2, with the deadline for the sprint being the practical of week 3. Further tasks regarding setting up the GitLab CI and writing the report were divided among people who had the time to do so, as other similar tasks can be divided in later sprints to the other people. The sprint can be seen in figure 2.

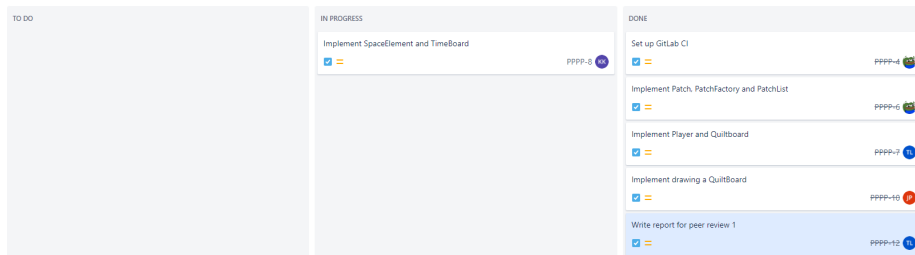


Figure 2: Jira board during sprint 1

3 Continuous integration pipeline

The CI currently implemented consists of the build phase and the test phase. For the build phase, maven is used to build a jar by using "script:- mvn jar:jar", with the tags "pppp", as mentioned in the announcement on canvas. For the tests, a similar implementation is done by using maven, i.e. "script: - mvn test".

4 Testing

As agile follows test-driven development, tests were written beforehand actual code was implemented. This test would involve what the class would try to implement, and should the test pass, the class is sufficient in this case. If it fails, it means that the class should be changed. Due to this, all tests should provide sufficient coverage, as code was specifically written for this test. It also means that more tests should be unnecessary, as minimal code is written to pass the test. The test results and coverage can be found in figure 3. From this it can be seen that line coverage is 94% and branch coverage 87%, which is sufficient, but it could be improved by looking what lines and branches are missing from the tests.

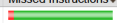
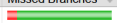






Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Ctxy	Missed Lines	Missed Methods	Missed Classes
com.patchwork.app.backend		94%		90%	20 72	27 264	14 35	3 10
com.patchwork.app		0%		n/a	2 2	6 6	2 2	1 1
com.patchwork.app.frontend		92%		75%	6 17	2 28	1 5	0 1
com.patchwork.app.utils		100%		100%	0 28	0 50	0 26	0 7
Total	181 of 3,320	94%	13 of 102	87%	28 119	35 348	17 68	4 19

Figure 3: Test coverage during sprint 1

5 Design patterns

The current design implements 3 patterns as discussed in the lectures. These are the Factory Method Pattern for the PatchFactory and Patch, the Observer pattern for the Game and TextUI, and the Strategy pattern for the GameInteraction and its subclasses.

For the factory pattern, the PatchFactory corresponds to the ConcreteCreator, and the Patch to the ConcreteProduct. Abstraction of these classes should not be necessary in this project, because this one factory only produces a single class of product, neither of which need abstraction as these are its only uses in the project.

The strategy pattern is implented such that GameInteraction corresponds to Strategy, and the classes GameInteraction1 and GameInteraction2 correspond to the different ConcreteStrategy classes as seen in the lectures.

Lastly, the Game class is in this case the ConcreteSubject, and the TextUI is the ConcreteObserver. Similar to the implementation for the factory method, abstraction was left out of this project, as in this project there is only a single subject and a single observer, so abstraction would be redundant. This pattern was chosen, because it allows for consistency between the Game and the TextUI, as TextUI should always be able to show the correct state of the game to the players.

6 Maintainability

7 Requirements change

8 Refactoring

9 Debugging

10 Functioning of the product