ALGORITHMS

UTRECHT UNIVERSITY

Investigation of Algorithms in Minesweeper

Author:
Mehrad Haghshenas (Student Number: 2822865)

Date:
November 12, 2023

1 Introduction

This project aims to investigate and analyse the *Minesweeper* game implemented by *Mitchell Sayer*. The selected Minesweeper code can be found on the following url, GitHub repository.[1] Accordingly, we will perform a static analysis and suggest potential code improvements where deemed necessary. The enhancements are done through *refactoring* the code. By doing so, we aim to address *code smells* and *bugs*, with the ultimate goal of optimizing the game's performance in terms of time complexity and space complexity.

The reason of choosing an investigation of *Games and Puzzles source Code* was specifically because I recently delved deep into the *Monogame* framework which is used for creating games in C#. Accordingly, even in a simple game like Tetris various aspects need to be taken into consideration to provide a satisfactory experience for the user. If the game is too slow, the logic is flawed, or the graphical user interface is unattractive, the user will be less willing to play the game. The first two issues can arise from code smells and bugs respectively. Therefore, I have decided to perform a static analysis on the famous Minesweeper game.

This paper is structured as following: Section 2 provides a historical background along with the logic and complexities behind the game. In section 3, an overview of the implementation code will be provided. Section 4 and 5 explain the experimental platform and provide a static analysis of the code respectively. Accordingly, in section 6, the proposed refactoring of the codes are provided. Section 7 provides the two tests, hypotheses, and the methodologies used in each test. Furthermore, section 8, provides the results and concludes this paper.

2 Background

To our surprise, the origin of the Minesweeper game is ambiguous. Nevertheless, the general consensus is that the first variation of the game, named *Mined-Out*, was developed in 1983 for home computers. The windows version of the game was later released by Microsoft in 1990, created by Robert Donner and Curt Johnson.[4,5] Moreover, despite the game's simplicity and widespread familiarity, the logic behind the game is rather complex. In fact, in a paper written by *Richard Kaye*, he mentions that the *Minesweeper consistency problem* is NP-complete. This implies that finding a polynomial time solution for this problem using a deterministic Turing machine is not possible. Likewise, he formulates the Minesweeper consistency problem as the following: *Given a grid partially marked with numbers, the problem is determining whether there exists a pattern of mines which satisfy the requirements of the numbers.*[2] This is very similar to the SAT problem and hence is a *decision problem* with a Boolean outcome. Being NP-complete underscores the challenge in converting the game's complex logic into an implementable form.[1]

[1] This was the second link in the *C2* option of the project ideas on Moodle.

3 Code Overview

Before delving deep into the experiments we will provide an overview of the code under analysis. The code can be found on the provided link in the introduction section. Nevertheless, we have attached it as an appendix to the end of this paper for further reference. In the following some basic statistics has been presented:

- **Number of Classes:** 1 (Minesweeper)
- **Number of Methods:** 13. These include:
  - main(String[] args) – main method
  - Minesweeper() – constructor
  - addRandomMines()
  - showTile(int r, int c)
  - clearEmpty(int row, int col)
  - checkLose()
  - checkWin()
  - surroundingClosed(int x, int y)
  - markItem(int x, int y, int n)
  - knownMineCount(int x, int y)
  - openNonMines(int x, int y)
  - solveGame()
  - actionPerformed(ActionEvent event)
- **Lines of Code:** The number of LOC is approximately 319 including comments and white lines.[2]

The game specifically comprises of three sections:

- *GUI Components:* This incorporates the JFrame, the colours, and in general all aspects related to the graphical user interface.
- *Game Logic:* This incorporates the logic behind the game; for example, the number of mines around a cell, the situations of winning and losing, etc. are all handled in this section.
- *Event Handling:* This section handles how the game should react to an event caused by the user.

[2] Note that the LOC depends on the structure of the code and how white spaces are used. For example a conditional statement can be written in one line instead of multiple lines.

In the code, various primitive and compound data types have been used; such as:
Primitive Data Types:

- **int:** Used for variables like mine counts, array dimensions, and loop counters.
- **boolean:** Used for flags such as lost and canSolve.

Compound Data Types:

- **Arrays:** 2D Array of `int` for tracking the number of each cell.
- **ArrayList:** `ArrayList<Integer>` for managing marked mines.

Some of the external libraries used in the code are:

- Java Swing (javax.swing.*) for GUI components.
- AWT (java.awt.*) for layout and color.

4 Experimental platform

The *hardware* used for the experiments was a macOS Ventura (Version 13.1) with 16 GB of RAM and an Apple M1 chip. The Apple's M1 chip includes 8 CPU cores. The software used was TMCBeans version 11.1. TMCBeans is similar to Netbeans with an auxiliary Test My Code plugin. Note that TMCBeans might cause some overhead which has been exempted from our analysis.

5 Static Analysis

At the first glance, some of the potential *improvements* to the code are:

- **Modularity:** The author has combined the GUI section, game logic, and event handler all in one class. With respect to the design perspective this is not a good practice. Using a design framework such as *MVC (Model-View-Controller)* will not only make the code more understandable, but may also lead to potential enhancements in terms of performance.
- **Inefficient Data Structures:** Replacing marked mines data structure from *ArrayList<Integer>* to *HashSet<Integer>*. This may lead to an improvement in the lookup times, as HashSet offers *O(1)* complexity compared to *O(n)* for ArrayList. Nevertheless, the space complexity is *O(n)* for both, with HashSet having slightly more space complexity due to hashing. In short, this change may improve the time complexity.

- **Method Complexity:** Some methods, like actionPerformed and addRandomMines, are quite long and complex. In order to achieve a better code structure, these methods should be broken down. Also, long methods have a high *cyclomatic complexity* which is undesired and makes the reasoning of the methods demanding.
- **Exception Handling:** This is not done in the code but is useful to achieve complete robustness.
- **Documentation and Comments:** The code has little to no comments.

Likewise, we have used *PMD*, an extensible cross-language static code analyzer.[8] [3] Particularly, by running `pmd check -d /Users/mehradhq/NetBeansProjects/Minesweeper/src -R rulesets/java/quickstart.xml -f text` in the command line, where the first argument is the path to the source files and the second is the location to the set of rules, the following *code smells* have been found:

`Suggestion1:` *All classes, interfaces, enums, and annotations must belong to a named package.*
File: MineSweeper.java:10
`Suggestion2:` *This final field could be made static.*
File: MineSweeper.java:22
`Suggestion3:` *Avoid using implementation types like 'ArrayList'; use the interface instead.*
File: MineSweeper.java:23
`Suggestion4:` *This for loop can be replaced by a foreach loop.*
File: MineSweeper.java:40
`Suggestion5:` *This statement should have braces.*
File: MineSweeper.java:40
`Suggestion6:` *Avoid using implementation types like 'ArrayList'; use the interface instead.*
File: MineSweeper.java:55
`Suggestion7:` *This statement should have braces.*
File: MineSweeper.java:74
`Suggestion8:` *This statement should have braces.*
File: MineSweeper.java:76
`Suggestion9:` *This statement should have braces.*
File: MineSweeper.java:78
`Suggestion10:` *This statement should have braces.*
File: MineSweeper.java:80
`Suggestion11:` *This statement should have braces.*
File: MineSweeper.java:82

[3] We have added how to setup PMD in the Appendix

`Suggestion12:` *This statement should have braces.*
File: MineSweeper.java:84
`Suggestion13:` *This statement should have braces.*
File: MineSweeper.java:86
`Suggestion14:` *This statement should have braces.*
File: MineSweeper.java:88
`Suggestion15:` *This statement should have braces.*
File: MineSweeper.java:109
`Suggestion16:` *This statement should have braces.*
File: MineSweeper.java:111
`Suggestion17:` *This statement should have braces.*
File: MineSweeper.java:113
`Suggestion18:` *This statement should have braces.*
File: MineSweeper.java:125
`Suggestion19:` *This statement should have braces.*
File: MineSweeper.java:137
`Suggestion20:` *This statement should have braces.*
File: MineSweeper.java:153
`Suggestion21:` *This statement should have braces.*
File: MineSweeper.java:161
`Suggestion22:` *This for loop can be replaced by a foreach loop.*
File: MineSweeper.java:165
`Suggestion23:` *This statement should have braces.*
File: MineSweeper.java:173
`Suggestion24:` *This statement should have braces.*
File: MineSweeper.java:182
`Suggestion25:` *This statement should have braces.*
File: MineSweeper.java:196
`Suggestion26:` *This statement should have braces.*
File: MineSweeper.java:216
`Suggestion27:` *This statement should have braces.*
File: MineSweeper.java:249
`Suggestion28:` *This if statement can be replaced by ‘return condition;‘*
File: MineSweeper.java:262
`Suggestion29:` *This statement should have braces.*
File: MineSweeper.java:263
`Suggestion30:` *This statement should have braces.*
File: MineSweeper.java:265
`Suggestion31:` *This for loop can be replaced by a foreach loop.*
File: MineSweeper.java:272

`Suggestion32:` *This statement should have braces.*
File: MineSweeper.java:284
`Suggestion33:` *This for loop can be replaced by a foreach loop.*
File: MineSweeper.java:288

The proposed code smells can be categorized into the following groups:

- It is a good practice to have a package for each Java class in a project. The suggestion 1 indicates the absence of a package.
- Suggestion 2 is for converting the final variable to be static. This can cause two improvements. First of all static variable are resolved at compile time (for primitive types and string constants). Therefore, this will lead to a slight improvement in the performance. Furthermore, static final fields are associated with the class meaning that they are allocated only once, leading to a memory efficiency.
- Suggestion 3 states that we should use an interface instead of *ArrayList*. This results into more flexibility in the fact that we can change the implementation without altering the rest of our code.
- The rest of the suggestions are more related to maintainability and readability. Accordingly, replacing *for* loop for *foreach* loop will enhance the readability. Likewise, including braces for conditional statements even when they only have one statement within the block, enhances the readability.

The Cyclomatic complexities of each method are mentioned as following - note these numbers have been achieved my running the following command in the terminal *pmd check -d /Users/mehradhq/NetBeansProjects/Minesweeper/src -R /Users/mehradhq/Downloads/rule.xml -f text*. The only difference is the set of rules we have defined which incorporates the cyclomatic complexities as well. These set of rules are included in the appendix as well.

`Cyclomatic for class MineSweeper:` *The class 'MineSweeper' has a total cyclomatic complexity of 126 (highest 27).*
File: MineSweeper.java:11
`Cyclomatic for method main(String[]):` *The method 'main(String[])' has a cyclomatic complexity of 1.*
File: MineSweeper.java:26
`Cyclomatic for constructor MineSweeper():` *The constructor 'MineSweeper()' has a cyclomatic complexity of 3.*
File: MineSweeper.java:32
`Cyclomatic for method addRandomMines():` *The method 'addRandomMines()' has a cyclomatic complexity of 27.*
File: MineSweeper.java:54

`Cyclomatic for method showTile(int, int):` *The method 'showTile(int, int)' has a cyclomatic complexity of 6.*
File: MineSweeper.java:96
`Cyclomatic for method clearEmpty(int, int):` *The method 'clearEmpty(int, int)' has a cyclomatic complexity of 9.*
File: MineSweeper.java:119
`Cyclomatic for method checkLose():` *The method 'checkLose()' has a cyclomatic complexity of 9.*
File: MineSweeper.java:133
`Cyclomatic for method checkWin():` *The method 'checkWin()' has a cyclomatic complexity of 9.*
File: MineSweeper.java:157
`Cyclomatic for method surroundingClosed(int, int):` *The method 'surroundingClosed(int, int)' has a cyclomatic complexity of 8.*
File: MineSweeper.java:177
`Cyclomatic for method markItem(int, int, int):` *The method 'markItem(int, int, int)' has a cyclomatic complexity of 10.*
File: MineSweeper.java:190
`Cyclomatic for method knownMineCount(int, int):` *The method 'knownMineCount(int, int)' has a cyclomatic complexity of 8.*
File: MineSweeper.java:210
`Cyclomatic for method openNonMines(int, int):` *The method 'openNonMines(int, int)' has a cyclomatic complexity of 8.*
File: MineSweeper.java:224
`Cyclomatic for method solveGame():` *The method 'solveGame()' has a cyclomatic complexity of 11.*
File: MineSweeper.java:240
`Cyclomatic for method actionPerformed(ActionEvent):` *The method 'actionPerformed(ActionEvent)' has a cyclomatic complexity of 17.*
File: MineSweeper.java:271

6 Refactoring

In the previous section, we have mentioned the code smells and some potential improvements. In this section, we will narrow our attention to only some aspects of the code. Specifically, the following refactoring will be applied to the code:[6, 7]

1. A package will be added to the classes in the project satisfying suggestion 1.
2. The *MINE* variable which is final will be *static* as well satisfying suggestion 2.
3. The *marked* variable will be of type *Set<Integer>* instead of *ArrayList<Integer>* satisfying suggestion 3 and the subpart *Inefficient Data Structures*.

4. The methods *actionPerformed* and *addRandomMines* will be broken down. This is because these two methods are the only ones with a Cyclomatic complexity higher than 10. The aim will be to have all methods will complexity less than 10. This satisfies the *Method Complexity part* mentioned in the previous section.

5. In one of our experiments, we will divide the GUI, Logic, and Event Handler to see if a good design will lead to any performance enhancement or not. This will test the subpart *Modularity* of the previous section.

To see how the methods were broken into smaller methods and how the class was split please review the refactored code which is presented in the appendix. In general, in all these approaches, the aim was to generate smaller code blocks such that the overall project still has the same *semantics & functionality* as the original code.

7 Methodology

In this experiment, two tests will be carried out. The first test assesses the impact of comprehensive refactoring on the scalability and performance of the Minesweeper game. The null hypothesis of this test is as following:
Null Hypothesis (H0): "Refactoring (including refactoring techniques mentioned in 1, 2, 3, and 4 of section *Refactoring*) of the Minesweeper code does not significantly improve the game's scalability in terms of handling larger grid sizes. This means that the refactoring will not lead to a more efficient performance in execution time or memory usage."
Alternative Hypothesis (H1): "Refactoring of the Minesweeper code, as described above, significantly improves the game's scalability in terms of handling larger grid sizes, leading to more efficient performance in both execution time and memory usage."
Independent Variable: Techniques 1, 2, 3, and 4 of the refactoring section will all form the independent variable. Accordingly, making *MINE* static, changing the data structure of *marked* from *ArrayList<Integer>* to *Set<Integer>*, and decreasing the cyclomatic complexity of the methods will be the independent variables.[4]
Dependent Variables:

- Execution time for *starting* the game as grid size increases.
- Memory usage as a function of grid size.

This hypothesis aims to test the overall impact of a series of code improvements on the performance and scalability of the Minesweeper game, particularly when the grid size becomes larger.

[4]Note in a more comprehensive analysis, each of these factors like refactoring data structures or code structure could be analyzed separately. However, in our paper, we will in general test how does conforming to standard design structures benefit in terms of time and space complexity compared to not abiding by the design standards.

The next test is exactly the same as before with the addition of splitting the class into three sections of *Logic, Gui, and EventHandler*. The hypothesis is as following:
Null Hypothesis (H'0): "Refactoring (all techniques in section *Refactoring*) of the Minesweeper code does not significantly improve the game's scalability in terms of handling larger grid sizes. This means that the refactoring will not lead to a more efficient performance in execution time or memory usage."
Alternative Hypothesis (H'1): "Refactoring of the Minesweeper code, as described above, significantly improves the game's scalability in terms of handling larger grid sizes, leading to more efficient performance in both execution time and memory usage."
Independent Variable: All techniques in section refactoring are the independent variables.
Dependent Variables:

- Execution time for *starting* the game as grid size increases.
- Memory usage as a function of grid size.

In general to answer the above two hypotheses, we have created two refactored version of the code. The first of which has only incorporated techniques 1, 2, 3, and 4 and the second refactored version has all techniques mentioned in the refactoring section. In short, the second refactored version has split the GUI, Logic, and Event handling in addition. Afterwards, for each of these refactored versions along with the original version of the code, we conduct the following experiment:
We set the grid size 10, 20, 30,... ,100. For each grid size the code was run 10 ten times. We have taken the average of the total time and memory for each grid size. This was to prevent any outliers. As a result, for each code version (we have three in total), we achieve 20 numbers, ten related to the number and ten related to the memory. Each of these numbers correspond to specific grid size. Ultimately, the time and the memory were measured by defining the following method in each project:

```
private static long[] runTest(int gridSize) {
    long startTime = System.currentTimeMillis();
    Runtime runtime = Runtime.getRuntime();
    runtime.gc();
    long memoryBefore = runtime.totalMemory() - runtime.freeMemory();

    new MineSweeper2(gridSize);

    long memoryAfter = runtime.totalMemory() - runtime.freeMemory();
    long endTime = System.currentTimeMillis();

    long memoryUsed = memoryAfter - memoryBefore;
    long duration = endTime - startTime;

    return new long[]{memoryUsed, duration};
}
```

Note that in *runTest*, the grid size is dynamic. Also before measuring the memory garbage collection is done to delete any existing objects. Moreover, the memory achieved by this code includes the overhead of the JVM. However, given that all three projects incorporate the JVM overhead, for comparison reasons this will not impact our experiment. Last but not least, the time measured is the *startup* time of each experiment. Meaning that we only measure the time that it takes for the game to startup. Nevertheless we deem this as the primary time as the number assignment and logic incorporation in the GUI happens at the startup of the game.

8 Results

The following tables indicate the results achieved by conducting the experiment.

Grid Size	Code 1 Time (ms)	Code 2 Time (ms)	Code 3 Startup Time (ms)
10	173	258	1263
20	156	133	1451
30	176	153	1876
40	245	181	2569
50	354	200	2447
60	568	231	3316
70	841	290	2914
80	1316	286	3733
90	1611	346	4101
100	2406	456	4698

Table 1: Average Time for each Grid Size across three code bases

Grid Size	Code 1 Memory (bytes)	Code 2 Memory (bytes)	Code 3 Memory Used (bytes)
10	1576676	3872440	7142192
20	4021598	3574340	27821792
30	8653293	3614076	67912832
40	17744654	5042543	36416880
50	21820005	3559532	48120640
60	40758936	3600794	23166488
70	63699699	6250956	94135128
80	128204491	3711813	92122256
90	201587040	4255257	110093144
100	652940115	6710794	123369472

Table 2: Average Memory for each Grid Size across three code bases

The following indicate the box plots achieved for each code base (which are referred to as *test* in the box plots).

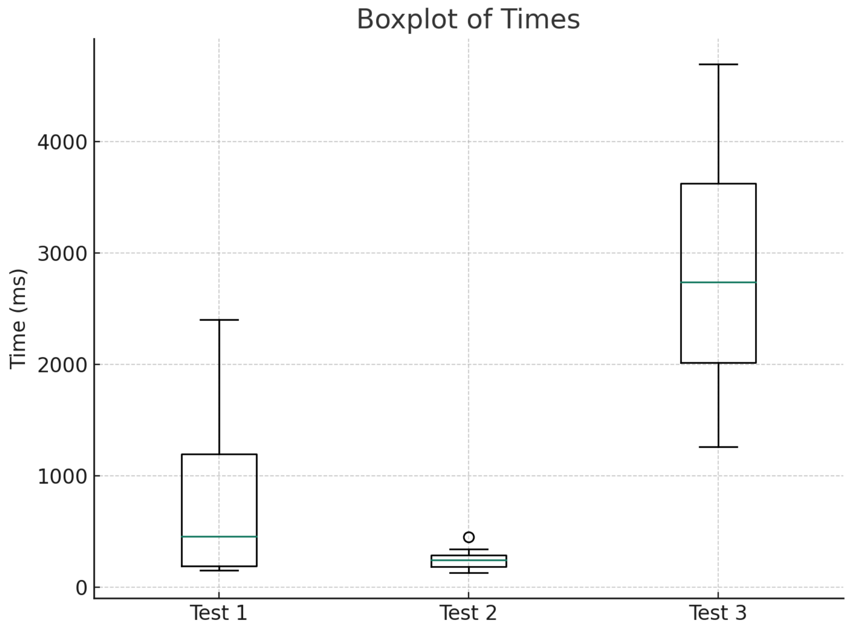

Figure 1: Box plots for time complexity of the three code bases.

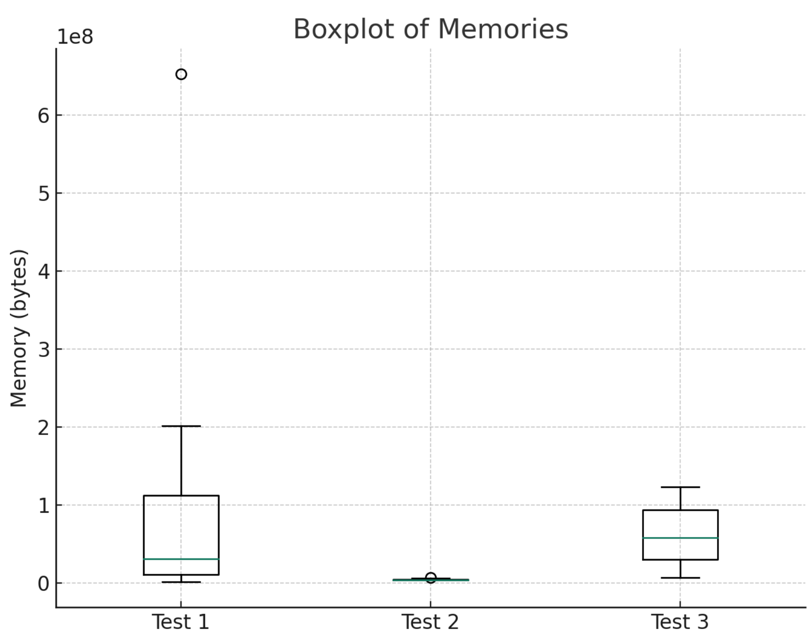

Figure 2: Box plots for space complexity of the three code bases.

Finally, the following indicate the linear graph of the three code bases:

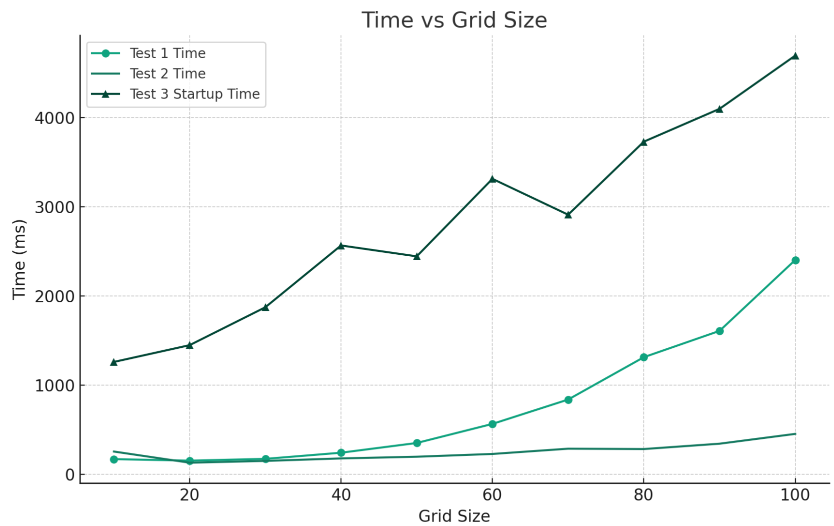

Figure 3: Linear graph of the time complexity for the three code bases.

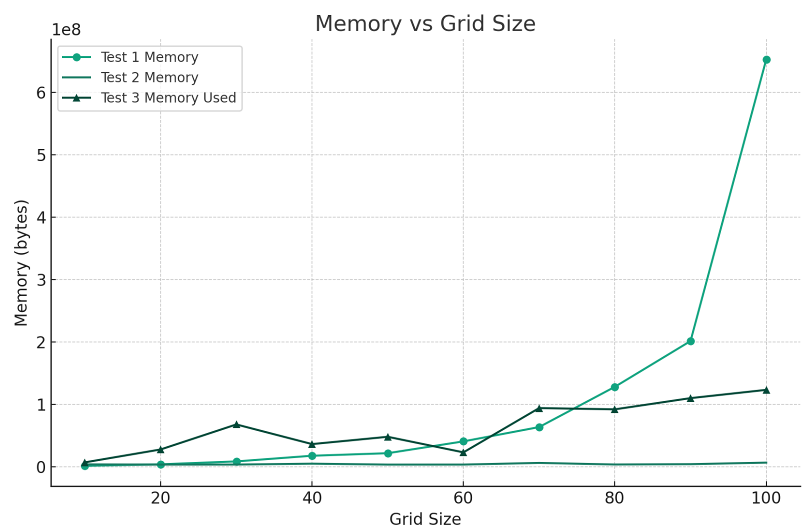

Figure 4: Linear graph of the space complexity for the three code bases.

The plots indicate the following result: *As seen in figure 1, code base 1, has the least time complexity and code base three has the most. This indicates that techniques 1, 2, 3, 4 improve the performance of the code in terms of time complexity. However, splitting the code into three different classes causes additional overhead in the time complexity.*

Therefore, splitting the classes might make the code more understandable; but, it increases the running time of the project. [5] *In terms of memory (figure 2), the results were the same as the time complexity. However, in terms of memory the three code bases did not perform that differently, whereas with regards to time complexity major differences were observed. Considering scalability, the code base outperforms the other two. Accordingly, figure 3 and 4 indicate that the code base 1 is not scalable at all and a drastic increase is seen in terms of time and memory when the grid size increases.*
To recap, we will answer the hypotheses mentioned in section 7. The mean of the numbers in columns 2, 3, and 4 of table 1 - which indicates the total startup time - are 784.6 ms, 253.4 ms, and 2836.8 ms respectively. The mean of the number in columns 2, 3, and 4 of table 2 - which indicates the memory use of each code version - are 114,100,650.7, 4,419,254.5, and 63,030,082.4 bytes in order.

- **H0:** if we run a t-test between columns 2 and 3 of the table 1 we will get a T-statistic of 2.4903 and a P-value of 0.0344. This indicates that the mean of column 2 is significantly higher than column 3 with a significance level of $\alpha = 0.05$. Thus, the null hypothesis can be rejected. In other words, refactoring the code will lead to an enhancement in terms of time complexity. Running a t-test between column 2 and 3 of table 2 will give T-statistic of 1.7416 and P-value of 0.1156. Again, the null hypothesis can be rejected with the same α, and hence, refactoring the code will lead to an enhancement in terms of space complexity. To encapsulate, H0 is rejected and the alternative hypothesis is accepted.

- **H'0:** Accordingly, seeing the results of the graphs, one can clearly conclude that the null hypothesis *H'0* is accepted as the numbers show no improvement in terms of time or space complexity when we refactor the code to version 3. However, for the sake of completeness we will run the t-tests on columns 2 and 4 of table 1. Accordingly, we get a T-statistic of -12.1548 and a P-value of approximately $6.90 * 10^{-7}$. Meaning that the null hypothesis can be rejected, but because the T-statistic is negative, this means that there is a significant evidence with a significance level of $\alpha = 0.05$ that splitting the interface and the logic has *more* time complexity than the non refactored code. For table 2, the T-statistic is 0.9317 and the P-value is 0.3758. Therefore, with a significance level of $\alpha = 0.05$, we cannot conclude that refactoring enhances the code in terms of space complexity. To sum up, refactoring the code in the shape of code base three, will surely lead to more time complexity, whereas for space complexity there are no evidence for any improvements. Therefore the null hypothesis H'0 is accepted with a significance level of 0.05.

[5] This has come much to our surprise. However, we think that the reason may root in the fact that the overall cyclomatic complexity of the three classes exceed the one class. Also the refactoring of the class might need further optimization such as reducing redundant code which are present in each of the three classes.

As a result, introducing a package, making the final variable static, changing the variable type from ArrayList to an interface Set, and most importantly breaking down the methods to have a maximum of 10 cyclomatic complexity of each method will increase the memory and time performance of the project. The cyclomatic complexity of each method in code base 2 and 3 are included in the appendix. For further work, one can run an experiment to analyse the impact of each of the refactoring techniques separately. Moreover, the impact of exception handling using *try & catch* can be investigated in terms of impact on the time and memory performance. Last but not least, more rules can be incorporated in PMD to achieve more suggestions for code smells which might lead to further optimization or we could use a richer software such as *Sonar*.[3]

References

[1] Minesweeper and NP-completeness. https://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm
[2] Kaye, R. (2000). Minesweeper is NP-complete. Mathematical Intelligencer, 22(2), 9-15.
[3] Yazir, O. (2022, January 1). SonarQube Tutorial All Details with Examples! Software Test Academy. https://www.swtestacademy.com/sonarqube-tutorial/
[4] Wikipedia contributors. (2023b, November 2). Minesweeper (video game). Wikipedia. https://en.wikipedia.org/wiki/Minesweeper_(video_game)
[5] Project, F. V. G. The history of Minesweeper. FreeMinesweeper.org. https://freeminesweeper.org/minesweeper-history.php
[6] Refactoring.Guru. Code smells. https://refactoring.guru/refactoring/smells
[7] Refactoring home page. https://www.refactoring.com/
[8] PMD. https://pmd.github.io/

Appendix

This code was written by Mitchell Sayer on 5/9/15. This is the original version of the code.

```
    import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;

/**
 * Created by Mitchell Sayer on 5/9/15.
 */
public class Minesweeper implements ActionListener {

    JFrame frame = new JFrame("Minesweeper");
```

```
    JButton reset = new JButton("Reset");
    JButton solve = new JButton("Solve");
    JToggleButton[][] buttons = new JToggleButton[20][20];
    int[][] counts = new int [20][20];
    Container grid = new Container();
    boolean lost = false;
    boolean firstLost = true;
    boolean canSolve = false;
    int mineCount = 30;
    final int MINE = 10;
    ArrayList<Integer> marked = new ArrayList<>();

    public static void main(String[] args)
    {
        new Minesweeper();
    }
    //lets gooo

    public Minesweeper()
    {
        frame.setSize(900, 900);
        frame.setLayout(new BorderLayout());
        frame.add(reset, BorderLayout.NORTH);
        frame.add(solve, BorderLayout.SOUTH);
        reset.addActionListener(this);
        solve.addActionListener(this);
        grid.setLayout(new GridLayout(20, 20));
        for (int r = 0; r < buttons.length; r++)
            for (int c = 0; c < buttons[0].length; c++) {
                buttons[r][c] = new JToggleButton();
                buttons[r][c].addActionListener(this);
                grid.add(buttons[r][c]);
                buttons[r][c].setSize(frame.getWidth() / 20, frame.getHeight() / 22);
            }
        frame.add(grid,BorderLayout.CENTER);
        addRandomMines();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public void addRandomMines()
    {
        ArrayList<Integer> mineList = new ArrayList<>();
        for (int x = 0; x < counts.length; x++) {
            for (int y = 0; y < counts[0].length; y++){
                mineList.add((x*100)+y);
            }
        }
        counts = new int[20][20];
        for (int i = 0; i < mineCount; i++) {
```

```
            int choice = (int)(Math.random()*mineList.size());
            counts[mineList.get(choice)/100][mineList.get(choice)%100] = MINE;
            mineList.remove(choice);
        }


        for (int x = 0; x < counts.length; x++) {
            for (int y = 0; y < counts[0].length; y++){
                if (counts[x][y]!=MINE) {
                    int mineCount = 0;
                    if (x > 0 && y > 0 && counts[x - 1][y - 1] == MINE)
                        mineCount++;
                    if (y > 0 && counts[x][y - 1] == MINE)
                        mineCount++;
                    if (x > 0 && counts[x - 1][y] == MINE)
                        mineCount++;
                    if (x < counts.length - 1 && counts[x + 1][y] == MINE)
                        mineCount++;
                    if (y < counts.length - 1 && counts[x][y + 1] == MINE)
                        mineCount++;
                    if (x < counts.length - 1 && y < counts.length - 1 && counts[x + 1][y + 1] == MINE)
                        mineCount++;
                    if (x > 0 && y < counts.length - 1 && counts[x - 1][y + 1] == MINE)
                        mineCount++;
                    if (x < counts.length - 1 && y > 0 && counts[x + 1][y - 1] == MINE)
                        mineCount++;
                    counts[x][y] = mineCount;
                }
            }
        }
    }

    public void showTile(int r, int c)
    {
        if (counts[r][c] == 0) {
            buttons[r][c].setText("");
            buttons[r][c].setSelected(true);
        }
        else if (counts[r][c]==MINE) {
            buttons[r][c].setForeground(Color.red);
            buttons[r][c].setText("X");
            buttons[r][c].setSelected(true);
        }
        else {
            buttons[r][c].setText(counts[r][c] + "");
```

```
            if (counts[r][c]==1)
                buttons[r][c].setForeground(Color.blue);
            else if (counts[r][c]==2)
                buttons[r][c].setForeground(Color.magenta);
            else if (counts[r][c]==3)
                buttons[r][c].setForeground(Color.green);
            buttons[r][c].setSelected(true);
        }
    }

    public void clearEmpty(int row, int col) {
        for ( int r = row - 1; r <= row + 1; r++ ) {
            for (int c = col - 1; c <= col + 1; c++) {
                if (r >= 0 && r < counts.length && c >= 0 && c < counts[0].length) {
                    if (!buttons[r][c].isSelected()) {
                        showTile(r, c);
                        if (counts[r][c] == 0)
                            clearEmpty(r, c);
                    }
                }
            }
        }
    }

    public boolean checkLose() {
        boolean won = true;
        for (int x = 0; x < buttons.length; x++) {
            for (int y = 0; y < buttons[0].length; y++){
                if (counts[x][y]==MINE&&buttons[x][y].isSelected())
                    won = false;
            }
        }
        if (!won) {
            for (int x = 0; x < buttons.length; x++) {
                for (int y = 0; y < buttons[0].length; y++){
                    buttons[x][y].setEnabled(false);
                    if (counts[x][y]==MINE) {
                        buttons[x][y].setEnabled(true);
                        showTile(x,y);
                    }
                }
            }
            return true;
        }
        else
            return false;
    }

    public boolean checkWin() {
        boolean won = true;
```

```
        for (int x = 0; x < buttons.length; x++) {
            for (int y = 0; y < buttons[0].length; y++){
                if (counts[x][y]!=MINE&&!buttons[x][y].isSelected())
                    won = false;
            }
        }
        if (won&&!lost) {
            for (int x = 0; x < buttons.length; x++) {
                for (int y = 0; y < buttons[0].length; y++){
                    buttons[x][y].setEnabled(false);
                }
            }
            return true;
        }
        else
            return false;
    }

    public int surroundingClosed(int x, int y) {
        int count = 0;
        for ( int r = x - 1; r <= x + 1; r++ ) {
            for (int c = y - 1; c <= y + 1; c++) {
                if (r >= 0 && r < counts.length && c >= 0 && c < counts
[0].length) {
                    if (!buttons[r][c].isSelected())
                        count++;
                }
            }
        }
        return count;
    }

    public void markItem(int x, int y, int n) {
        int count = 0;
        for ( int r = x - 1; r <= x + 1; r++ ) {
            for (int c = y - 1; c <= y + 1; c++) {
                if (r >= 0 && r < counts.length && c >= 0 && c < counts
[0].length) {
                    if (!buttons[r][c].isSelected()) {
                        if (count>n)
                            return;
                        else {
                            if (!marked.contains(r*100+c)) {
                                marked.add(r * 100 + c);
                                count++;
                            }
                        }
                    }
                }
            }
        }
```

```
    }

    public int knownMineCount(int x,int y) {
        int count = 0;
        for ( int r = x - 1; r <= x + 1; r++ ) {
            for (int c = y - 1; c <= y + 1; c++) {
                if (r >= 0 && r < counts.length && c >= 0 && c < counts[0].length) {
                    int arrayVal = r*100+c;
                    if (marked.contains(arrayVal))
                        count++;
                }
            }
        }
        return count;
    }

    public int openNonMines(int x, int y) {
        int count = 0;
        for ( int r = x - 1; r <= x + 1; r++ ) {
            for (int c = y - 1; c <= y + 1; c++) {
                if (r >= 0 && r < counts.length && c >= 0 && c < counts[0].length) {
                    int arrayVal = r*100+c;
                    if (!marked.contains(arrayVal)) {
                        showTile(r,c);
                        count++;
                    }
                }
            }
        }
        return count;
    }

    public boolean solveGame() {
        int count = 0;
        int dif=0;
        for (int x = 0; x < counts.length; x++) {
            for (int y = 0; y < counts[0].length; y++) {
                if (buttons[x][y].isSelected()) {
                    int surround = surroundingClosed(x, y);
                    int curCount = counts[x][y];
                    int kmc = knownMineCount(x, y);
                    if (surround == curCount)
                        markItem(x, y, curCount);
                    if (surround > curCount && kmc == curCount&&!marked.contains(x*100+y)) {
                        int cOld = count;
                        count+=openNonMines(x, y);
                        dif=count-cOld;
                    }
```

```
                }
            }
        }
        if (marked.size()==30&&checkWin()) {
            canSolve = true;
            return false;
        }
        if (dif>0)
            return true;
        else
            return false;

    }

    @Override
    public void actionPerformed(ActionEvent event) {
        if (event.getSource().equals(reset)) {
            for (int r = 0; r < buttons.length; r++) {
                for (int c = 0; c < buttons[0].length; c++) {
                    buttons[r][c].setEnabled(true);
                    buttons[r][c].setSelected(false);
                    buttons[r][c].setText("");
                }
            }
            canSolve=false;
            marked.clear();
            addRandomMines();
        }
        else if (event.getSource().equals(solve)) {
            while (solveGame())
                canSolve=false;
            if (canSolve||checkWin()) {
                JOptionPane.showMessageDialog(frame, "The computer wins dumbass");
                for (int x = 0; x < buttons.length; x++) {
                    for (int y = 0; y < buttons[0].length; y++){
                        buttons[x][y].setEnabled(false);
                    }
                }
            }
            else if (!canSolve) {
                JOptionPane.showMessageDialog(frame, "Uh oh, you are gonna have to guess, dumbass");
            }
        }
        else {
            for (int r = 0; r < buttons.length; r++) {
                for (int c = 0; c < buttons[0].length; c++) {
                    if (event.getSource().equals(buttons[r][c])) {
                        if (counts[r][c] == 0) {
                            clearEmpty(r,c);
```

```
                        }
                        showTile(r, c);
                        if (checkWin()) {
                            JOptionPane.showMessageDialog(frame, "YOU
    WIN DUMBASS");
                            return;
                        }
                        else if (checkLose()) {
                            JOptionPane.showMessageDialog(frame, "you
    lose dumbass");
                            return;
                        }
                    }
                }
            }
        }
    }
}
```

The rule for providing he Cyclomatic complexity using PMD is as following:

```
    <?xml version="1.0"?>
<ruleset name="Custom Ruleset"
         xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
    https://pmd.sourceforge.io/ruleset_2_0_0.xsd">
    <description>
        This is a custom ruleset to check cyclomatic complexity in Java
    code.
    </description>

    <!-- Cyclomatic Complexity rule -->
    <rule ref="category/java/design.xml/CyclomaticComplexity">
        <properties>
            <property name="classReportLevel" value="80"/>
            <property name="methodReportLevel" value="1"/>
            <property name="cycloOptions" value=""/>
        </properties>
    </rule>
</ruleset>
```

The following installs PMD on a MAC computer. The lines should be entered in the terminal:

```
$ cd $HOME
$ curl -OL https://github.com/pmd/pmd/releases/download/pmd_releases%2F7
    .0.0-rc4/pmd-dist-7.0.0-rc4-bin.zip
$ unzip pmd-dist-7.0.0-rc4-bin.zip
$ alias pmd="$HOME/pmd-bin-7.0.0-rc4/bin/pmd"
```

The following are the cyclomatic complexities of the refactored code version 2. As it can be seen the methods have less cyclomatic complexities. Nevertheless, we did not achieve our goal in having a maximum of 10 for each method as SolveGame() has 11.

Cyclomatic for class MineSweeper2: *The class 'MineSweeper2' has a total cyclomatic complexity of 127 (highest 11).* File: MineSweeper2.java:14

Cyclomatic for method main(String[]): *The method 'main(String[])' has a cyclomatic complexity of 3.* File: MineSweeper2.java:30

Cyclomatic for method runPerformanceTest(int): *The method 'runPerformanceTest(int)' has a cyclomatic complexity of 1.* File: MineSweeper2.java:51

Cyclomatic for constructor MineSweeper2(int): *The constructor 'MineSweeper2(int)' has a cyclomatic complexity of 3.* File: MineSweeper2.java:68

Cyclomatic for method addRandomMines(): *The method 'addRandomMines()' has a cyclomatic complexity of 1.* File: MineSweeper2.java:91

Cyclomatic for method initializeMineList(): *The method 'initializeMineList()' has a cyclomatic complexity of 3.* File: MineSweeper2.java:97

Cyclomatic for method placeMines(ArrayList<Integer>): *The method 'placeMines(ArrayList¡Integer¿)' has a cyclomatic complexity of 2.* File: MineSweeper2.java:107

Cyclomatic for method calculateAdjacentMines(): *The method 'calculateAdjacentMines()' has a cyclomatic complexity of 4.* File: MineSweeper2.java:118

Cyclomatic for method getAdjacentMineCount(int, int): *The method 'getAdjacentMineCount(int, int)' has a cyclomatic complexity of 10.* File: MineSweeper2.java:128

Cyclomatic for method showTile(int, int): *The method 'showTile(int, int)' has a cyclomatic complexity of 6.* File: MineSweeper2.java:145

Cyclomatic for method clearEmpty(int, int): *The method 'clearEmpty(int, int)' has a cyclomatic complexity of 9.* File: MineSweeper2.java:168

Cyclomatic for method checkLose(): *The method 'checkLose()' has a cyclomatic complexity of 9.* File: MineSweeper2.java:182

Cyclomatic for method checkWin(): *The method 'checkWin()' has a cyclomatic complexity of 9.* File: MineSweeper2.java:206

Cyclomatic for method surroundingClosed(int, int): *The method 'surroundingClosed(int, int)' has a cyclomatic complexity of 8.* File: MineSweeper2.java:226

Cyclomatic for method markItem(int, int, int): *The method 'markItem(int, int, int)' has a cyclomatic complexity of 10.* File: MineSweeper2.java:239

Cyclomatic for method knownMineCount(int, int): *The method 'knownMineCount(int, int)' has a cyclomatic complexity of 8.* File: MineSweeper2.java:259

Cyclomatic for method openNonMines(int, int): *The method 'openNonMines(int, int)' has a cyclomatic complexity of 8.* File: MineSweeper2.java:273

Cyclomatic for method solveGame(): *The method 'solveGame()' has a cyclomatic complexity of 11.* File: MineSweeper2.java:289

Cyclomatic for method actionPerformed(ActionEvent): *The method 'actionPerformed(ActionEvent)' has a cyclomatic complexity of 3.* File: MineSweeper2.java:320

`Cyclomatic for method resetGame():` *The method 'resetGame()' has a cyclomatic complexity of 3.* File: MineSweeper2.java:332
`Cyclomatic for method solveGame2():` *The method 'solveGame2()' has a cyclomatic complexity of 5.* File: MineSweeper2.java:345
`Cyclomatic for method handleTileClick(ActionEvent):` *The method 'handleTileClick(ActionEvent)' has a cyclomatic complexity of 4.* File: MineSweeper2.java:357
`Cyclomatic for method processTileClick(int, int):` *The method 'processTileClick(int, int)' has a cyclomatic complexity of 4.* File: MineSweeper2.java:368
`Cyclomatic for method disableAllButtons():` *The method 'disableAllButtons()' has a cyclomatic complexity of 3.* File: MineSweeper2.java:380