Final Report AOOP: A 2D Game API

Joppe Smink, Linus Wass, Oscar Falck

April 2023

Abstract

In this report we will conclude the final project of the Advanced Object-Oriented Programming course at Halmstad University. It features an in-depth explanation and dissection of the structure, development, and testing phases of a general usage API for 2D tile-based games. The API has subsequently been utilized to create two games as examples and/or proofs of concept; namely Sokoban and 2048, which also will be discussed in the report. The report is intended for a university reader or a programmer that is interested in polymorphism and game development.

1 Introduction

API's are becoming increasingly necessary for large scale systems, code portability and reduction in development time. In this report, the implementation of Java Swing based 2D Game API is explained in-depth, from the API structure and the smallest components to the finalized code and practical implementation examples. The report is structured in a way that leads the reader through the entire development process. In section 2, the structure and implementation of the Game API and its assisting libraries are explained in-depth, with a subsection for each major component. This section includes textual explanations along with example code. Section 3 contains the testing phase of the project, explaining how and what is being tested. Section 4 mentions some interesting parts of the development process, going in-depth regarding problems and solutions the group came up with. The group's experiences concerning Git and version control is explained in section 5. Lastly, a list of potential improvements is present in section 6 and a conclusion and related results are present in section 7.

2 Structure and Implementation

In this section the API structure, relations and dependencies are explained, as well as their literal source code. The structure of the API is most readily explained in the following UML diagram. It contains the main Game class [1], which uses the Tile class [2] along with the SoundSystem [3] and TileView [4] interfaces. A default GraphicView implementation is added to the API, and the five classes in the lower half of the UML are the realizations of the API.

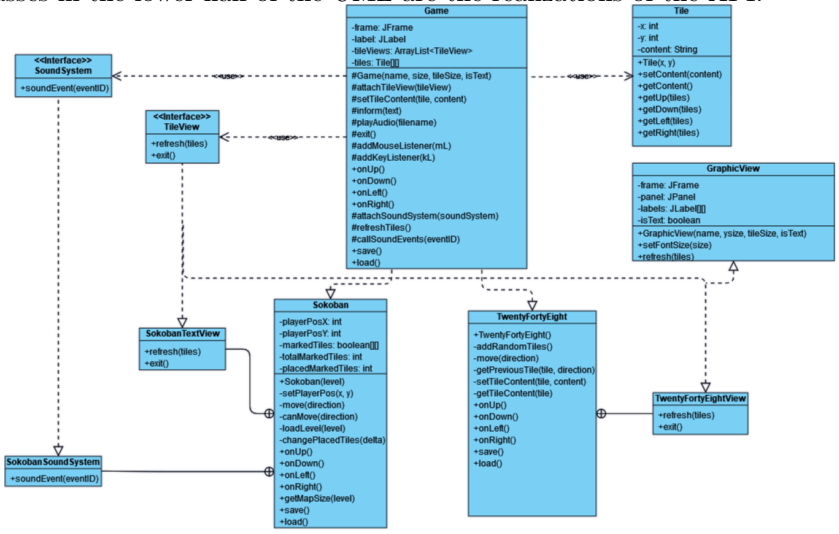

Figure 1. A UML diagram of the project

2.1 API Structure

2.1.1 Game class

The API is defined in the Game.java source file [1], which includes a declaration of the abstract template class Game. This class is the backbone of the API and all programs that want to utilize the API will need to inherit from the class. The Game class creates a JFrame for user input, saving, and loading; a two-dimensional array of the class Tile; and defines a couple of abstract functions essential to the base functionality of the API. The group opted for using an abstract class for the main API class, because this makes it possible to define abstract methods to turn it into a framework, while still adding boiler-plate code to turn the class into an API. By using this approach, we get a "best of both worlds" situation, blending the abstract functionality of interfaces and the explicit code of regular classes.

2.1.2 Tile class

The Tile class [2] is essentially a wrapper class for the String class, with some additional information specific to the Game board, such as its x and y coordinates, and additional functions to make accessing and manipulating neighbouring tiles easier.

2.1.3 TileView interface and its implementations

The TileView interface [4] is an interface that defines how a visual representation of the Game board should be presented and updated when a Tile value is changed. While technically a player could play and even complete a game with only the Game's movement functions (after game-specific rule implementations), this would be quite a difficult task without any visual way of interacting with the Game board. This is where the TileView interface comes in; the Game class has an ArrayList of TileView implementations, which are all informed after the content of a Tile is updated. Seeing how it is an interface, it is entirely up to the programmer to define their own implementations.

The group opted for this decision, rather than making TileView an abstract class like the Game class, because there is no universal code that needs to be run for the TileView to work, and the API should not make any assumptions and consequently restrict the developer. It is impossible to predict whether the developer would want the most bare-bones implementation used solely for debugging, the most extravagant visual system with bells and whistles attached to every component, or something in the middle.

In this project, two distinct types (practically three classes) of implementations of the TileView interface are provided as a proof of concept; namely the Swing based GraphicView and the text based SokobanTextView and TwentyFortyEightView classes, defined in their respective implementations [5][6]. The GraphicView class creates a second JFrame filled with JLabels to render the

sprites, whereas the text-based classes use the PrintStream System.out to write the Game board to the console.

2.1.4 SoundSystem interface and its implementations

The SoundSystem interface [3] defines a template for how the code should respond on event-based triggers. While this code is called after its intended usage; namely audio playback, it provides everything necessary for an arbitrary event-based system. The Game implementation is expected to define its own event ID's and call the callSoundEvents(eventID) method defined in the Game API, which in turn calls the soundEvent(eventID) function for every attached SoundSystem instance, which in turn runs the necessary event handler code (which is intended to be audio playback). The reasoning for the SoundSystem being an interface is identical to the reasoning for the TileView's interface approach.

2.2 API Code Implementation

2.2.1 Main

The example implementation of the Game API starts out in the Main class [7]. This class contains the necessary code for a graphical application used to pick the desired game. Besides the Java main function, it only defines one other function:

- The **startMenu()** function creates a JFrame that contains two JPanels: one for the button area and one for the text area; two buttons, one for each game; a JLabel for the text; and a JComboBox for choosing the Sokoban level. It includes two ActionListeners, one for each button. When the Sokoban button is pressed the program instantiates the Sokoban class, with the selected JComboBox value in its constructor, and similarly, a new instance of the TwentyFortyEight class is instantiated when the 2048 button is pressed.

2.2.2 Tile

The Tile class [2] can be seen as the base of the Game API, representing the smallest possible state variable in the game board. It is a wrapper class for the String class, with some additional variables and functions to localize a Tile instance among its peers. Besides getter and setter functions for the so-called content of the Tile, the class defines four additional methods to make Tile content manipulation easier. These functions are defined as such;

- The functions **getUp/getDown/getLeft/getRight(tiles)** are rather self-explanatory: they return the tile next to the instance of the current tile, based on the complete board of tiles sent in the parameter. If the current instance happens to be at an edge, the function will notice this and return null instead, to avoid any potential exceptions being thrown.

2.2.3 TileView

The TileView interface [4] is the code responsible for presenting the Game board to the user. Depending on the type of game, the implementation will vary, and in this project three examples have been developed as a proof of concept. Due to it being an interface, everything is left up to the developer to implement the representation as they choose.

- The **refresh(tiles)** is the main function of the TileView interface and should be called whenever the content of a Tile is changed (the **setTileContent(tile, content)** function in the Game API does this automatically). The new Game board is sent as a parameter and is free to be done with as the developer chooses.

```
public void refresh(Tile[][] tiles) {
    System.out.println("\n-------------------\n");

    for(int i = 0; i < tiles.length; i++) {
        for(int j = 0; j < tiles[0].length; j++) {
            String c = tiles[j][i].getContent();
            if(c.equals("")) {
                c = " ";
            }
            System.out.print(c + " ");
        }
        System.out.println();
    }
}
```

Code snippet 1. The text based TileView for 2048

- The **exit()** function is called for all attached TileViews whenever the Game's **exit()** function is called, such as on a win condition. Depending on the implementation, this function may stay empty, or clean up any left-over code.

2.2.4 SoundSystem

The SoundSystem interface [3] is responsible for the handling of event-based audio playback. A realization of the interface may be attached to the Game instance, after which it will be called when the **callSoundEvents(eventID)** method is called. The only function in the SoundSystem interface is the following function.

- The **soundEvent(eventID)** function is an abstract function that needs to be defined for the Game-specific use case. The Game implementation defines all potential sound event IDs and is responsible for calling the **callSoundEvents(eventID)** function with the appropriate ID when a sound

ought to be played. A SoundSystem implementation with the proper event ID to audio file correspondence should be added to the Game class in the initialization.

2.2.5 Game

The Game class [1] is the backbone of the API, with the Tile, TileView and SoundSystem units supplementing the implementation [2][4][3]. It manages this task by being an abstract class that defines the necessary elements and functions for a 2D tile-based game. The most important of these functions are explained below.

- The **Game(name, size, tileSize, isText)** constructor initializes a two-dimensional array of tiles to hold the content of the Game board, as well as a JFrame that is responsible for all player-game communication. This JFrame contains a button to which the action listeners are attached, as well as two buttons for saving and loading, respectively. Lastly, it also adds a default GraphicView TileView, to make it possible for the developer to get up and running as quickly as possible.
- The **onUp/onDown/onLeft/onRight()** functions are abstract declarations used to perform specific actions in the cardinal directions. They are never called locally and must be explicitly called by the programmer (either by utilizing the predefined Mouse- or KeyListener functions, or in any other potential function, such as in a response to web sockets or HID drivers).
- The **refreshTiles()** function contains a loop that goes through all added TileViews and informs each that the content of a Tile has been changed, and that the view should be updated. This has some interesting implications, such as the programmer being able to add multiple instances of the same TileView implementation for multiple screens. For example, by adding the included GraphicView implementation twice, the code creates two JFrames that both get updated after a move has been registered. These updates are consecutive, due to the code not being multi-threaded, but because of the speed they appear to be instantaneous.
- The **callSoundEvents(eventID)** function is similar in nature to the **refreshTiles()** function: it loops through all added SoundSystems and calls their respective **soundEvent(eventID)** functions, informing every SoundSystem of an event trigger.

2.2.6 Sokoban

The Sokoban class [5] is an example implementation of the Game API, utilizing nearly every function in the base class.

- The **Sokoban(level)** constructor calls its superclass' constructor, **Game(name, size, tileSize, isText)** (as required when inheriting from superclasses that do not have the default constructor). After the superclass returns from its constructor, the Sokoban constructor continues and initializes a two-dimensional boolean array responsible for keeping track of where the marked tiles are. This is necessary because there is no marked player sprite, and as such no marked player Tile type, which makes it impossible to know whether the player is standing on a marked tile or an unmarked tile. This means that without explicitly keeping track of where the marked tiles are, the marked tiles would become regular tiles once the player moves over them. After initializing the two-dimensional boolean array, the code attaches the components responsible for the user interface, namely: the Sokoban SoundSystem, the implemented SokobanTextView, and a KeyListener. It finally loads the given level using the **loadLevel(level)** function.

- The **loadLevel(level)** function loads a premade Sokoban level from the given file, by reading the content character by character and checking for hard-coded values, while skipping the new line character. Depending on the Tile type, the code will do one of three things besides marking the Tile type in the Game's Tile array; mark a marked tile in the boolean array, set the player's location to the current coordinates, or do nothing besides storing the type.

- The **move(direction)** function is responsible for all movement and is called in all four of the cardinal directions, specifying the direction in the parameter. On execution, it immediately checks whether the desired move is legal by calling the **canMove(direction)** function, exiting if it returns a negative. Once it has been established that the move is allowed, the function continues to set up the variables necessary for performing a move. Seeing as this is Sokoban, both the adjacent tile and the tile after the adjacent in the direction of movement must be checked, due to the possibility of crate movement. This is done by using the **getUp/getDown/getRight/getLeft(tiles)** functions defined in the Tile class. However, seeing as these functions are only related to the Tile, it is still necessary to also define delta variables in the X and Y directions to get the absolute values used in the boolean array and to update the player position. The code also keeps track of the number of crates on marked tiles, completing the game if the goal has been reached, and subtracting one when a crate is moved off a marked Tile to avoid prematurely ending the game.

```
if(tiles[playerPosX+deltaX][playerPosY+deltaY]
                    .getContent().equals(CRATEMARKED)) {
    changePlacedTiles(-1);
}

if(next.getContent().equals(CRATE) ||
             next.getContent().equals(CRATEMARKED)) {

    setPlayerPos(playerPosX+deltaX,playerPosY+deltaY);

    if(markedTiles[playerPosX+deltaX]
                        [playerPosY+deltaY] == true) {
        super.setTileContent(nextNext, CRATEMARKED);
        changePlacedTiles(+1);
        callSoundEvents(MARKED_TILE);
    } else {
        super.setTileContent(nextNext, CRATE);
        callSoundEvents(BOX_MOVE);
    }

    return;
}

setPlayerPos(playerPosX+deltaX, playerPosY+deltaY);
```

Code snippet 2. The core movement code for Sokoban

- The **canMove(direction)** function takes in the desired direction of movement and returns a boolean. It does this by, just like most Tile manipulation in the Game API, calling the Tile defined **getUp/getDown/getLeft-/getRight(tiles)** functions. These functions are called twice, one for each step away from the player, and uses the returned Tiles to extract the content in local variables. These variables are subsequently compared to predefined legal/illegal content types and a boolean value respecting these requirements is returned. One example is the content in the Tile directly adjacent to the player being blank, which would return an automatic pass. Another example is the content in the tile directly adjacent to the player being a wall, which would return an automatic denial. A third example is the content in the Tile directly adjacent to the player being a crate, and the content in the Tile behind it being blank, which would also return in a pass.

2.2.7 2048

2048 is the lesser game implementation of the two, both in terms of complexity and in terms of API usage [6]. Besides the helper functions to deal with empty contents/null Tiles and other API implementation specific functions, it only needs two functions to adhere to the given rules. These two functions are explained below.

- The **addRandomTiles()** function loops through and runs the following code for every Tile in the Game board: for each Tile, it checks whether it is empty or not, and if it is, the code performs a random check with a 10% probability of passing. If it passes this check, another random check is run, after which it first checks for a 20% pass rate for a new Tile with the value of 4. If this check fails, there is a 30% for it to become a 2, and otherwise the new Tile value will become a 1 (with a remaining 50% pass rate). After the code has performed these checks for each Tile in the Game board, it goes through all Tiles once more, searching for any empty or neighbouring Tiles with the same value. If none of these Tiles exist, no more possible moves are available and the game ends with a loss.

- The **move(direction)** function also loops through every Tile in the Game board, only this time taking the desired direction into account. The looping starts at the border of the desired direction (i.e., at the upper border when the up key is pressed, et cetera), after which it moves the currently selected Tile as far to the desired direction as possible, until it reaches the border, another Tile with a different value or another Tile with the same value, the last of which would result in a merge (i.e., the Tile at the top border exits immediately, the second Tile from the top border moves up one step if the Tile above is empty or has the same content, et cetera). This piece of code is then repeated for each subsequent Tile in the opposite direction (i.e., every Tile one step down if we follow the same example). This results in all Tiles being moved to the desired direction, one Tile at a time, and merging when possible.

```
for(int i = 0; i < 4; i++) {
    for(int j = 0; j < 4; j++) {
        switch(direction) {
            case UP:
            default:
                current = tiles[j][i];
                break;

            case DOWN:
                current = tiles[j][3-i];
                break;

            case LEFT:
                current = tiles[j][i];
                break;

            case RIGHT:
                current = tiles[3-j][i];
                break;
        }

        previous=getPreviousTile(current, direction);

        for(int k = 0; k < 4; k++) {
            if(previous == null) {
                break;
            }

            currentContent = getTileContent(current);
            previousContent =getTileContent(previous);

            if(previousContent != 0) {
                if(currentContent==previousContent) {
                    setTileContent(previous,
                                   2*currentContent);
                    setTileContent(current, 0);

                    if(2*currentContent == 2048) {
                        inform("You win!");
                        exit();
                        Main.main(null);
                    }
                }

                break;
```

```
                }

                setTileContent(previous, currentContent);
                setTileContent(current, 0);
                current = previous;
                previous = getPreviousTile(current,
                                           direction);
            }
        }
    }
```

Code snippet 2. The core movement code for 2048

3 Testing

The project is tested for stability and functionality by using JUnit, a unit testing framework for Java [8]. The reasoning for why the group opted to use JUnit, is because it is becoming the industry standard for writing and executing unit tests in Java, due to it being open source, its transferability, and its compatibility with other tools.

Due to JUnit being a core part of the obligatory lab exercises earlier in the course, the group had prior experience with the framework, something which assisted the development of this project. The course literature also helped the group, due to its examples of JUnit's usage [9]. JUnit was first tried with Visual Studio Code, but because Visual Studio Code ultimately is an editor, rather than a full-fledged IDE, like Eclipse (which has JUnit integration built-in), there were some complications. This resulted in this part of the project being written and executed in Eclipse.

The tests were written for Sokoban, seeing as this implementation is the most complex and has the most implemented functionality. The group argued that by testing a run of a Sokoban game, the Game logic, controls, conditions, Tile contents, saving and loading, and edge cases would be tested, effectively testing the entire API. The actual audial/visual systems could not be tested via JUnit, due to there being multiple potential points of failure, and were as such not included in the testing phase.

The Testing, defined in the *Test.java* file [10], starts out by initializing an instance of Sokoban with level 1, after which the player position is stored in their respective X and Y variables. The code subsequently manually calls the **onLeft()** function, triggering a move to the left into a wall (due to it being level 1), after which an assert statements checks whether the position in the X direction has changed. The code then calls the **onUp()** function, checks the Y coordinate, after which the **onRight()** function is called and the X coordinate is checked once more. The Tile above the player should at this point in time be a crate, and the content is as such checked for the appropriate value. The test follows up by calling the **save()** function, after which it enters a for loop,

executing an **onLeft()**, **onUp()**, and **load()** 50 times for stress testing and a proper saving/loading functionality. Were the saving and loading faulty, the next stage of the test would not work; namely the solution to the level. The code calls the proper functions in the proper order, stopping prior to the last move. It subsequently checks whether the total amount of placed marked tiles is 3 (there are a total of 4 in level 1), after which it calls the last **onUp()** call, which should increase the number, triggering the win condition. The win condition cannot be tested, but instead the total amount of placed marked tiles is compared to 4. If all these tests pass, the unit test passes, and it can be deducted that the Game API works.

It should be noted, however, that most variables in the Game API are either private och protected, and are as such not accessible in external files, such as the *Test.java* file. The group solved this issue by utilizing the reflection API, as seen in the code below.

```
Sokoban game = new Sokoban(1);
Class<Sokoban> c = (Class<Sokoban>) game.getClass();

Field playerPosX = c.getDeclaredField("playerPosX");
Field playerPosY = c.getDeclaredField("playerPosY");
Field placedMarkedTiles = c.getDeclaredField(
                                        "placedMarkedTiles");
playerPosX.setAccessible(true);
playerPosY.setAccessible(true);
placedMarkedTiles.setAccessible(true);

int oldPlayerX = (int) playerPosX.get(game);
int oldPlayerY = (int) playerPosY.get(game);

//Move left and check coordinates
game.onLeft();
assert(oldPlayerX == (int) playerPosX.get(game));

//Move up and check coordinates
game.onUp();
assert(oldPlayerY == (int) playerPosY.get(game)+1);

//Move right and check coordinates
game.onRight();
assert(oldPlayerX == (int) playerPosX.get(game)-1);

oldPlayerX = (int) playerPosX.get(game);
oldPlayerY = (int) playerPosY.get(game);

//Check content of Tile above
Field CRATE = c.getDeclaredField("CRATE");
```

```
CRATE.setAccessible(true);

assert(game.tiles[oldPlayerX][oldPlayerY-1].getContent()
                                        .equals(CRATE.get(game)));

//Save the game and subsequently move and load 50 times,
                                            overloading test
game.save();
for(int i = 0; i < 50; i++) {
        game.onLeft();
        game.onUp();
        game.load();
}

//Solution to level 1
/*Not included in the snippet for formatting purposes*/

//Check marked boxes
assert((int) placedMarkedTiles.get(game) == 3);

game.onUp();

//Check marked condition/win condition
assert((int) placedMarkedTiles.get(game) == 4);
```

Code snippet 3. The testing code

4 Interesting development hurdles

During development, the group encountered a few problems and other hurdles which had to be solved in interesting ways. This section covers a few of these problems and their explanations.

- The team discovered a minor yet potentially crucial bug during development, without the assistance of JUnit. In the example implementation of Sokoban, a graphical view with sprites is utilized [5]. These sprites are read as Images, and the file paths to the sprites are hardcoded as static final Strings (due to Java not supporting pre-processor macros). When comparing the contents of two Tiles, the code initially used the equality operator on the two Strings. This initially worked with no problems, precisely because the "macros" are in fact real Strings located in memory. The code would always use the same variable whenever setting the content of a Tile, which resulted in the content of the Tile and the predefined Strings being the same instance of the String class and thus being equal. However, once a Game has been saved, exited, and loaded again, the hardcoded variables have become completely new instances of

the String class, and when they are compared to the de-serialized Tile contents, would return in a negative. This is where the group realized their mistake, and subsequently swapped out all equality operators with the **equals()** function.

- When developing the controller window, the group initially added the ActionListeners to the JFrame itself. The group realized later in development that the most straightforward and concise way to trigger saving and loading would be by adding specific buttons to the already existing controller window. This approach solved the problem the group was facing; however, it did introduce a new problem. When you add focusable components (such as a JButton) to a container, the component takes over the focus of the container. Due to Java's design, ActionListeners are only active when their parent component is focused, and the Game would as such not respond to any inputs. The group noticed this error and decided to add a separate JButton solely responsible for housing the input ActionListeners, and not the typical JButton press ActionListener. By using this approach, the player can click on the control button, which does nothing but move the control to the JButton, which in turn houses the input ActionListeners responsible for the **onUp/onDown/onLeft/onRight()** functions. It would also have been possible to add these ActionListeners to both buttons responsible for saving and loading rather than adding a third button solely for housing the ActionListener, however, this solution works just as well, and the group preferred it to other potential solutions.

5 Version Control

During the development of the project, project version control was used to back up the code. Seeing as the members of the group were all writing together at the same time to ensure a full understanding of the code, the git repository was never branched, and the code was only pushed to the main branch. Furthermore, while we first tried to install a git plug-in to the chosen IDE, we were reaching obstacles due to log-in errors, and we opted to use git in its command line environment instead.

Overall, while we did not use git to its maximum capabilities and mainly used it as a backup/file sharing service, the experience was both interesting and informative. The group feels more comfortable with using git than before the project, and they feel that it will be an essential part of the development process of future projects.

6 Potential improvements

While the API has been completed and all the goals have been met, a project is never fully complete and there are always areas to add, remove, or tweak. A few of these suggestions are described below.

6.1 API

To make the API even more robust, it would be possible to add even more functions for customizability and freedom in development, such as making the controller JFrame optional and customizable, adding support for controller drivers, et cetera.

6.2 Game implementations

The Sokoban implementation of the Game API has been developed under the assumption that all levels would be supplied in external files. This simplified the development process, because it eliminated the need for an algorithm for level generation, a topic which can become rather complex, especially for a game like Sokoban. However, because Sokoban is based on the Game API, which requires each game to implement their own method; 2048, a game dependant on randomized content, was able to incorporate this. If a map generator were something to be desired, it would be possible to implement this in a map generator interface, where the developer supplies the interface with the rules specific for the game in question.

7 Results

All in all, the project was a success. The team felt that they were able to utilize their newly acquired knowledge in a challenging yet intriguing way, and the result of the project shows this. The developed Game API has been used to implement two separate games, and the team felt that the final project fulfils the given requirements. It became a general, flexible API; implementing attachable and detachable visual and audial components (of which both the components utilize subject-observer patterns for their implementations), as well as attachable and detachable controllers (due to the controlling being purely dependant on external calls to public functions, as is evident in the unit testing code).

As discussed in section 4, not all development problems necessarily stem from bad design, and it could be trivial human errors which cause the most detrimental problems. This is why brainstorming, reviewing and rewriting code together is such an important part of the development process and why it is not uncommon for teams to hire someone with a new set of eyes when they are struggling to solve a particular problem.

References

[1] J. Smink, L. Wass, and O. Falck. *Game API*. Location: Game.java. 2023.

[2] J. Smink, L. Wass, and O. Falck. *Tile Component*. Location: Tile.java. 2023.

[3] J. Smink, L. Wass, and O. Falck. *SoundSystem Component*. Location: SoundSystem.java. 2023.

[4] J. Smink, L. Wass, and O. Falck. *TileView Component*. Location: TileView.java. 2023.

[5] J. Smink, L. Wass, and O. Falck. *Sokoban Implementation*. Location: Sokoban.java. 2023.

[6] J. Smink, L. Wass, and O. Falck. *2048 Implementation*. Location: TwentyFortyEight.java. 2023.

[7] J. Smink, L. Wass, and O. Falck. *Example Entry Point*. Location: Main.java. 2023.

[8] Wikipedia contributors. *JUnit*. Accessed 2023-04-26. 2023. URL: `https://en.wikipedia.org/wiki/JUnit`.

[9] C. Horstmann. *Object Oriented Design & Patterns (3rd ed.)* 2024.

[10] J. Smink, L. Wass, and O. Falck. *Unit Testing Code*. Location: Test.java. 2023.