

Project Report Team 3

Koushik Kannepally s2925877

Oebele Lijzenga s1954385

Tommy Lin s1840932

Jordy van der Poel s2692937

June 21, 2022

Contents

1	Project documentation	2
2	Agile team work in Jira	7
3	Continuous integration pipeline	8
4	Testing	10
5	Requirements change	12
6	Design patterns	14
7	Maintainability	15
8	Refactoring	18
9	Debugging	19
10	Functioning of the product	20
11	Appendices	21
A	Spotbugs Security and Static Analysis	21
B	Individual Coverage Reports	23
C	Jira board images	23

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.

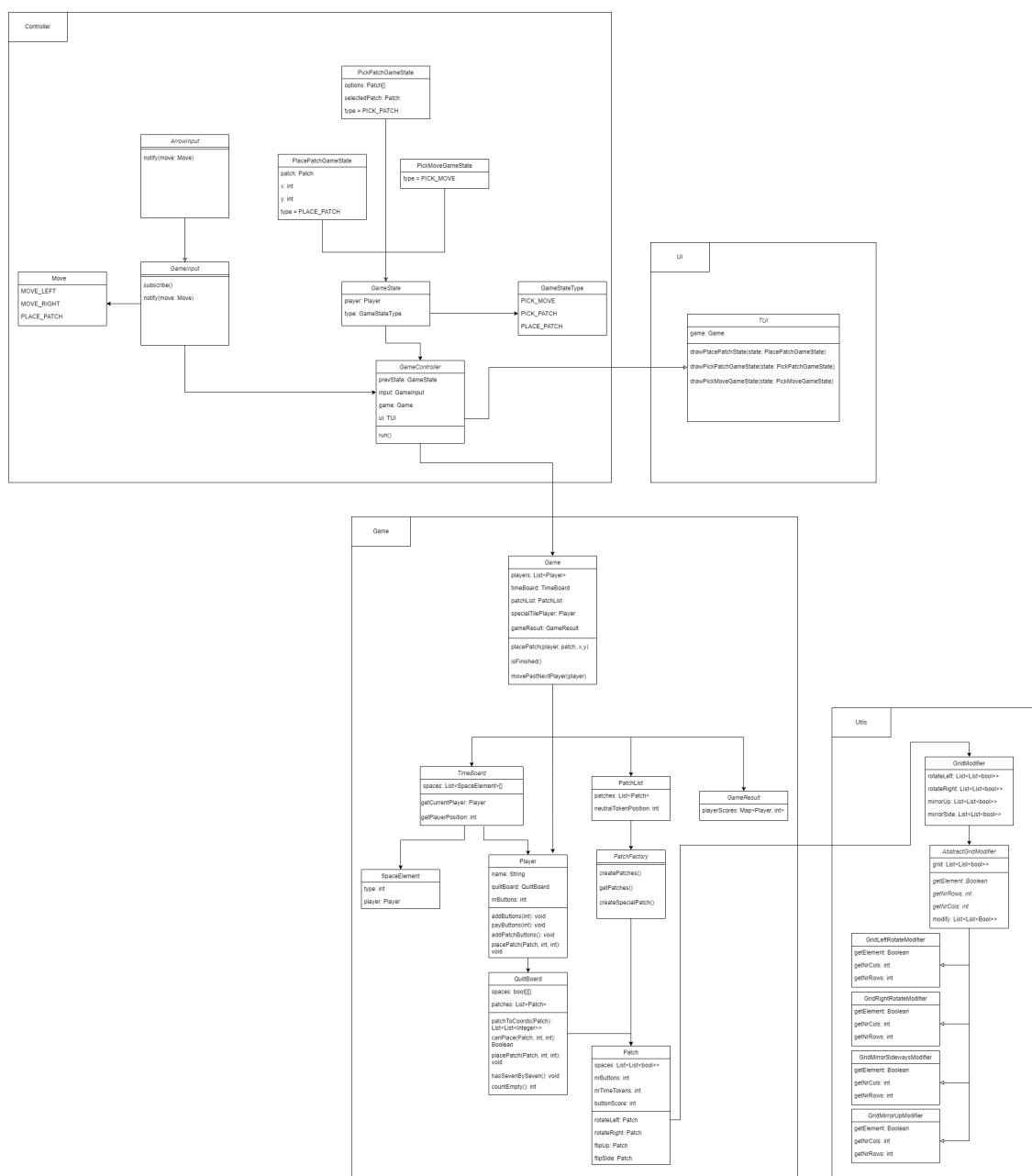
The Player class is the implementation of a single player within the game, who has their own name, QuiltBoard and their number of buttons. The methods for this class are used to change the button count, or to pay for a patch.

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.



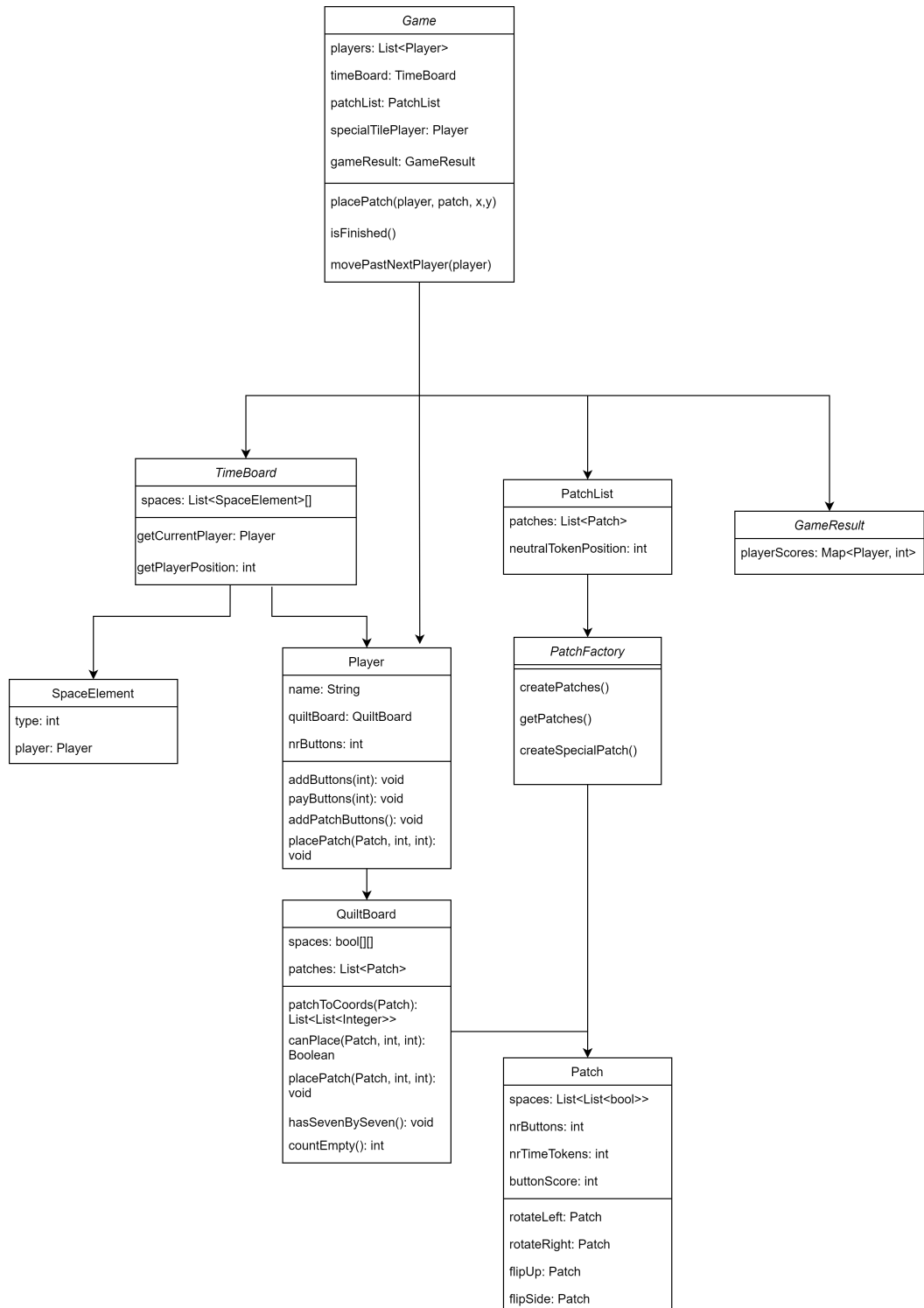


Figure 2: Class diagram for Game component

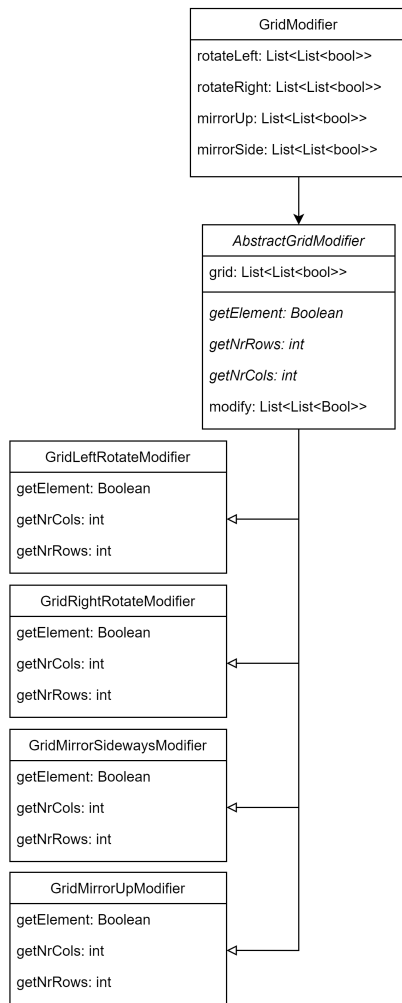


Figure 3: Class diagram for Utils component

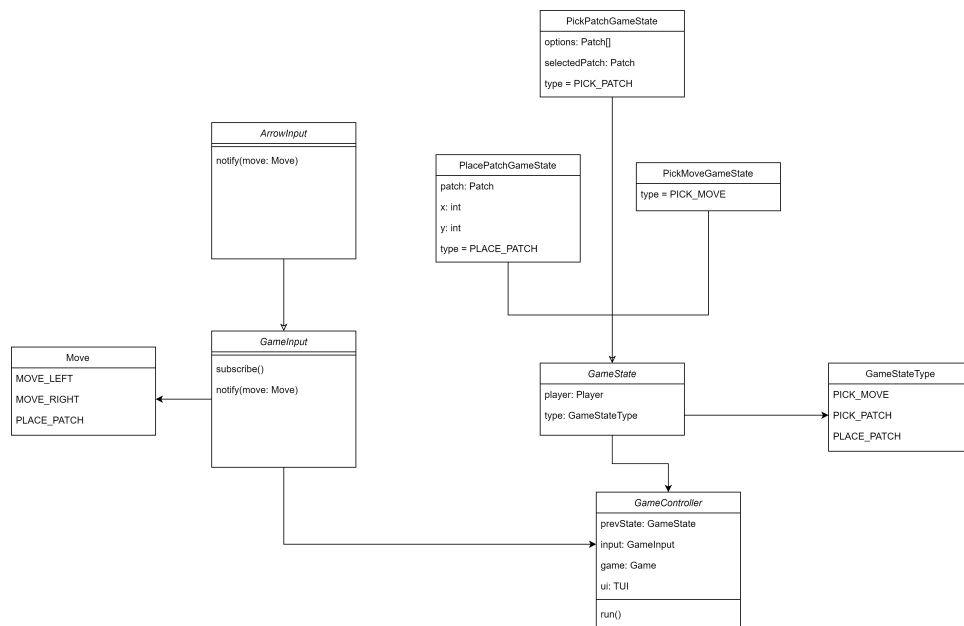


Figure 4: Class diagram for Controller component

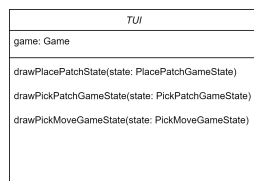


Figure 5: Class diagram for UI component

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 class, the QuiltBoard and Player classes, the TimeBoard class or on the TUI. All of these issues were estimated to take roughly the same amount of time, so the work was divided equally. Additional tasks regarding the structure of the project, for example, setting up the GitLab CI, were divided among people who had the time to do so. The board during sprint 1 can be seen in Figure 26 located in Appendix C.

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, which implements the actual game rules, GameController, which handles the overall game control, 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 Jira board during this sprint can be seen in Figure 27 in Appendix C.

For the final sprint, most individual work was done, and it was mostly improving documentation. As such, there were no main individual tasks; instead, larger overarching stories were used, which everyone contributed to. This can be seen in Figure 28 in Appendix C.

3 Continuous integration pipeline

```
stages:
  - build
  - test
  - analyze

sast:
  variables:
    SAST_EXCLUDED_ANALYZERS: bandit, brakeman, eslint,
      flawfinder, gosec, kubesecc, nodejs-scan, phpcs-security-audit,
      pmd-apex, security-code-scan, semgrep, sobelow
  stage: test
include:
  - template: Security/SAST.gitlab-ci.yml

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

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

analyze:
  image: maven:3.8.5-openjdk-18
  stage: analyze
  tags:
    - pppp
  script:
    - mvn install
    - mvn site
  artifacts:
    when: always
    paths:
      - target/site/*
```

Figure 6: The Gitlab CI file

Gitlab CI is used to automatically build, test and analyze the application. All stages but one 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.

SAST

The GitLab SAST (Static Application Security Testing) feature is enabled which automatically checks for potential security issues using SpotBugs. SAST is run in parallel with the test stage. The results of this check is shown above the 'Merge' button in a merge request, see the first figure of appendix A for an example. The advantage of using SpotBugs for security checking is that it prevents common security errors from being implemented unnoticed.

Static Analysis

In the analysis stage, SpotBugs is used to analyze the quality of the Java code. It generates an HTML report which is exported as an artifact from the CI pipeline. Such an HTML page is shown in the second figure of appendix A. This stage also produces the test stage coverage HTML as a side-effect. The purpose of this stage is to alert the developer of code smells. The developer can also run SpotBugs locally to check for any issues before committing. Sadly, there was not enough time at our disposal to fix many of the detected code smells. However, it is nice to be made aware of them.

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 overall coverage report can be found in figure 7. The coverage for the individual packages can be found in Appendix A, Figures B - B.

Overall both line and branch coverage are above the required 80%. Furthermore, almost every individual class has over 80% line and branch coverage. Exceptions to this are the classes App, ScannerInput, and some of the adapted classes. App is the main class which starts the Patchwork game, and as such is not tested. ScannerInput has an untested method run(), inherited from Runnable, which could not be tested. However, the methods which are used during this running loop are tested, and as such the overall function has been tested. The adapted classes will be discussed in more detail in Section 5.

Furthermore, several classes, such as Move, 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, these classes simply do not have branches.

Tests would test both happy paths - A test involving a proper execution of the code, and unhappy paths, in which invalid input is also tested. During a happy path, the tested states should change accordingly, and no exceptions are expected. An example can be seen in Figure 8. For testing unhappy paths, an invalid input or situation is tested, in which both an exception are expected, and it is asserted that the tested states have not changed due to an invalid input. An example of this can be seen in Figure 9.

Finally, 10 overall system tests were written. These system tests would mainly test the function of the GameController. These were done twice, once with tests that are based on user input, and one without. The former would ensure that the inputs would result in a correct game flow. The latter would make sure that if there were any issues found, that it can be excluded that it is due to an incorrect game implementation. The non-input tests were also sent to the twin team, as implementing user input tests for another system involves many steps which would have to massively change their implementation, solely due to different ways of handling user input, which are very hard to adapt.

Element	Missed Instructions	Cov *	Missed Branches	Cov *	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	
com.patchwork.app	<div><div></div></div>	0%	<div><div></div></div>	0%	14	14	47	47	10	10	1	
com.patchwork.app.utils	<div><div></div></div>	83%	<div><div></div></div>	100%	3	28	9	50	3	26	1	
com.patchwork.app.frontend	<div><div></div></div>	87%	<div><div></div></div>	93%	17	80	37	233	14	43	2	
com.patchwork.app.backend.controller.GameController	<div><div></div></div>	95%	<div><div></div></div>	89%	11	83	10	193	2	37	0	
com.patchwork.app.backend.controller.GameStates	<div><div></div></div>	97%	<div><div></div></div>	91%	2	33	1	70	0	20	0	
com.patchwork.app.adapters	<div><div></div></div>	97%	<div><div></div></div>	100%	5	54	7	116	5	42	0	
com.patchwork.app.backend.model	<div><div></div></div>	99%	<div><div></div></div>	88%	20	153	8	438	3	73	0	
com.patchwork.app.backend.exceptions	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1	0	
com.patchwork.app.backend.controller.inputs	<div><div></div></div>	100%	<div><div></div></div>	100%	0	8	0	15	0	7	0	
Total		370 of 6,747	94%	43 of 377	88%	72	454	119	1,164	37	259	4

Figure 7: Test coverage of the project

```

@Test
public void setupTest() {
    assertEquals(3, game.patchList.getAvailablePatches().size());
    assertEquals(2, game.players.size());
    assertNull(game.specialTilePlayer);
    assertFalse(game.isFinished());
    assertTrue(game.players.contains(startingPlayer));
    assertTrue(game.players.contains(otherPlayer));
    ...
}

```

Figure 8: Example of a test involving happy path, in this case a correct start of the game.

```

@Test
public void testPlacePatchNotEnoughButtons() {
    startingPlayer.nrButtons = 0;
    Patch patch1 = game.patchList.getAvailablePatches().get(0);
    assertTrue(patch1.buttonCost > 0);
    assertThrows(
        GameException.class,
        () -> game.placePatch(startingPlayer, patch1, 5, 5));
    assertEquals(0, startingPlayer.quiltBoard.patches.size());
}

```

Figure 9: Example of a test involving unhappy path, trying to buy a patch with insufficient buttons.

5 Requirements change

For the requirements change, 10 back-end only user tests were given by the twin team. These 10 tests are listed together with their tested function in Table 1. The goal was to implement adapters such that these tests would also pass for our implementation of Patchwork.

To make those adapters, adapter patterns were used. To accomplish this, the desired interfaces required for the tests were made, and then an intermediate adapter was made, which implemented the required interface, and extended our already existing class which corresponded to the same function. An example can be seen in Figure 10, where the PurchasablePatch adapter is shown. In this case, the PurchasablePatch class implements the twin team’s IPatch interface, and extends the already existing Patch functionality. A super constructor is used to be able to construct the PurchasablePatch according to their specification. To accomplish this, an additional method was used to translate their usage of String into a List<List<Boolean>>, as our implementations for constructing a patch were different.

In a similar fashion, the other interfaces and classes were adapted, extending existing code, and implementing their interfaces, to make sure that all their tests would also pass. This result of the test coverage report can be seen in Figure 5. From this it can be seen that the adapters only adapt what was necessary for the tests, resulting in almost only a 100% coverage rate whenever relevant. The exception to this is PatchBoard, which had to use a try/catch block with an unused error, due to a super method from the already existing QuiltBoard which uses this error, which of course can not be tested in an external test which does not use this error. Furthermore, LeatherPatch has only a coverage of 58%. However, this is due to LeatherPatch implmenting IPatch, which contains methods such as rotating and flipping the patch, which are not used. As such, the lack of 100% coverage for some adapters can be justified.

Test case	Tested function(s)
GameConstructorTest	Correct setup at start of game
playerTest	Correct setup at start of game An unusual input or event
testAdvanceTimeToken	Correct player takes turn Correct execution of “Advance and Receive Buttons”
testPlaceableAndNonPlaceable	Correct execution of “Take and Place Patch”
testSetAndGetPosition	An unusual input or event Setting patch token position
testMovingToken	An unusual input or event Moving the time token
matrixManipulationTest	Correctly creating, rotating and flipping of a patch
testRemovePatch	Correctly removing a patch after it is used
testPatchSelection	Determine the 3 correct available patches
testMovePatchToken	PatchToken changes when patches are chosen

Table 1: The 10 twin team tests and the tested function

```

public class PurchasablePatch extends Patch implements IPatch {

    public PurchasablePatch(String patchString,
        int buttonCost, int timeTokenCost, int buttonScore) {
        super( stringToList(patchString), buttonCost, timeTokenCost, buttonScore);
    }

    //Adapt their patch string into our List<List>> Boolean
    private static List<List<Boolean>> stringToList(String patchString){
        List<List<Boolean>> spaces = new ArrayList<>();
        //Split on empty space " "
        String[] strings = patchString.split(" ");
        /*
        Convert each element of array to true/false;
        note that due to rectangular matrix, maxX is always same
        */
        int maxX = strings[0].length();
        //Loop over array
        for (String string : strings) {
            List<Boolean> rowSpaces = new ArrayList<>();
            for (int x = 0; x < maxX; x++) {
                char c = string.charAt(x);
                rowSpaces.add(c == ('1'));
            }
            spaces.add(rowSpaces);
        }
        return spaces;
    }
    ...
}

```

Figure 10: Example of an adapter, in this case part of PurchasablePatch, implementing the twin team's IPatch interface, and extending our existing Patch class

com.patchwork.app.adapters

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
LeatherPatch		58%		n/a	5	6	5	7	5	6	0
PatchBoard		92%		100%	0	7	2	19	0	5	0
PurchasablePatch		100%		100%	0	13	0	30	0	8	0
AdaptedGame		100%		100%	0	6	0	19	0	5	0
Deck		100%		100%	0	9	0	17	0	7	0
AbstractToken		100%		100%	0	6	0	11	0	4	0
AdaptedTimeBoard		100%		n/a	0	2	0	6	0	2	0
PlayerType		100%		n/a	0	1	0	2	0	1	0
Bank		100%		n/a	0	2	0	3	0	2	0
TimeToken		100%		n/a	0	1	0	1	0	1	0
PatchToken		100%		n/a	0	1	0	1	0	1	0
Total	12 of 497	97%	0 of 24	100%	5	54	7	116	5	42	0

Figure 11: Line and branch coverage of the adapted classes

6 Design patterns

The MVC pattern is used to structure the entire application. Game and the classes it depends on represent the model, TUI/GUI and GameInput are the view and GameController is the controller. This structure was applied because this application is adequately small for this division to work decently well.

User input is communicated to GameController using the observer pattern. GameInput (or rather, an implementation of it) is the subject to which a GameInputObserver (interface) implementation like GameController can subscribe in order to observe new inputs of the Move enum.

The factory pattern is used for constructing patches. PatchFactory has the createTimeBoardPatches method which creates the patches surrounding the time board. Furthermore, there is the createSpecialPatch which creates the 1x1 patch which has no cost and no reward. This application of the factory pattern is quite useful as the code for constructing many different patches is not very nice to read and would cause a lot of clutter in the Patch class otherwise. Having this construction elsewhere and only calling it by name is much nicer here.

The composite pattern is applied in order to represent the state of the game in GameController. This information is used by GameController itself and is used by the TUI in order to draw the right information. GameState is an abstract class which has property *type* of type GameStateType (enum), and the abstract method *getInstructionsString*. All four game states (PickMove, PickPatch, PlacePatch, Finished) extend GameState. This composition allows GameController to represent a game state using only a single variable, and uses its *type* property to safely cast game states to their specific types when needed.

The adapter pattern is applied in order to make the twin team tests work. Adapters allow different implementations of similar concepts to be made interoperable by adding some code in between instead of changing these implementations themselves. On a higher level, GUI (which extends GameInput) could also be considered an adapter because it adapts the JavaFX key press API based on key codes to the GameInput API which uses the Move enum.

Finally, there is also an application of a design pattern which is probably a bit less appropriate. The GridModifier class in the utils package uses the strategy pattern in order to implement different operations on patches. Use of the strategy pattern allows for implementing a grid modification using only a few small functions. At first it seemed useful because four grid modifications were implemented (rotate left/right and mirroring up/right). The use of the strategy pattern was already a bit convoluted for four modifications but later on we realized that only rotation needed to be implemented. This made the strategy pattern implementation even more overkill, so unless more patch manipulation options might be added in the future, this pattern is not needed here and makes the code less maintainable due to its added size.

7 Maintainability

Category 1: Improving Maintainability for Units

Units are the smallest groups of code(methods) that can be maintained and has to be limited to fewer lines of code(15 lines), fewer parameters(max 4), No code duplication and fewer branches

Reducing the Size of units to 15 lines: In our case , few of the units were having more than 15 lines of code which made it a little difficult to analyze and understand them. Also the longer units had code which satisfies multiple functionalities which were explained through comments.This has been calculated using Tool "Metrics Reloaded" on IntelliJ. These large units have been extracted and refactored into multiple units according to their functionality with not more than 15 lines of code. Figure 12,13 shows the metrics of number of lines of code before refactoring and after the refactoring

Limiting the Number of Branches to four:Measuring of number of branch points can be calculated by the minimum number of paths needed to cover all branches created by all branch points in the specific unit. This is called branch coverage. To calculate this we use McCabe complexity which should be Less than or equal to 5 obtained by using Code Metrics plugin in IntelliJ. To achieve this few unnecessary if statements were removed which can be showed through an example from Figure 14 and Figure 15 shows the McCabe compexity of the unit after refactoring which is 5

Category 2: Improving Maintainability for Modules

Concept of a module translates to a class in object-oriented languages such as Java. This module-level guideline addresses relationships between classes to achieve loose coupling.

Coupling Architecture components loosely: Having a good architecture makes it easier to find the source code that you are looking for and to understand how (high-level) components interact with other components. A component is part of the top-level division of a system. It is defined by a system's software architecture, so its boundaries should be quite clear and should have few entry points and a limited amount of information shared among components(encapsulated). In our case the high level components / modules are GameInput, GameController, TUI and Game. There are only few function calls between them which proves that they are loosely coupled it can be seen through Figure 16.

Category 3: Improving System level Maintainability

Writing Automated Tests: The testing framework used to write automated tests is JUnit. Several test cases have been written even covering the system tests to cover testing of end-to-end functionalities of the application.All the functionalities of the application can be tested with once click to see if any changed made to the application while developing any feature is causing any bugs or issues in the application. More on testing is covered in the section 4 which also includes the coverage of automated tests written.

Writing Clean Code:One of the important rule for writing clean code are to use limited comments and to remove code in comments. Our application has very few comments in it since while refactoring most the comments are turned into new units replacing the comment with a justified method name explaining the functionality of the unit without any additional comments. And also unrelated comments are removed from the code which can make the code-base more confusing. Also the dead code and the code in the comments is also eliminated. Variable names are well defined with a concise and meaningful name explaining what the variable stores. Exceptions were also handled properly for the top level components and also specific exceptions are thrown in cases failure of the game according to rules

method	CLOC	JLOC	LOC	NCLOC	RLOC
com.patchwork.app.backend.QuiltBoard.hasSevenBySeven()	9	0	41	32	27.89%
com.patchwork.app.frontend.TUI.drawPatches(List<Patch>, int, PrintStream)	0	0	31	31	18.90%
com.patchwork.app.backend.TimeBoard.TimeBoard(List<Player>, int)	0	0	30	30	30.00%
com.patchwork.app.backend.TimeBoard.movePlayer(Player, int)	3	0	28	25	28.00%
com.patchwork.app.backend.Game.placePatch(Player, Patch, int, int)	6	10	36	24	20.00%
com.patchwork.app.backend.Game.movePastNextPlayer(Player)	7	7	28	19	15.56%
com.patchwork.app.backend.Game.finalizeGame()	6	3	24	18	13.33%
com.patchwork.app.backend.GameController.handlePlacePatchMovePatch(Patch, Move)	0	0	18	18	8.18%
com.patchwork.app.frontend.TUI.drawTimeBoard(PrintStream)	1	0	18	18	10.98%
com.patchwork.app.backend.GameController.handlePlacePatch(Patch, Move)	1	0	17	16	7.73%
com.patchwork.app.backend.Inputs.ScannerInput.getMoveFromInput(String)	0	0	16	16	50.00%
com.patchwork.app.frontend.TUI.drawQuiltBoardWithPatch(QuiltBoard, Patch, int, int)	3	0	19	16	11.59%
com.patchwork.app.backend.GameController.getMove()	0	0	15	15	6.82%
com.patchwork.app.backend.Game.getSpaceElementsMoveResult(Player, List<SpaceEl>)	5	8	22	14	12.22%
com.patchwork.app.backend.GameController.mainloop()	0	0	14	14	6.36%
com.patchwork.app.backend.QuiltBoard.canPlace(Patch, int, int)	8	0	22	14	14.97%
com.patchwork.app.backend.TimeBoard.getCurrentPlayer()	0	0	14	14	14.00%
com.patchwork.app.backend.GameController.handleMove(Move)	0	0	13	13	5.91%

Figure 12: Number of Lines of Code of Units before Refactoring for Maintainability

method	CLOC	JLOC	LOC	NCLOC	RLOC
com.patchwork.app.backend.GameController.handlePlacePatch(Patch, Move)	1	0	17	16	7.73%
com.patchwork.app.backend.Inputs.ScannerInput.getMoveFromInput(String)	0	0	16	16	50.00%
com.patchwork.app.backend.TimeBoard.arrangingTimeBoard(List<Player>)	0	0	16	16	13.11%
com.patchwork.app.frontend.TUI.drawQuiltBoardWithPatch(QuiltBoard, Patch, int, int)	3	0	19	16	11.59%
com.patchwork.app.backend.GameController.getMove()	0	0	15	15	6.82%
com.patchwork.app.backend.QuiltBoard.hasSevenBySeven()	7	0	22	15	14.47%
com.patchwork.app.backend.Game.getSpaceElementsMoveResult(Player, List<SpaceEl>)	5	8	22	14	11.96%
com.patchwork.app.backend.GameController.mainloop()	0	0	14	14	6.36%
com.patchwork.app.backend.QuiltBoard.canPlace(Patch, int, int)	8	0	22	14	14.47%
com.patchwork.app.backend.TimeBoard.getCurrentPlayer()	0	0	14	14	11.48%
com.patchwork.app.backend.Game.getWinningPlayerandHisScore(Map<Player, Integer>)	3	0	16	13	8.70%
com.patchwork.app.backend.Game.performMovingthePlayer(Player)	1	0	14	13	7.61%
com.patchwork.app.backend.GameController.handleMove(Move)	0	0	13	13	5.91%
com.patchwork.app.backend.Inputs.MockGameInput.run()	0	0	13	13	61.90%
com.patchwork.app.backend.QuiltBoard.placePatch(Patch, int, int)	8	0	21	13	13.82%
com.patchwork.app.utils.GridModifier.AbstractGridModifier.modify()	0	0	13	13	52.00%
com.patchwork.app.backend.Game.placePatch(Player, Patch, int, int)	4	10	22	12	11.96%
com.patchwork.app.backend.GameController.handlePlacePatchRotatePatch(Patch, Move)	0	0	12	12	5.45%
com.patchwork.app.backend.GameController.handleSelectMove(Move, T, List<T>)	1	0	13	12	5.91%
com.patchwork.app.backend.GameController.waitForGameCycle(int)	0	0	12	12	5.45%
com.patchwork.app.backend.PatchList.getAvailablePatches()	2	0	14	12	41.18%
com.patchwork.app.frontend.TUI.drawTimeBoardSpaces(int, int, int, boolean, PrintStream)	0	0	12	12	7.32%
com.patchwork.app.backend.Game.performPlacingPatch(Player, Patch, int, int)	2	0	13	11	7.07%

Figure 13: Number of Lines of Code of Units After Refactoring for Maintainability


```

public void arrangingTimeBoard( List<Player> players ) {
    List<Integer> buttonsIndicesList = Arrays.asList(8,15, 23, 27, 32,35,41,49);
    List<Integer> specialPatchIndicesList = Arrays.asList(20, 29, 37, 43, 48, 51);
    for (int i = 0; i < NR_SPACES; i++) {
        if(i == 0){
            add_both_players_onTimeBoard(players);
        }
        else if(buttonsIndicesList.contains(i)){
            add_button_to_TimeBoard();
        }
        else if(specialPatchIndicesList.contains(i)){
            add_special_patch_to_TimeBoard();
        }
        else{
            this.spaces.add(new ArrayList<SpaceElement>());
        }
    }
}

24+ private void arrangingTimeBoard( List<Player> players ) {
25+     Set<Integer> buttonsIndicesList = new HashSet<>(Arrays.asList(8,15, 23, 27, 32,35,41,49));
26+     Set<Integer> specialPatchIndicesList = new HashSet<>(Arrays.asList(20, 29, 37, 43, 48, 51));
27+     add_both_players_onTimeBoard(players);
28+     for (int i = 0; i < NR_SPACES; i++) {
29+         if(buttonsIndicesList.contains(i)){
30+             add_button_to_TimeBoard();
31+         }
32+         else if(specialPatchIndicesList.contains(i)){
33+             add_special_patch_to_TimeBoard();
34+         }
35+         else{
36+             this.spaces.add(new ArrayList<SpaceElement>());
37+         }
38+     }
39+ }

```

Figure 14: Refactoring the unit to reduce the branch points

```

private void arrangingTimeBoard( List<Player> players ) { Complexity is 5 Everything is cool!
    Set<Integer> buttonsIndicesList = new HashSet<>(Arrays.asList(8,15, 23, 27, 32,35,41,49));
    Set<Integer> specialPatchIndicesList = new HashSet<>(Arrays.asList(20, 29, 37, 43, 48, 51));
    add_both_players_onTimeBoard(players);
    for (int i = 1; i < NR_SPACES; i++) {
        if(buttonsIndicesList.contains(i)){
            add_button_to_TimeBoard();
        }
        else if(specialPatchIndicesList.contains(i)){
            add_special_patch_to_TimeBoard();
        }
        else{
            this.spaces.add(new ArrayList<>());
        }
    }
}

```

Figure 15: McCabe Complexity of a unit

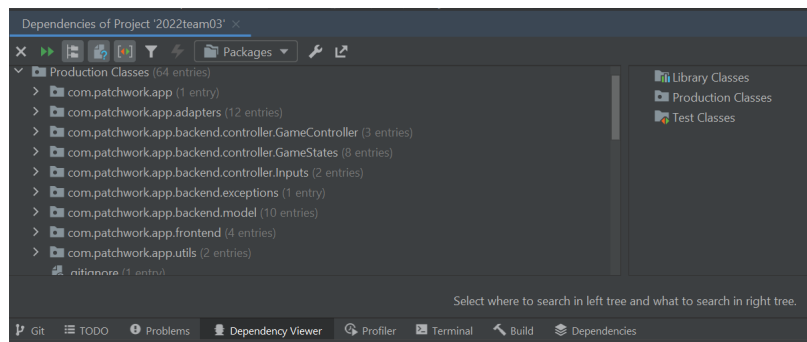


Figure 16: Entrypoints for Components

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 generic 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

The problem that was encountered and debugged was a case of tests only failing in the CI pipeline. The *defect* in question was a supposedly unused value WAITING of the Move enum which was previously used to indicate that no input was received. WAITING should not be used anywhere anymore after previous refactor but in fact, it was still set as the initial output move for MockGameInput which is used exclusively for testing. This *defect* caused an *infection* in GameController by providing the WAITING move to GameController on its first game cycle. This *infection* propagated as a race condition to GameFlowTest which has a helper function *executeMoves* for inputting moves and then waiting until they are processed. The *executeMoves* function would think that the moves it provided itself had been processed, but it was actually the WAITING move. As a result, this function exited before the right move was processed which caused assertions in unittests to be executed too early. Thus, a *failure* occurred in the form of failing unit tests.

The problem has only occurred when executing tests in the CI pipeline. It might be the case that the CI worker is always under high CPU load which causes a different execution order from that of a laptop under low CPU load. The fact that the issue initially occurred on CI, was seemingly not reproducible on a laptop and was consistently occurring on CI provided the *track, reproduce* and *automate* steps of TRAFFIC automatically. To *find origins* of the problem, first some additional *assertions* were added to failing test cases to figure out if the test was implemented incorrectly, or that the move had not been processed after *executeMoves* had finished. The latter turned out to be the case. Further investigation was done by looking at the output of the failing test cases. It was noticed that the set of failing tests were random and the failure always occurred on the first assertion of the test case. Since considerable time had been put in verifying the behaviour of *executeMoves*, it was reasonable to assume that there was an issue in the initial setup of the game. Thus it was concluded that possible origins of the infection were any mock class used for testing, or *AbstractGameTest* which sets up and tears down all system tests.

When *focusing* on the possible origins of the infection, some *logging* was added to GameController as it was the next class in the infection chain (after *executeMoves*). Furthermore, GameController was recently refactored and therefore a likely suspect of any issue. From the logs it was noted that a WAITING move was being input and processed at the start of each game. The problem was then *isolated* to MockGameInput by inspecting the code which caused the bad initial input to be found. The problem was *corrected* by setting the initial input value to null, indicating that there is no input provided yet.

10 Functioning of the product

The final product is presented as a TUI wrapped in a GUI with only a text box, which made it easier to deal with user input. The user makes choices using the WASD keys to change their choice and Enter to confirm it. Q and E are used to rotate patches when placing them. The H key triggers some help information to be shown.

The product implements the gameflow of Patchwork: at the start of the player's turn, they pick whether to advance on the time board and receive buttons or to buy and place a patch on their quilt board. Advancing past the next player works as specified in the game rules and also rewards special patches and buttons if applicable.

If the player chooses to pick a patch, three options are presented and the selected patch must be placed on the players quiltboard. The player is also advanced the correct amount of steps and special patches, buttons and the 7x7 special tile are correctly rewarded if applicable. When both players have reached the end of the time board, the game is finished: the scores are calculated and the winner is shown.

One minor missing feature is that the entire collection of patches around the time board is never shown, meaning that players cannot make moves which think far ahead.

Application of skills

In order to properly develop this game, different skills and techniques were applied in order to facilitate the team and the development process. To start off, a *design* was drawn in the form of a class diagram to establish the architecture of the product and have the team all on the same page. Next up, was the division of the task and organization of the work to be done. The *agile approach using Jira* provided a clear method to prioritize the work, divide it into sprints and keep track of the progress.

Now that the team had agreed on the work to be done, the actual development could start. By applying a *test-driven development* approach, the amount of code written is only minimal. Furthermore, since the functionality of the code is validated from the start, bugs are more easily noticed, resulting in a higher quality product. The *continuous integration* pipeline, which is run automatically after each push to the repository, ensures that the code indeed builds and complies to the defined test set. Whenever tests failed, *debugging* methods had to be applied in order to find and resolve the bug that causes the defect.

In order to keep the code nicely structured, several *design patterns* were used in the design, improving the flexibility and extensibility of the code. By applying *maintainability* guidelines, the code is easily readable and maintainable, both during development and in the future. In improving this structure in the codebase, *refactoring* has played a big role: several elements have been refactored in order to fix the lack of flexibility or to improve the maintainability of the code. The *requirements change*, however, created some overhead to the code, as several adapters had to be created in order to facilitate logic in the provided test cases.

11 Appendices

A Spotbugs Security and Static Analysis

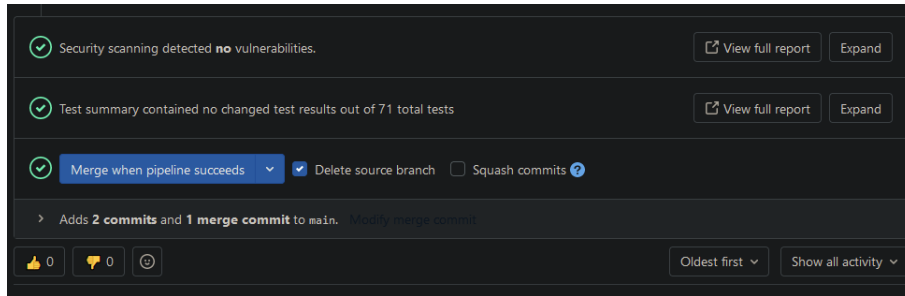


Figure 17: SAST Menu in a Merge Request

Summary

Classes	Bugs	Errors	Missing Classes
41	22	0	0

Files

Class	Bugs
com.patchwork.app.backend.Game	1
com.patchwork.app.backend.GameController	7
com.patchwork.app.backend.GameControllerFactory	1
com.patchwork.app.backend.GameFactory	2
com.patchwork.app.backend.GameResult	2
com.patchwork.app.backend.Patch	1
com.patchwork.app.backend.PatchFactory	1
com.patchwork.app.backend.PatchList	2
com.patchwork.app.backend.QuitBoard	1
com.patchwork.app.backend.TimeBoard\$SpaceElement	1
com.patchwork.app.backend.Inputs.MockGameInput	1
com.patchwork.app.backend.Inputs.ScannerInput	1
com.patchwork.app.frontend.TUI	1

com.patchwork.app.backend.Game

Bug	Category	Details	Line	Priority
Switch statement found in com.patchwork.app.backend.Game.getSpaceElementsMoveResult(Player, List) where default case is missing	STYLE	SF_SWITCH_NO_DEFAULT	119	Medium

com.patchwork.app.backend.GameController

Bug	Category	Details	Line	Priority
Switch statement found in com.patchwork.app.backend.GameController.handleMove(Move) where default case is missing	STYLE	SF_SWITCH_NO_DEFAULT	113	Medium
Switch statement found in com.patchwork.app.backend.GameController.handlePlacePatchMovePatch(Patch, Move) where default case is missing	STYLE	SF_SWITCH_NO_DEFAULT	190	Medium
Switch statement found in com.patchwork.app.backend.GameController.handlePlacePatchRotatePatch(Patch, Move) where default case is missing	STYLE	SF_SWITCH_NO_DEFAULT	210	Medium
Method intentionally throws RuntimeException.	BAD_PRACTICE	THROWS_METHOD_THROWS_RUNTIMEEXCEPTION	241	Medium
Method intentionally throws RuntimeException.	BAD_PRACTICE	THROWS_METHOD_THROWS_RUNTIMEEXCEPTION	224	Medium
Method intentionally throws RuntimeException.	BAD_PRACTICE	THROWS_METHOD_THROWS_RUNTIMEEXCEPTION	53	Medium
Unread field: com.patchwork.app.backend.GameController.turnFinished	PERFORMANCE	URF_UNREAD_FIELD	18	Medium

Figure 18: Example of a Part of a Spotbugs Report

B Individual Coverage Reports

com.patchwork.app.backend.controller.GameController

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
GameControllerFactory		91%		n/a	1	7	1	12	1	7	0
GameController		93%		82%	11	65	12	169	0	27	0
Total	45 of 716	93%	12 of 69	82%	12	72	13	181	1	34	0

Figure 19: Coverage report of controller package

com.patchwork.app.backend.exceptions

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
GameException		100%		n/a	0	1	0	2	0	1	0
Total	0 of 4	100%	0 of 0	n/a	0	1	0	2	0	1	0

Figure 20: Coverage report of exception package

com.patchwork.app.frontend

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
TUI		99%		98%	2	49	1	115	1	17	0
Total	2 of 643	99%	1 of 64	98%	2	49	1	115	1	17	0

Figure 21: Coverage report of front end package

C Jira board images

com.patchwork.app.backend.controller.Inputs




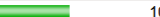






Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
ScannerInput		70%		50%	2	6	5	15	1	4	0
MockGameInput		80%		100%	0	4	3	11	0	3	0
ScannerCommands		100%		n/a	0	4	0	7	0	4	0
GameInput		100%		100%	0	4	0	9	0	3	0
ScannerCommandsFactory		100%		n/a	0	2	0	13	0	2	0
Total	20 of 179	88%	2 of 8	75%	2	20	8	55	1	16	0

Figure 22: Coverage report of input package

com.patchwork.app.backend.model

























Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
GameFactory		88%		80%	3	9	3	17	3	9	0
Game		95%		80%	8	35	6	78	0	11	0
Patch		98%		78%	3	13	0	19	0	6	0
QuitBoard		99%		90%	6	42	0	72	0	10	0
GameResult		100%		n/a	0	1	0	4	0	1	0
TimeBoard.SpaceElement		100%		n/a	0	2	0	5	0	2	0
PatchListFactory		100%		n/a	0	2	0	4	0	2	0
TimeBoard.SpaceElementType		100%		n/a	0	1	0	4	0	1	0
Player		100%		100%	0	9	0	19	0	7	0
PatchList		100%		100%	0	6	0	20	0	3	0
TimeBoard		100%		100%	0	24	0	72	0	13	0
PatchFactory		100%		100%	0	5	0	116	0	4	0
Total	31 of 3,720	99%	18 of 159	88%	20	149	9	430	3	69	0

Figure 23: Coverage report of model package

com.patchwork.app.backend.controller.GameStates



















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
Finished		95%		100%	1	4	1	9	1	3	0
GameState		100%		n/a	0	1	0	1	0	1	0
MoveResult		100%		n/a	0	3	0	6	0	3	0
PickMove.MoveOption		100%		n/a	0	1	0	3	0	1	0
PlacePatch		100%		n/a	0	3	0	11	0	3	0
PickPatch		100%		n/a	0	3	0	11	0	3	0
GameStateType		100%		n/a	0	1	0	5	0	1	0
PickMove		100%		100%	0	5	0	22	0	4	0
Move		100%		100%	0	15	0	15	0	5	0
Total	1 of 414	99%	0 of 24	100%	1	36	1	83	1	24	0

Figure 24: Coverage report of gamestates package

com.patchwork.app.utils















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed
GridModifier.GridLeftRotateModifier		100%		n/a	0	4	0	6	0	4	0
GridModifier.GridMirrorUpModifier		100%		n/a	0	4	0	6	0	4	0
GridModifier.GridMirrorSidewaysModifier		100%		n/a	0	4	0	6	0	4	0
GridModifier.GridRightRotateModifier		100%		n/a	0	4	0	6	0	4	0
GridModifier		100%		n/a	0	5	0	5	0	5	0
GridModifier.AbstractGridModifier		100%		100%	0	4	0	12	0	2	0
ConsoleColor		100%		n/a	0	3	0	9	0	3	0
Total	0 of 266	100%	0 of 4	100%	0	28	0	50	0	26	0

Figure 25: Coverage report of utils package

TO DO	IN PROGRESS	DONE
	Implement SpaceElement and TimeBoard <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-8	Set up GitLab CI <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-4
		Implement Patch, PatchFactory and PatchList <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-6
		Implement Player and Quiltboard <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-7
		Implement drawing a QuiltBoard <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-10
		Write report for peer review 1 <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-12

Figure 26: Jira board during sprint 1

NOG DOEN	ACTIEF	GEREED
Implement placing a patch on the board <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-13	Implement Game and GameResult <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-9	Implement drawing TimeBoard and the surrounding Patches <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-11
Implementing feedback from sprint 1 <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-16	Implement control flow (making moves, picking patches etc.) <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-14	Writing improvement report for canvas <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-19
Writing report for peer review sprint 2 <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-17	Write Test cases for SpaceElement and Timeboard <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-15	
Implementing game interaction inputs <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-20	Changing lists in the TimeBoard and QuiltBoard due to static sizes <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-18	
	Implementing game interaction <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-21	

Figure 27: Jira board during sprint 2

TO DO	IN PROGRESS	DONE
	Improve documentation according to improvement plan <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-30	Finalize game flow and add more system tests <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-28
	Improve design pattern documentation <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-33	Write GUI which only shows text so that controls can be made more easy to use. <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-32
	Make maintainability report more concise <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-34	Update functioning of final product according to new GUI <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-35
	Final improvements to the report. <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-37	Implement special patches being removed after they are picked up. <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-36
	Add comments to complex pieces of code <input checked="" type="checkbox"/> <input type="checkbox"/> PPPP-38	

Figure 28: Jira board during sprint 3