CE881 Assignment 2

# Assignment 2 Report

Sudoku Puzzle & Solver App

Student ID: FK18726
Reg Number: 1804162
3-31-2022

Page Limit: 5 Pages of Content

Pages: 5 Pages (5 pages of content + title page, table of contents, figures, and references)

# Contents

Introduction ..... 1
App Description ..... 1
Design & Features ..... 1
Full App Navigation & Lifecycle ..... 1
Implemented Features ..... 1
Additions to the Prototype ..... 2
Implementation ..... 2
Key & Interesting Parts ..... 2
Device Rotation ..... 2
Splash Screen ..... 3
Greeting Prefix ..... 3
Daily Streak ..... 3
Puzzle Generation ..... 3
Saving & Loading of Grid State ..... 4
Puzzle Solution Generation (Solver) ..... 4
UML Class Diagram ..... 4
Testing ..... 4
MainActivity ..... 5
NewGameActivity ..... 5
GameActivity ..... 5
SolverActivity ..... 5
Conclusion ..... 5
Reflection ..... 5
Future Work ..... 5
Figures ..... 6
Tables ..... 6
Existing Puzzle & Solution ..... 8
Screenshots (App Processes) ..... 8
Screenshots (Landscape – Orientations) ..... 10
Code Snippets ..... 10
UML Diagram ..... 14
References ..... 15

Reg Number: 1804162

# Introduction

## App Description

Sudoku is a popular puzzle game, requiring the player to form a full 9x9 grid with numbers ranging from 1 to 9; without these numbers repeating in a row, column, or sub-box. This app serves to provide new or existing sudoku players with two features: a puzzle generator, and a puzzle solver, of which are most commonly seen in separate applications. With this, a puzzle can be generated to solve with a specified difficulty, and existing puzzles can be entered into the solver in order to generate a solution.

# Design & Features

## Full App Navigation & Lifecycle

When the app is first launched, the 'MainActivity' activity is launched; presenting the user with a splash screen (Figure 2) for a few seconds before redirecting the content view to the main menu layout (Figure 4). If this is the first time the user has opened the app, then they will be prompted with a dialog fragment to enter their first name (Figure 3). This is to allow the user to be properly greeted by the app, as well as using the current time of day to get the correct prefix for the greeting.

Opening the options menu (Figure 5) provides the user with two options: opening the 'AboutActivity' activity, or changing their saved name. Selecting 'Change Name' prompts the user with a dialog fragment to enter their first name as before (Figure 6), whilst selecting 'About' leads to the 'AboutActivity' being started, which provides information on the different features of the application and the basic rules of Sudoku (Figure 7).

From the main menu, the user can select 3 options: 'Continue Game', 'New Game', and 'Solver'. Selecting 'Solver' will result in the 'SolverActivity' activity being started (Figure 8). The user will be presented with an empty grid, and buttons in which to interact with the grid. From here, the user can enter a valid puzzle that they wish to be solved (Figure 9), before pressing the 'Solve' button to generate a solution to their puzzle (Figure 10). If the puzzle entered by the user is invalid, then the user is warned and no solution is generated (Figure 11).

Selecting 'New Game' will result in an alert dialog appearing, prompting the user to confirm if they wish to start a new game and overwrite any previous saved puzzle (Figure 12). Confirming this prompt results in the 'NewGameActivity' activity being started. When this activity is started, the user will be prompted with another alert dialog to select a difficulty for the generated puzzle (Figure 13). After a difficulty has been selected and confirmed, a puzzle will be generated by the activity (Figure 14). As with the 'SolverActivity' activity, the user will be presented with a grid, and buttons with which to interact with the grid. After filling the grid, the user should press the 'Check' button. If the entered solution is incorrect, the user will be warned with an alert dialog that the solution is incorrect (Figure 15). If the solution is correct, then the user will be notified with an alert dialog, and prompted to start a new puzzle (Figure 16). If the user has provided a correct solution 24 hours after their last saved attempt, then their daily streak will also be incremented.

Selecting 'Continue Game' will result in the 'GameActivity' activity being started. This functions nearly identical to the 'NewGameActivity' activity, however instead of prompting the user to select a difficulty, it instead loads a previously saved puzzle. If no puzzle has been started (or if the saved grid is empty), then the user will be redirected to the 'NewGameActivity' activity to start a new puzzle.

Pressing the 'Back' button from the main menu will prompt the user with an alert dialog to confirm that they wish to leave the application (Figure 17). If the user confirms this choice, then the app will finish.

## Implemented Features

At the current state of my app, all of my essential features from my prototype have been implemented, with only 1 desirable feature not yet being implemented; the theme selection. The main cause of this not being implemented was due to time constraints, as my plan from the prototype scheduled the tasks to be completed to ensure that the main features of the application were working

fully. As this is only a desirable feature, it was not a requirement that this feature be implemented, however this provides an opportunity for future development; which will be discussed later in this report. The full grid of planned key features and their current status in the app can be seen in Table 1.

### Additions to the Prototype

During my plan from the prototype of this app, there were four main tasks and features that I had planned to finish implementing:

- Finishing the grid view:
  - Allowing numbers to be displayed in the grid cells.
- Full puzzle generation mode:
  - Development of the puzzle generation algorithm and validity checks to generate a valid puzzle.
- Full solver mode:
  - Development of the puzzle solving algorithm to solve an entered puzzle.
- App layout and appearance improvements:
  - Implementation of a proper main menu.
  - Revisit themes and existing layouts.

At the current state of my application, all of these features have successfully been implemented and tested fully (test cases will be discussed in the next section). The time taken to complete these tasks stated within my original plan was realistic, giving me time after finishing the implementation of these features to clean up the rest of my application. In addition to these planned features, I had also made landscape versions of each layout; ensuring rotation is handled gracefully.

## Implementation

This section will discuss the implementation of key and interesting parts of the application, and why some decision have been made. It will also provide a UML class diagram of the project structure; allowing for easy visualisation of what classes extend higher classes, and where inner classes are created.

### Key & Interesting Parts

### Device Rotation

To deal with device rotation, the first lines of code that I had to implement were in relation to the ‘AndroidManifest.xml’ file, and how it manages configuration changes. This allows all changes in orientation, screen size, and other configuration types to be dealt with automatically; preventing the app or activity from restarting or errors from occurring. However, whilst this allowed the app to remain functional on orientation change, this alone was far from a ‘good’ solution.

After implementing the previously mentioned lines of code, I created landscape alternatives of each of the layouts present in my project. The exceptions to this was for the splash screen, and for the ‘AboutActivity’ layouts. This is because these layouts are fairly basic, only being used to display a single item that is either centered, or takes up the full screen. The landscape variations of the other layouts were made to improve the usability of the application by rearranging parts of the layout to make good use of the full screen space. These landscape variations can be seen in the following figures: Figure 18, Figure 19, Figure 20, and Figure 21.

Although the configuration changes are handled by the application automatically, we need to be able to manually update the content view to use the new layouts. In order to do this, the ‘onConfigurationChanged()’ method had to be overridden, to allow for the layouts to be changed depending on if the orientation is portrait or landscape. However, this also poses a new issue. Because the layout instance has been changed, the previously referenced views and layout objects are still referenced using the old layout. These also need to be updated after the new layout has been set; to ensure that the parts and views are displayed correctly.

The code relevant to this implementation can be seen in Snippet 1 and in Snippet 2.

## Splash Screen

The splash screen implementation is fairly simple. When the app first loads, the 'MainActivity' activity will use the splash screen layout. An instance of an 'AsyncTask()' class is then used in order to wait for 5 seconds, before changing the content view to that of the main menu layout.

Using the 'AsyncTask()' class allows for this to be managed within a separate thread. With this, the new thread can be paused to wait for a set amount of time, without the main thread becoming unresponsive. Whilst this implementation currently waits a set amount of time, for a larger application it could be set to wait until the application has fully loaded to ensure that it is only present for as long as it needs to be.

The code relevant to this implementation can be seen in Snippet 3.

## Greeting Prefix

The greeting prefix forms the first part of the greeting that the user will see in the main menu, and should accurately reflect the time of day. In order to accomplish this, the first step was to retrieve an instance of a calendar. A 'Date' variable containing the date and time set within the device can then be used to set the time for this calendar instance, allowing us to access the current hour of the day (using a 24-hour format).

From here, it's simply a case of a few if statements. Morning is generally considered to be the hours that are less than 12:00pm, and afternoon is generally considered to be the hours that are between 12:00pm and 6:00pm. Outside of these ranges should be considered evening, as 'Good Night' is not considered a greeting. From these ranges, the following prefixes can be obtained: 'Good Morning', 'Good Afternoon', and 'Good Evening'.

The code relevant to this implementation can be seen in Snippet 4.

## Daily Streak

The daily streak is a record of how many consecutive days that a puzzle has been completed on. A similar approach as to what was taken for retrieving the correct prefix is also used here.

The first step is to load the previously saved day of the year, as well as the current streak. These are both saved together once the streak is updated, as it allows for easy comparison to see if the streak should be updated after a puzzle is completed. After these are retrieved, the current day of the year needs to be retrieved. The current day of the year is retrieved by repeating the first couple of steps for retrieving the prefix for the greeting, as described above. However, instead of then retrieving the time of day, we retrieve the day of the year (in a 365-day format).

The first check done is to see if more than two days have passed since the last puzzle was completed. We check to see if the current day value is larger than the saved day of the year, however is not only 1 day larger. If this condition is met, then more than a day has passed since the last puzzle, so the streak is reset to 0 and the current day is saved. If this condition is not met, then the streak and the last saved day remain unchanged. When a puzzle has been successfully completed, the second check is performed. We first check to see if the streak is 0, indicating that no streak currently exists. If so, then the streak is incremented and the day is saved. The next check is to see if the current day is 1 higher than the saved day. If so, then again the streak is updated and the day is saved. This ensures that the streak does not increment if a user completes more than one puzzle per day.

The code relevant to this implementation can be seen in Snippet 5.

## Puzzle Generation

The puzzle generation algorithm is the 'main' part of this application. It allows for a valid full grid to be created, before copying this grid and removing values randomly in order to create the empty cells for the user to enter numbers into. This grid is then again copied. Having 3 grids allows one to store the full solution for the puzzle, and for 1 to store the basic state of the puzzle if the user wishes to reset their progress.

The puzzle generation first starts with filling some values randomly in the grid, as otherwise each puzzle would be identical. In order to accomplish this, a number from 1 to 9 is randomly assigned to the top left, middle, and bottom right sub-boxes. This is done, as these sub-boxes do not interact with each other in any way; resulting in a good starting point for valid solutions to be made.

After this, the rest of the solution is generated by recursively assigning valid numbers from 1 to 9 into the empty cells. Numbers are checked to be valid, by seeing if they are already present in the current row, column, and sub-box. If the value is invalid, then the next number is checked. If no numbers are valid, then the previous cell is incremented, and the process repeats. If the number is valid, then it is assigned to that cell and the next cell is then checked. Once every cell has been filled, then the recursion ends. In order to remove cells for the user to enter numbers into, percentages relative to the selected difficulty are used as the 'chance' that a cell in the grid will be removed. Using a random number between 0 and 1, if it is below the threshold relative to the difficulty, the cell value is removed.

The code relevant to this implementation can be seen in Snippet 6 and in Snippet 7.

### Saving & Loading of Grid State

Saving and loading of the grid state is performed every time the state of the grid changes, to ensure that there is no loss of progress. As with the daily streak and entered name, the grid state is saved into the player preferences files, ensuring that it is persistent between closing and opening the application. However, as this does not support saving of 2D arrays, the values of the grids (solution, basic puzzle, and the interactive grid) had to be iterated through and stored as a string. This both allows player preferences to be used, as well as being more efficient than storing a raw 2D array. To retrieve the values from this string, the reverse process had to be used; iterating through a 9x9 2D array, and assigning string values in order by keeping a count of the current index of the character in the string.

The code relevant to this implementation can be seen in Snippet 8 and in Snippet 9.

### Puzzle Solution Generation (Solver)

The algorithm used to generate a solution to the entered puzzle is nearly identical to that of generating a puzzle for the user to complete, as that model works by first generating a full solution, before removing cells. Because of this, we can add some simple changes to ensure that the generated solution is valid.

The first step is to ensure that the user's entered puzzle is valid. This is done in a similar way as to how the empty cells are checked recursively, however due to these values already being present in the grid, there were two options: making a new grid and then checking to see if that grid has entered values before assigning values recursively, or copying the methods to check to see if a number is valid, and then slightly changing them to instead count the occurrences of the number in that row, column, or sub-box (which shouldn't exceed 1). As the former approach involved instantiating new 2D arrays, I opted to use the latter.

If the entered solution is deemed valid, then the previous solution generation algorithm is performed; recursively assigning values to cells, whilst skipping cells that have been entered by the user. As this however could take longer, and to avoid issues such as infinite loops from crashing the application, this process is all performed within an instance of an 'AsyncTask()' class.

I will not be providing snippets for this algorithm, due to it being nearly identical to that of the puzzle generation.

### UML Class Diagram

The UML diagram of the full project can be seen in Figure 22.This diagram has been generated using the 'UML Generator' plugin for Android Studio, and shows how the different classes (outer and inner) interact. This figure is also submitted in the same folder alongside this report, for easier viewing.

## Testing

In order to test my app, I performed mostly unit tests. This involved testing each part of each activity, to ensure it responded to user inputs as expected. I have also made some use of integration

testing, however due to each activity functioning completely separate from each other this was not much more than ensuring that the intents launch the corresponding activity correctly. In the following sections I will outline the test cases performed for parts of each activity.

### MainActivity

The test cases for this activity can be found in Table 2.

### NewGameActivity

The test cases for this activity can be found in Table 3.

### GameActivity

As this activity and 'NewGameActivity' activity function the same, and use the same view and model, most test cases for 'NewGameActivity' had also needed to be tested in this activity, so will not be listed again.

The test cases for this activity can be found in table Table 4.

### SolverActivity

In order to test that the solution created is valid, I tested this activity by using an existing sudoku puzzle and solution [1], which can be seen in Figure 1. From this, it could be seen that my solver was able to generate the exact solution that was provided alongside the puzzle.

The test cases for this activity can be found in Table 5.

## Conclusion

### Reflection

In reflection of my project, I believe that I have used my time well, and that the app is well-developed. I have managed to include every planned feature of my app that was set from the prototype; which as can be seen from my test cases, work without issue. The theme and appearance of the app is consistent between activities, and an efficient use of multithreading is also used to ensure that the app does not crash when waiting for a long process to finish (such as the puzzle solver). I have also managed to cleanly deal with device orientation changes; with the layout changing to accommodate the device's orientation, whilst remaining consistent in style.

### Future Work

For future development of my application, one of the main aspects I wish to improve on and develop further is the layouts of the activities. Whilst every item is accessible and allows for the app to be used as intended, spending more time on improving the overall aesthetic of the app would aid to make it more pleasing to use. With this, I could also implement multiple themes: of which the user can select. This would help make the app more personal, as well as potentially provide some in-app purchases.

Another path I would like to take my existing app would be the implementation of different game modes. Currently, my app only implements the basic rules of Sudoku, however there are many variations of puzzles that could be implemented. To achieve this, the same basic functionality of the puzzle generator will present, with the user still requiring to input numbers into empty cells. However, when generating a puzzle, extra restrictions will be put in place. Depending on the type of restriction, extra checks will have to be made along with the existing row, column, and sub-box repetition checks.

Due to these restrictions removing the number of possible numbers in some squares, they actually make solving the puzzle easier. For this reason, more cells will have to be made blank for the user to solve when a specific type of sudoku puzzle is generated.

# Figures

All images will also be included in folders alongside this report, so that they can be more easily seen if required.

## Tables

| Feature | Brief Description | Type | Status |
|---|---|---|---|
| **Puzzle Generation** | Generates a valid puzzle for the user to solve. | Essential | Implemented |
| **Puzzle Solver** | Generates a solution to a valid entered puzzle. | Essential | Implemented |
| **Difficulty Selection** | Allows the user to select a relative amount of blank cells in the generated puzzle | Essential | Implemented |
| **About Page** | Provides information about the app modes, as well as basic Sudoku rules.. | Essential | Implemented |
| **Cell Highlighting** | Highlights the current row and column of the selected cell. | Essential | Implemented |
| **Button-Number Entering** | Allows for the layouts to remain consistent between devices. | Essential | Implemented |
| **Exit App Confirmation** | Prevents the user from accidentally exiting the application. | Essential | Implemented |
| **Daily Streak** | Provides incentive for the user to return to the application every day. | Desirable | Implemented |
| **Personal Greeting** | Provides personal touch to the application. | Desirable | Implemented |
| **Name Change** | Allows the user to alter their saved name. | Essential (If Personal Greeting is Implemented) | Implemented |
| **Theme Selection** | Allows the user to change the theme to their preference. | Desirable | Not Implemented |

*Table 1: Table of features*

| Test Case | Expected | Evaluation |
|---|---|---|
| **App is opened** | Splash screen should be displayed for some seconds, before changing to the main menu | Pass |
| **'Back' button is pressed** | User should be prompted to confirm that they wish to leave the app | Pass |
| **App opened for first time** | User should be prompted to enter their name | Pass |
| **Name is present** | A correct prefix using the time of day should be given to greet the user | pass |
| **'Change Name' is selected** | User should be prompted to enter their name | Pass |
| **'About' is selected** | About activity is started | Pass |
| **'Solver' is selected** | Solver activity is started | Pass |
| **'New Game is selected** | Confirmation prompt should appear | Pass |
| **User confirms 'New Game' prompt** | New Game activity is started | Pass |
| **'Continue Game is selected** | Continue Game activity is started | Pass |

*Table 2: 'MainActivity' test cases*

| Test Case | Expected | Evaluation |
|---|---|---|
| **Activity is started** | User prompted to select a difficulty, and current streak should be loaded | Pass |
| **User has not played in 2 days or longer** | Daily streak is set to 0 | Pass |
| **Difficulty is selected** | Grid is generated with a relative amount of empty cells | Pass |
| **A number is pressed without selecting a cell** | Nothing happens | Pass |
| **A number is pressed when selecting a cell** | The number is entered into the cell | Pass |
| **'Reset' is selected** | Grid is reset | Pass |
| **'Check' is selected with an incorrect solution** | User is warned that the solution is incorrect | Pass |
| **'Check' is selected with a correct solution** | User is prompted to start a new game | Pass |
| **Correct solution is submitted at least 1 day after the previously saved streak** | User streak is incremented by 1 | Pass |

*Table 3: 'NewGameActivity' test cases*

| Test Case | Expected | Evaluation |
|---|---|---|
| **No Saved Grid** | New Game activity should be launched | Pass |
| **Saved Grid** | Puzzle should be loaded successfully | Pass |

*Table 4: 'GameActivity' test cases*

| Test Case | Expected | Evaluation |
|---|---|---|
| **A number is pressed without selecting a cell** | Nothing happens | Pass |
| **A number is pressed when selecting a cell** | The number is entered into the cell | Pass |
| **'Reset' is selected** | Grid is reset | Pass |
| **'Solve is selected with a valid grid** | Puzzle is solved | Pass |
| **'Solve is selected with an invalid grid** | User is warned that the puzzle is invalid | Pass |

*Table 5: 'SolverActivity' test cases*

Reg Number: 1804162

## Existing Puzzle & Solution

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | 2 | | |
| | 8 | | | | 7 | | 9 | |
| 6 | | 2 | | | | 5 | | |
| | 7 | | | 6 | | | | |
| | | | 9 | | 1 | | | |
| | | | | 2 | | | 4 | |
| | | 5 | | | | 6 | | 3 |
| | 9 | | 4 | | | | 7 | |
| | | 6 | | | | | | |

(a) Sudoku Puzzle

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 9 | 5 | 7 | 6 | 1 | 3 | 2 | 8 | 4 |
| 4 | 8 | 3 | 2 | 5 | 7 | 1 | 9 | 6 |
| 6 | 1 | 2 | 8 | 4 | 9 | 5 | 3 | 7 |
| 1 | 7 | 8 | 3 | 6 | 4 | 9 | 5 | 2 |
| 5 | 2 | 4 | 9 | 7 | 1 | 3 | 6 | 8 |
| 3 | 6 | 9 | 5 | 2 | 8 | 7 | 4 | 1 |
| 8 | 4 | 5 | 7 | 9 | 2 | 6 | 1 | 3 |
| 2 | 9 | 1 | 4 | 3 | 6 | 8 | 7 | 5 |
| 7 | 3 | 6 | 1 | 8 | 5 | 4 | 2 | 9 |

(b) Solution

*Figure 1: Existing puzzle & solution used for testing of solver [1]*

## Screenshots (App Processes)

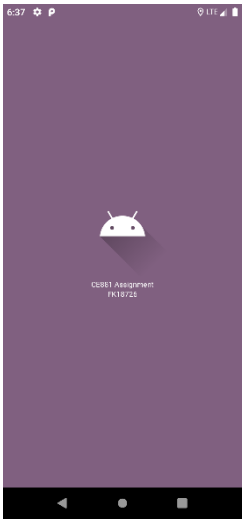


*Figure 2: Splash screen*

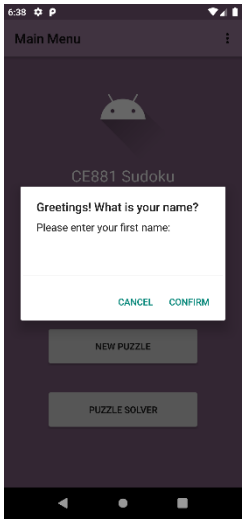


*Figure 3: First greetings*

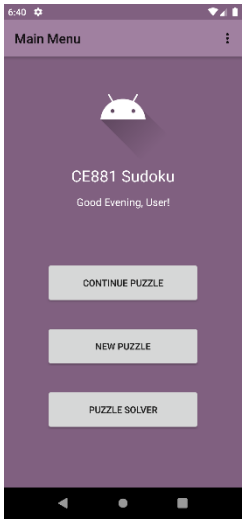


*Figure 4: Main menu*

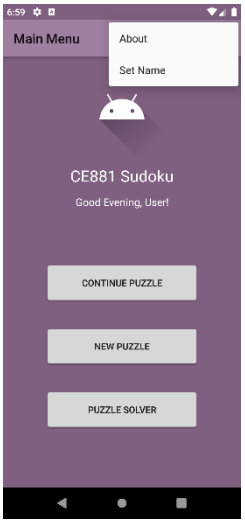


*Figure 5: Options Menu*

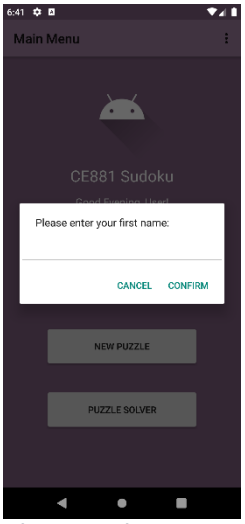


Figure 6: Change name

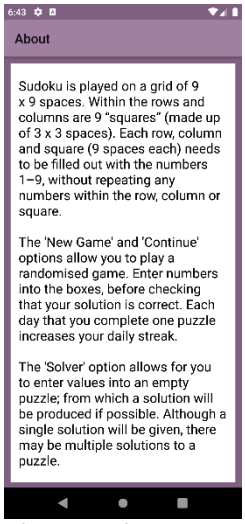


Figure 7: About

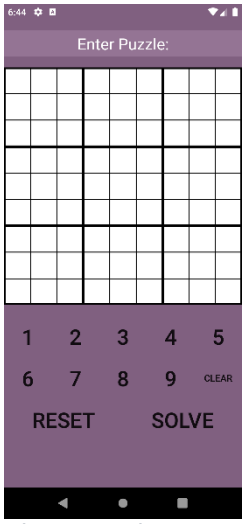


Figure 8: Solver

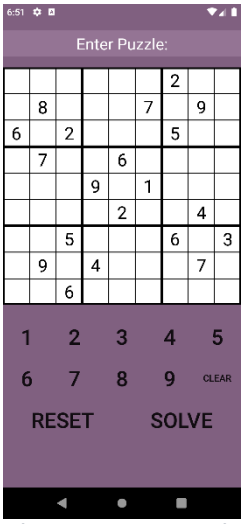


Figure 9: Enter puzzle

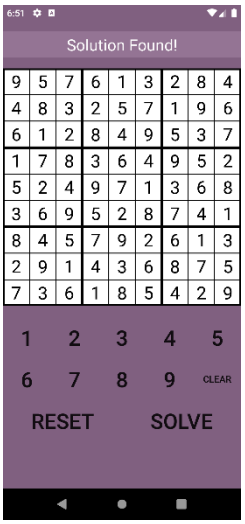


Figure 10: Puzzle solution

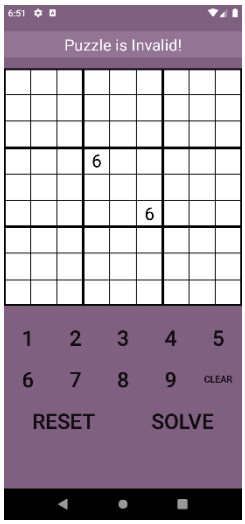


Figure 11: Invalid puzzle

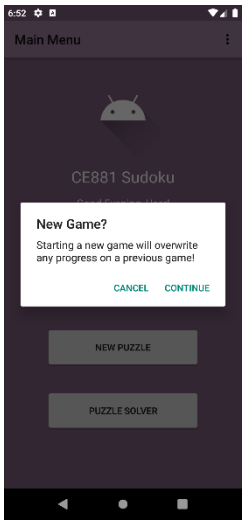


Figure 12: New game

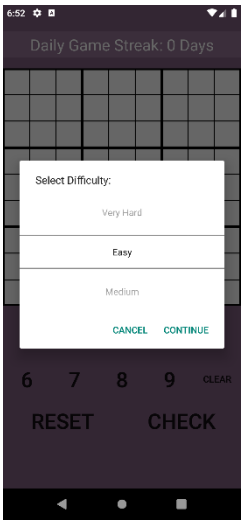


Figure 13: Difficulty select

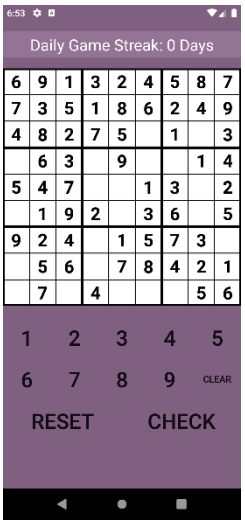


Figure 14: Generated puzzle

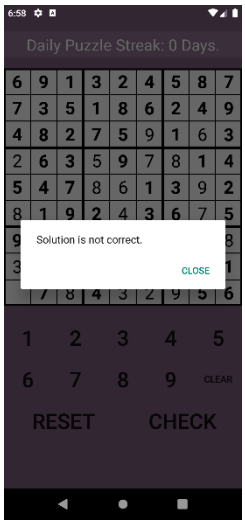


Figure 15: Incorrect solution

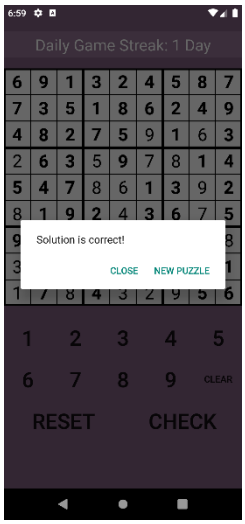


Figure 16: Correct solution

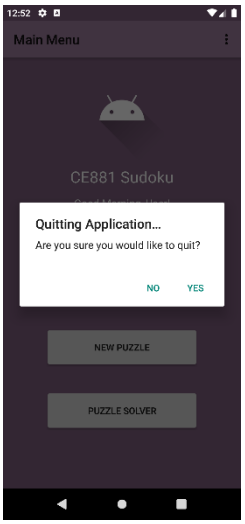


Figure 17: Quit application

## Screenshots (Landscape – Orientations)

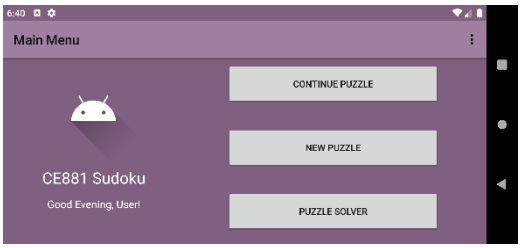


*Figure 18: Main menu (landscape)*

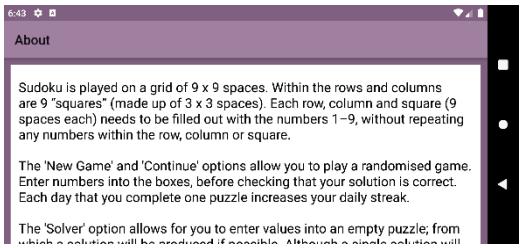


*Figure 19: About (landscape)*

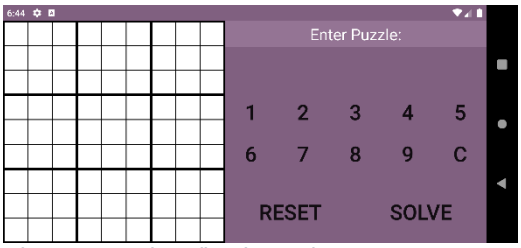


*Figure 20: Solver (landscape)*


*Figure 21: Generated puzzle (landscape)*

## Code Snippets

```
android:configChanges="keyboardHidden|orientation|screenLayout|uiMode|screenSize"
```

*Snippet 1: Dealing with configuration changes in AndroidManifest.xml*

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
	super.onConfigurationChanged(newConfig);
	if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
		setContentView(R.layout.activity_game);
		gameView = findViewById(R.id.GameView);
		game = gameView.getGame();
		loadGrids();
	} else if (newConfig.orientation == Configuration.ORIENTATION_PORTRAIT) {
		setContentView(R.layout.activity_game);
		gameView = findViewById(R.id.GameView);
		game = gameView.getGame();
		loadGrids();
	}
}
```

*Snippet 2: Example of manual rotation management*

```
private void endSplashScreenHandler() {
	setContentView(R.layout.activity_main);
	Objects.requireNonNull(getSupportActionBar()).show();
	greeting();
}
private class SplashScreen extends AsyncTask<Void, Void, Void> {
	@Override
	protected Void doInBackground(Void... voids) {
		try {
			Thread.sleep(5000);
			return null;
		} catch (Exception ignored) {}
		return null;
	}
	protected void onPostExecute(Void voids) {
		endSplashScreenHandler();
	}
}
```

*Snippet 3: Handling splash screen*

```
private void getPrefix() {
	Date date = new Date();
	Calendar cal = Calendar.getInstance();
	cal.setTime(date);
	int hour = cal.get(Calendar.HOUR_OF_DAY);
	if (hour < 12) {
		prefix = "Good Morning";
	} else if (hour < 18) {
		prefix = "Good Afternoon";
	} else {
		prefix = "Good Evening";
	}
}
```

*Snippet 4: Retrieving the correct prefix*

```
private void getStreak() {
	Date date = new Date();
	Calendar cal = Calendar.getInstance();
	cal.setTime(date);
	int day = cal.get(Calendar.DAY_OF_YEAR);
	int i = last_day;
	if (day > i && day != i + 1) {
		streak = 0;
		last_day = day;
		SharedPreferences.Editor editor = getSharedPreferences(PREFS_NAME, 0).edit();
		editor.putInt("streak", streak);
		editor.putInt("prev_day", last_day);
		editor.apply();
	}
}
private void updateStreak() {
	Date date = new Date();
	Calendar cal = Calendar.getInstance();
	cal.setTime(date);
	int day = cal.get(Calendar.DAY_OF_YEAR);
	if (streak == 0) {
		streak++;
		last_day = day;
	} else if (day > last_day) {
		if (day == last_day + 1) {
			streak++;
		} else {
			streak = 0;
		}
		last_day = day;
	}
	SharedPreferences.Editor editor = getSharedPreferences(PREFS_NAME, 0).edit();
	editor.putInt("streak", streak);
	editor.putInt("prev_day", last_day);
	editor.apply();
}
```

*Snippet 5: Retrieving and updating the saved streak*

```
private int randomNumber() {
	return (int) Math.floor((Math.random() * 9) + 1);
}
private void generateGrid() {
	for (int i = 0; i < 9; i += 3) {
		int number = randomNumber();
		for (int r = 0; r < 3; r++) {
			for (int c = 0; c < 3; c++) {
				while (!checkNumberSafe(i, i, number)) {
					number = randomNumber();
				}
				solution[i + r][i + c] = number;
			}
		}
	}
	fill();
}
```

*Snippet 6: Filling the 3 diagonal sub-boxes*

```
private boolean fill() {
	for (int r = 0; r < 9; r++) {
		for (int c = 0; c < 9; c++) {
			if (solution[r][c] == 0) {
				for (int number = 1; number <= 9; number++) {
					if (checkNumberSafe(r, c, number)) {
						solution[r][c] = number;
						if (fill()) {
							return true;
						}
						solution[r][c] = 0;
					}
				}
			return false;
			}
		}
	}
	return true;
}
```

*Snippet 7: Filling the rest of the grid*

```
private void saveGrids() {
	int[][] grid = game.getGrid();
	int[][] basic = game.getBasicGrid();
	int[][] solution = game.getSolution();
	String gridString = "";
	String basicString = "";
	String solutionString = "";
	for (int r = 0; r < 9; r++) {
		for (int c = 0; c < 9; c++) {
			gridString = gridString + grid[r][c];
			basicString = basicString + basic[r][c];
			solutionString = solutionString + solution[r][c];
		}
	}
	SharedPreferences.Editor editor = getSharedPreferences(PREFS_NAME, 0).edit();
	editor.putString("grid", gridString);
	editor.putString("basic", basicString);
	editor.putString("solution", solutionString);
	editor.apply();
}
```

*Snippet 8: Saving grid state*

```
private void loadGrids() {
	SharedPreferences settings = getSharedPreferences(PREFS_NAME, 0);
	String gridString = settings.getString("grid", "");
	String basicString = settings.getString("basic", "");
	String solutionString = settings.getString("solution", "");
	int[][] grid = new int[9][9];
	int[][] basic = new int[9][9];
	int[][] solution = new int[9][9];
	int counter = 0;
	for (int row = 0; row < 9; row++) {
		for (int col = 0; col < 9; col++) {
			grid[row][col] = gridString.charAt(counter) - '0';
			basic[row][col] = basicString.charAt(counter) - '0';
			solution[row][col] = solutionString.charAt(counter) - '0';
			counter++;
		}
	}
	game.loadGrid(grid);
	game.loadBasicGrid(basic);
	game.loadSolution(solution);
}
```

*Snippet 9: Loading grid state*

## UML Diagram

*Figure 22: UML Class Diagram of App*

## References

[1] S. Jones, P. Roach and S. Perkins, “Sudoku Puzzle Complexity,” *Proceedings of the 6th Research Student Workshop,* pp. 19-24, 2011.