

# Project Report Team 3

Koushik Kannepally s2925877

Oebele Lijzenga s1954385

Tommy Lin s1840932

Jordy van der Poel s2692937

June 2, 2022

Skills to be reviewed: S1, S3, S4, S6, S8

## Contents

<b>1</b>	<b>Project documentation</b>	<b>2</b>
<b>2</b>	<b>Agile team work in Jira</b>	<b>7</b>
<b>3</b>	<b>Continuous integration pipeline</b>	<b>8</b>
<b>4</b>	<b>Testing</b>	<b>10</b>
<b>5</b>	<b>Requirements change</b>	<b>11</b>
<b>6</b>	<b>Design patterns</b>	<b>12</b>
<b>7</b>	<b>Maintainability</b>	<b>13</b>
<b>8</b>	<b>Refactoring</b>	<b>14</b>
<b>9</b>	<b>Debugging</b>	<b>15</b>
<b>10</b>	<b>Functioning of the product</b>	<b>16</b>

# 1 Project documentation

For this project, three packages are defined; the game package, which includes classes related to the game rules. Secondly, the controller package, which contains classes for interacting and controlling the game states. Lastly, the UI package, to show the game.

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. 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 PatchFactory 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.

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. Methods in this class are used to place patches, or to determine the score of the quiltboard.

The Player class is the implementation of a single player within the game, who has their own name, QuiltBoard and their number of buttons. The methods for this class are used to change the button count, or to pay for a patch.

All of these classes are then used in the Game class, which implements the current game flow and rules.

To actually control the game states, the controller package is used. This package contains classes to determine the current game state, which are simply the different states a player can be in. An example of a state is the `PickPatchGameState`, in which a player can choose one of the three available patches to buy. Furthermore, the `GameInput` section is used for the inputs decided by the player. An example would be a player using the arrow keys on the keyboard to interact with the game, which then notifies the controller to change the game.

These two parts are used in the GameController, which keeps track of the game, game states and inputs. This class actually determines the flow of the current game, utilises player inputs and updates the game for the UI to draw.

To showcase the current game, the UI package consisting of the TUI is used. This class can draw the game based on the current game and game state from the GameController.

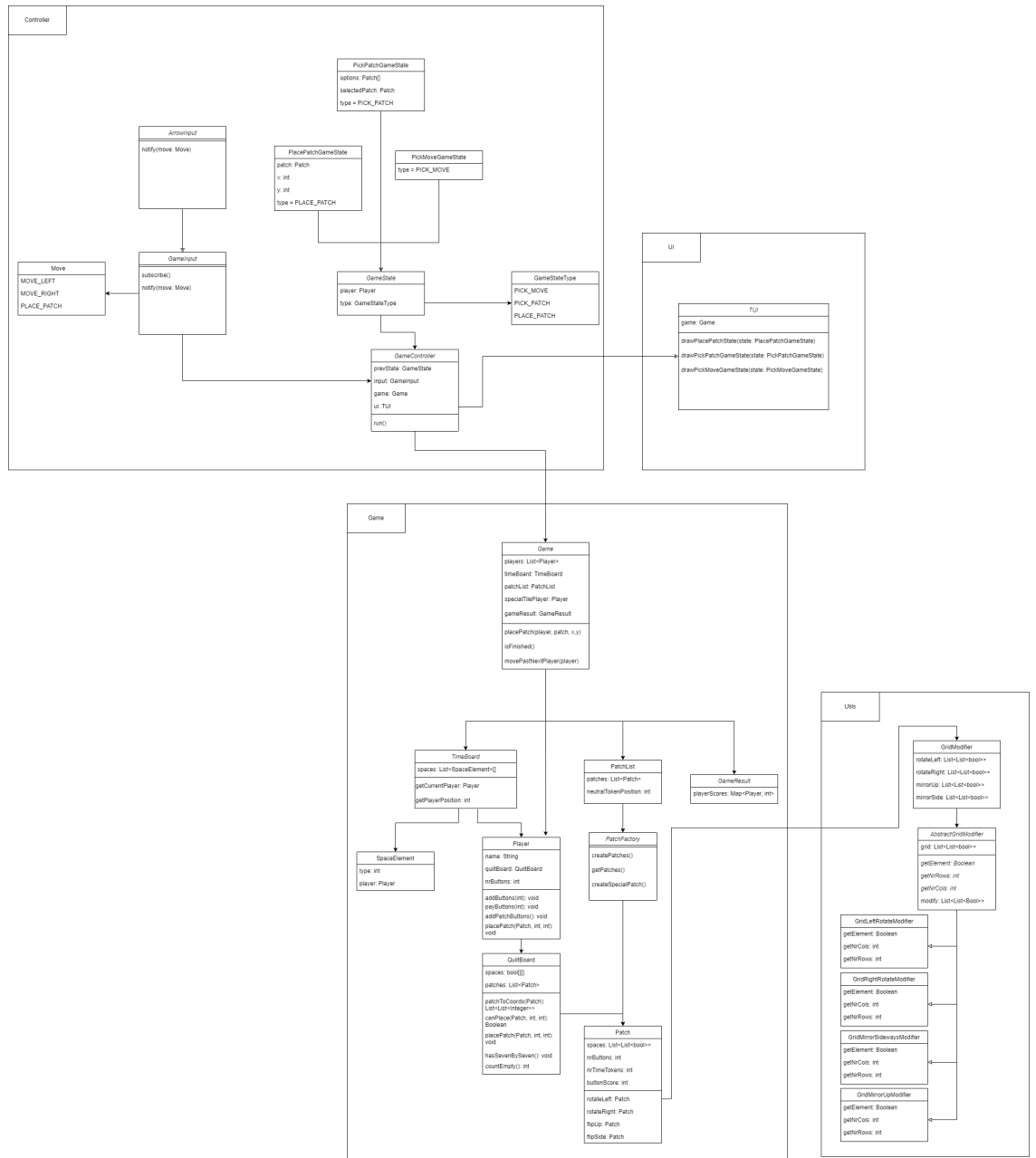


Figure 1: Complete class diagram

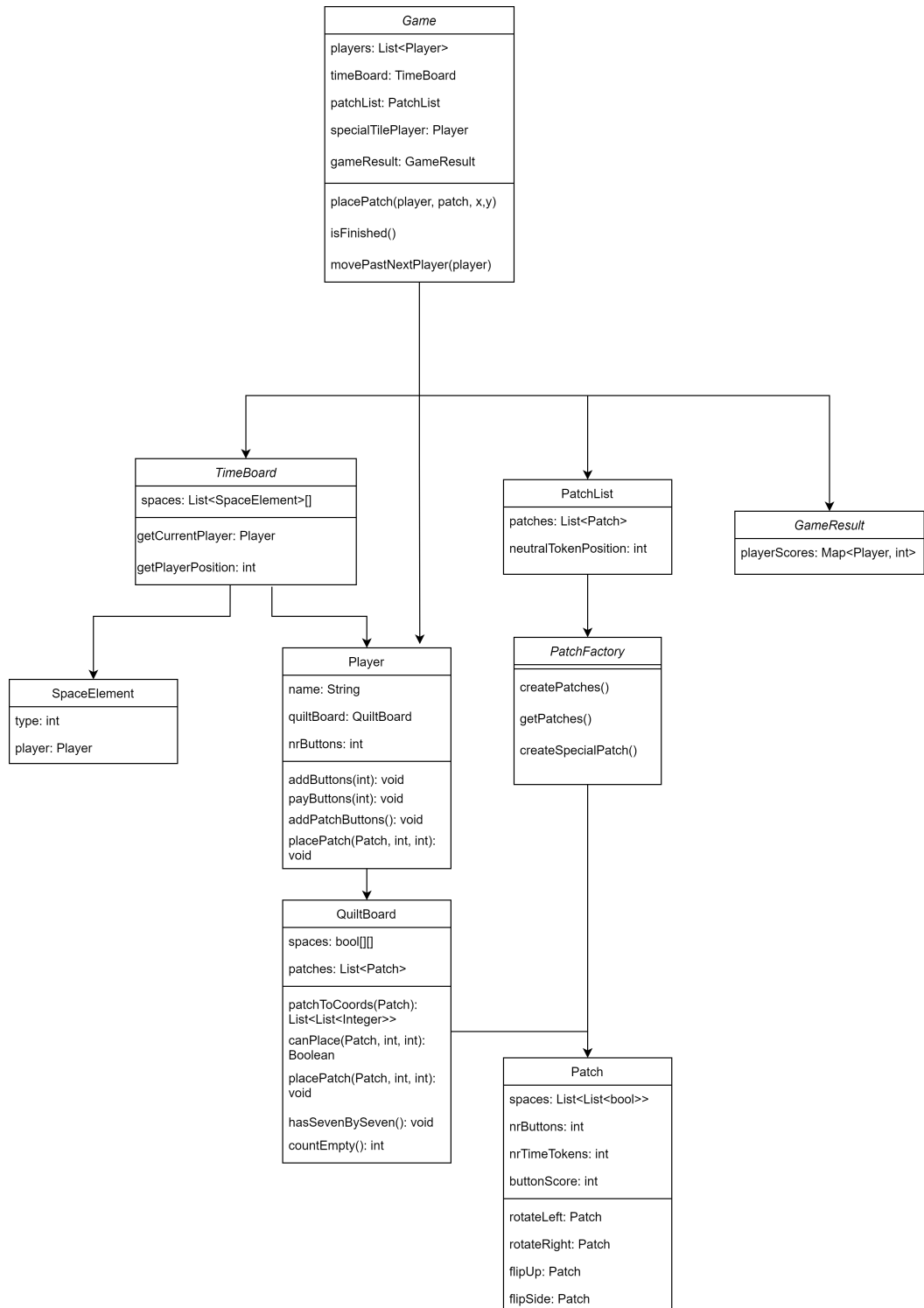


Figure 2: Class diagram for Game component

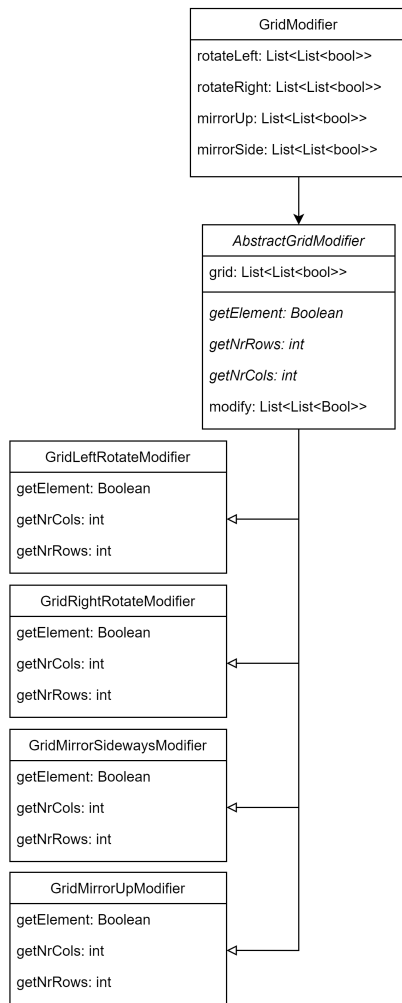


Figure 3: Class diagram for Utils component

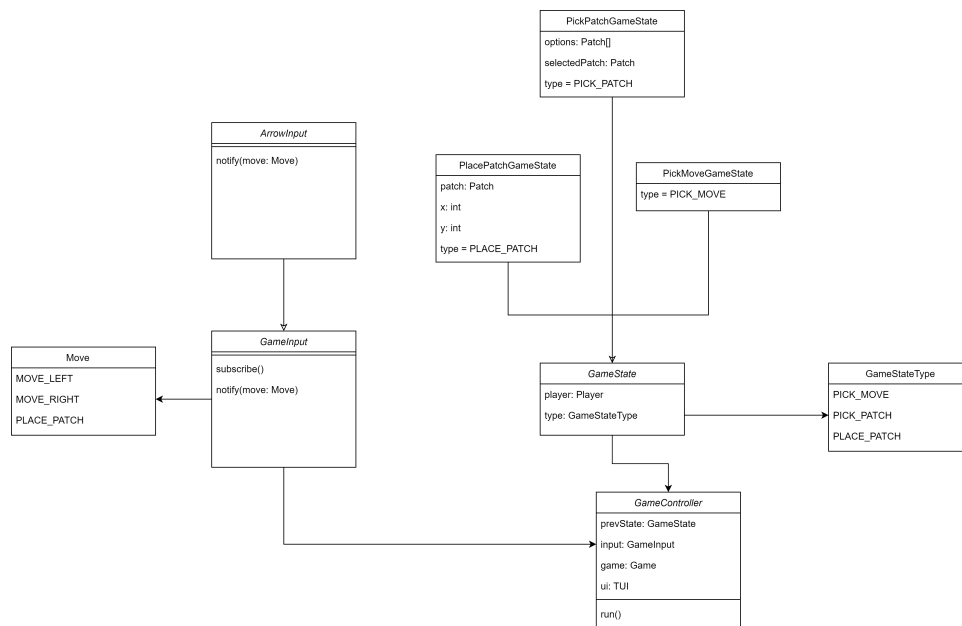


Figure 4: Class diagram for Controller component

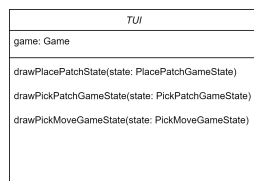


Figure 5: Class diagram for UI component

## 2 Agile team work in Jira

In the Jira environment, issues for each main task were made. For the first sprint, 4 main issues were made regarding the implementation of the game, so they could be divided per member. These issues were divided into working on the Patch functionality, the QuiltBoard and Player classes, the TimeBoard or on the TUI. All of these issues were estimated to be roughly 3 hours of work, so the work was divided roughly equally. Additional tasks regarding the structure of the project, for example, 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 6.

For sprint 2, again four main issues were determined so that it could be split divided over the group. This sprint would focus on implementing the Game and GameResult, which implements the actual game rules, GameController, which handles the overall game flow, i.e. taking turns in the game, the GameInteraction, which listens to user input and uses this to determine what interaction the GameController should take, and again, work on the TUI. Other smaller tasks, such as implementing the feedback given by the twin team, or writing the report for next sprint, were done together during the tutorial sessions. The sprint can be seen in 7.

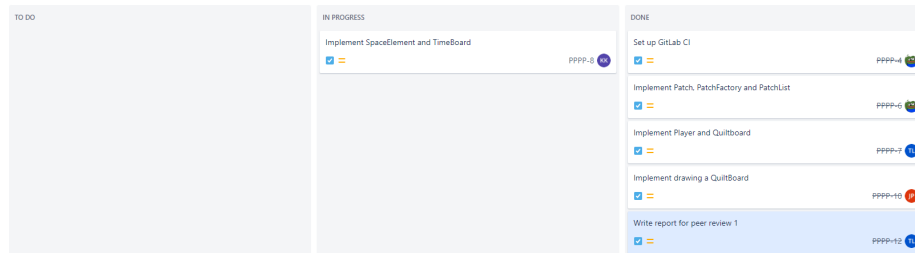


Figure 6: Jira board during sprint 1

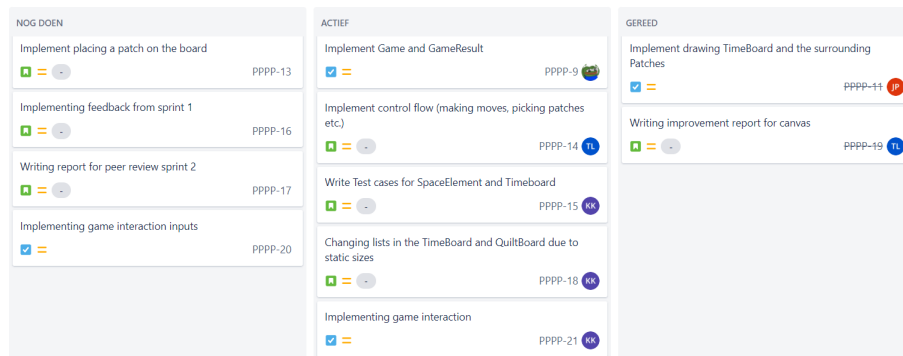


Figure 7: Jira board during sprint 2

### 3 Continuous integration pipeline

```
build:
  image: maven:3.8.5-openjdk-18
  stage: build
  tags:
    - pppp
  script:
    - mvn jar:jar
  artifacts:
    when: always
    paths:
      - target/*.jar

test:
  image: maven:3.8.5-openjdk-18
  stage: test
  tags:
    - pppp
  script:
    - mvn test
  artifacts:
    when: always
    reports:
      junit:
        - target/surefire-reports/TEST-*.xml
```

Figure 8: The Gitlab CI file

Gitlab CI is used to automatically build and test the application. The build and test stage both use the same Docker image with a fixed maven and openJDK version. This way we ensure that the application will always work with these compiler and tools function. The test stage only runs after the build stage as passed, thus preventing a double failure for a single problem.

**Build Stage** The build stage uses maven to compile the application into a single JAR. The resulting JAR file is declared as an artifact of the CI job so that it is available for download after the job is completed. There are several advantages to having a build stage in a CI pipeline. First of all, it ensures that the application can be compiled on a clean machine. Secondly, it is very easy to detect when a developer has accidentally pushed code that does not compile under the standard tooling for the project. Finally, it makes the resulting application easily available allowing for easier use and demonstration of the application.

**Test Stage** In the test stage maven is used to build and test the application. JUnit reports in the form of Surefire reports are declared as stage artifacts. The obvious advantage for this build stage is that it is easy to detect when a developer has pushed code that does not pass all unit tests under the standard tooling in the project. This is quite useful as it is easy to forget running all



unit tests before pushing code. Furthermore, for larger codebases it is common that the test suite is so large that it is not convenient for the developer to run all tests on their own device, this stage allows the developer to offload this to a Gitlab runner. Finally, JUnit reports are added as artifacts because it allows tools like Gitlab to format and present the results of the pipeline in their own manner allowing for easier analysis of test results as Gitlab CI stage output is not always nicely readable.

## 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 can be found in figures 9 - 11. Overall, most classes which are tested have good coverage (higher than 80%), except for Patch. This could be improved in a future sprint. Furthermore, several classes, such as Move, SpaceElement or the classes in the Utils package, do not have any branch coverage, this is due to these classes being utility classes to help support a larger class. This is not due to a testing issue.

### com.patchwork.app.backend





















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
GameController		88%		81%	10	40	21	153	0	9	0	1
Patch		42%		78%	7	13	8	19	4	6	0	1
Game		94%		80%	7	29	7	69	0	8	0	1
PatchFactory		99%		100%	1	5	2	116	1	4	0	1
TimeBoard		100%		100%	0	12	0	51	0	4	0	1
QuitBoard		100%		92%	4	34	0	59	0	6	0	1
PatchList		100%		100%	0	6	0	22	0	3	0	1
Player		100%		100%	0	8	0	18	0	5	0	1
Move		100%	n/a	n/a	0	1	0	7	0	1	0	1
TimeBoard.SpaceElementType		100%	n/a	n/a	0	1	0	4	0	1	0	1
TimeBoard.SpaceElement		100%	n/a	n/a	0	2	0	5	0	2	0	1
GameResult		100%	n/a	n/a	0	1	0	4	0	1	0	1
Total	176 of 4,146	95%	26 of 203	87%	29	152	38	527	5	50	0	12

Figure 9: Test coverage of the backend

### com.patchwork.app.frontend



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
TUI		99%		100%	1	43	1	104	1	13	0	1
Total	1 of 565	99%	0 of 60	100%	1	43	1	104	1	13	0	1

Figure 10: Test coverage of the backend

### com.patchwork.app.utils

Element	Missed Instructions	Cov	Missed Branches	Cov	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes	
<a href="#">ConsoleColor</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	3	0	9	0	3	0	1
<a href="#">GridModifier.AbstractGridModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	0	4	0	12	0	2	0	1	
<a href="#">GridModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	5	0	5	0	5	0	1
<a href="#">GridModifier.GridMirrorUpModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	4	0	6	0	4	0	1
<a href="#">GridModifier.GridRightRotateModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	4	0	6	0	4	0	1
<a href="#">GridModifier.GridMirrorSidewaysModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	4	0	6	0	4	0	1
<a href="#">GridModifier.GridLeftRotateModifier</a>	<div><div></div></div>	100%	<div><div></div></div>	100%	n/a	0	4	0	6	0	4	0	1
Total	0 of 266	100%	0 of 4	100%	0	28	0	50	0	26	0	7	

Figure 11: Test coverage of the backend

## 5 Requirements change

## 6 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.

## 7 Maintainability

## 8 Refactoring

Refactoring is used to improve the structure or quality of code. In our case, a lack of flexibility in instantiating `Game`, `GameController` and its dependencies was found. Basically, the constructor of `Game`, `GameController` and `PatchList` would instantiate the objects that they depend on in their constructor. This left little room for injecting modified versions of classes (i.e. mock classes) for testing. See an example below:

```
public Game() {
    this.players = new ArrayList<>();
    this.players.add(new Player("Player 1"));
    this.players.add(new Player("Player 2"));
    this.timeBoard = new TimeBoard(this.players);
    this.patchList = new PatchList();
    this.specialTilePlayer = null;
    this.result = null;
}
```

The instantiation of dependencies was offloaded to a new `GameFactory` class which creates players, a timeboard and a patchlist. In the new situation, `Game` now accepts these dependencies as arguments in its constructor and `GameFactory` is responsible for providing these. Similar refactoring was applied to `GameController` and `PatchList`.

Commits:

- 46926da71fb697162c042c8b5acee10b8f020de7
- 1bef18b35b44d79c6751fc9c9770bac4347a3cff
- f97a7bc7914258b8701c4d360339bdb07170ee15

Another refactoring that was applied was pulling generating system testing functionality into a separate `GameTest` abstract class. This class contains `setUp` and `tearDown` methods which create, start and stop a game during the test lifecycle. By extracting this functionality into a separate class instead of having separate implementations in each test class that needs it means that it is now easier to write tests which need an entire running game. Furthermore, the `Game` variant with mock classes injected into it is now only created in one single place, namely `GameTest`, which makes the code more maintainable.

Commit: 3f4485b926bb3e95a888e3248e818286e72701db

Finally, `Player` used to have the method `addPatchButtons` which would sum up the button reward based on patches of the players quiltboard. This method was moved to quiltboard and now only returns the number of buttons to award instead of adding them to the player immediately. This refactoring makes for better separation of concerns.

Commit: dbe8caeceb2de549522776e3e74817e6890fa9fb

## 9 Debugging

## 10    Functioning of the product