# Progress Report

## Requirements Update

Summary

The main focus was to get most of the functional requirements in place before moving onto the non-functional requirements. The main reason was to allow for testing before moving to design. Using the 80 20 rule this appeared to be a better strategy because most of the non-functional requirements aren’t needed for it to be a “game”. Also, the non-functional requirements can be polished up leading towards the last stages of development.

One thing to note is that non-functional requirements were still be incorporated into the functional design this is to save time later making there less chance of going back to rewrite things to fit the non-functional requirements.

#### Overview

**Completed or mostly implemented:**

* 1.1 Game Basics and Rule
* 1.2 Left Click Action
* 1.3 Right Click Action
* \*1.5 Timer and Scoring
* \*2.3 High Scores
* \*2.5 Colour coding

\*: Mostly completed

**Priority:**

1. 1.4 Game Variants
2. 2.2 Customisability
3. 2.1 Speed
4. 2.4 Visually pleasing

### Functional

#### 1.1 “Game Basics and Rules”

This concept was almost fully complete in the prototype stage (milestone 1) and only needs some tidying up and more testing done before it is done. This was close to being #1 priority because it defines the game itself. Progress was made on this by incorporating multiple files to handle different game variants. Also, some tidying up was done to make the code flow better and enable additions to be added easily.

The dot points outlined in milestone 1 have been implemented and tested to make sure the game holds up as intended. Another win case was added in the code to determine if all cells (but those with bombs) have been revealed. This was something in the original game that I felt needed to come back.

#### 1.2 “Left Click Action”

This requirement was implemented during the prototyping stage in milestone 1 and seemed to work well. Slight tweaks are still needed to make it more robust and failsafe. The prototype implementation was more or less “Get it working” not get it working the best way it can. Many improvements have been made since then.

The code has been refactored to enable smoother performance and to not double up on any checks. The dot points outlined in milestone 1 are all in place and working and have been tested quite a bit. The released bombs are now monitored as a count to enable easier check for win case. This allows for a smoother experience for the player.

#### 1.3 “Right Click Action”

This requirement was implemented in milestone 1 but has been revisited since. It now has a better process of checking if the player has won. And now flags can be place with right click on windows (previous but that was fixed during testing on both platforms).

The win case is no longer dependant on having a flag placed on the final bomb so the win check was slightly change to suit this better.

#### 1.4 “Game Variants”

As seen in the demo from milestone 1 the hexagon variant was implemented and working. However there was many small issues that have been addressed. The old code used to use two python programs to run each instance of the game. Now with the addition of a menu (will be described later in 1.5) both game variants can be launched from the game menu.

The hexagon game now is fully playable and has been tested enough to see that there is no know/obvious major bugs. Further testing will be done to make sure this variant will hold up during the final game.

The colour variant hasn’t yet been implemented because of time constraints with the project and I wanted to make sure the game was robust before adding another game variant. But due to the way the code has been factored it is wuite easy to add a window class to support a new variant of the game. This design choice was made to ensure future updates would be easy and effective to add.

#### 1.5 “Timer and Scoring”

This requirement was not applied during the prototyping phase in milestone 1 due to it not being needed since the main game mechanics were priority. Now they are in place there has been quite a few updates to how the game scores and how it shows the timer.

Currently the game uses a basic formula to give a starting “score” in seconds that counts down to 0. This initial “score” is given by:

This was chosen because it scales well how difficult the game is and how large the board is. Further tweaks may be made to this to allow for smoother scoring scale. Currently the scores are stored in an SQLite database, which allows for viewing of the high scores.

### Non-Functional

#### 2.1 “Speed”

This hasn’t been the main focus of the game but definitely been monitored throughout the development it will be revised during every major requirement addition. This is something hard to test until a rough Alpha game is completed/

#### 2.2 “Customisability”

Because design is completed, and the main focus is functionality this requirement cannot be implement until the end phase of development. This stage will be implemented towards the end once the core game is done.

#### 2.3 “High Scores”

Linking back to the timer and scoring this won’t be implemented until the timer and scoring is done but this will allow for persistent scores to be kept using SQLite

#### 2.4 “Visually pleasing”

one of the last things to be done along with the speed

#### 2.5 “Colour coding”

similar to the previous requirement

## Product Use Cases

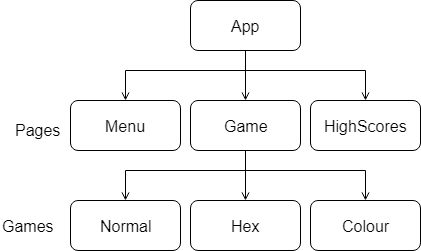
|  |  |  |  |
| --- | --- | --- | --- |
| Use Case Detail | Requirement  Relation | Status | Description |
| **1.0 Menu Screen** |  |  |  |
| 1.1 User clicks on play | 1.5, 2.4, 2.1 | Implemented | This should navigate the user to the game menu screen. |
| 1.2 User clicks on High Scores | 1.5, 2.3, 2.4 2.1 | Implemented | This should navigate the user to the high scores screen. |
| 1.3 User clicks on Quit | 2.1 | Implemented | This should close the entire game. |
| **2.0 Game Screen** |  |  |  |
| 2.1 User selects level | 1.1, 2.2 | Partially implemented | The levels are not yet pre-defined, and the user has to enter them manually. |
| 2.2 User clicks Go Home | 2.1, 2.4 | Implemented | This should navigate the user back to the main menu screen. |
| 2.3 User starts Game (Hex/Normal/Colour) | 1.4, 1.1 | Partially implemented | The user can currently play hex and normal and colour still needs to be added and tested. |
| **3.0 High Scores** |  |  |  |
| 3.1 User can view high scores stored | 2.3, 2.4 | Partially implemented | The user can view the high scores currently, but it isn’t neat and tidy yet still need some design to be added. |
| **4.0 Game** |  |  |  |
| 4.1 User can play Hex variant | 1.4, 2.1 | Implemented | The user can fully play the hex variant of the game |
| 4.2 User can play Colour variant | 1.4, 2.1 | Not yet implemented | Has not yet been made due to time constraints and I need to understand the variant better. |
| 4.3 User is scored | 1.5, 2.3 | Almost completed | The score has been added but may need tweaking |
| 4.4 Game is playable | 1.1, 1.2, 1.3, 2.5 | Implemented | The game works as defined in the requirements |
| **5.0 Visual Design** | 2.4 | Partially implemented | The foundations for design have been added not completed yet. |

## Software architecture

### Summary

The architecture that was used was designed so that it could work easily on both Windows and Unix. It is mainly an event driven system that incorporates some aspects from server-client such as the use of running another instance to handle the game while the “client” deals with the menu. This allows development to be split up into different parts allowing for easier creation of new features.

Because of this architecture choice the project was split into a few different phases. Each game variant can be independently made and tested before adding into the main application. This ensures effective development and more robust design. Below is a simple diagram created to illustrate the project. The Menu was developed separately and built so new pages can be added easily.



Because this project is a game, and the game requires a user to interact before anything happens the main focus was to use an event driven system. Using python bindings, functions were bound to run when the user clicks on something or in a certain way. This allowed the program not to go overboard when it was idle.

Languages/Tools/Platforms

The main programming language used was python. This was chosen because it is easy to use and has cross platform support. Not only that but it also comes with Tkinter a fairly decent graphics library allowing the implementation of the GUI. Python is an Object-Oriented language allowing for the use of most OO concepts, such as classes and objects. SQLite was also used inside of python to allow for the use of a database without needed an external server.

## Design

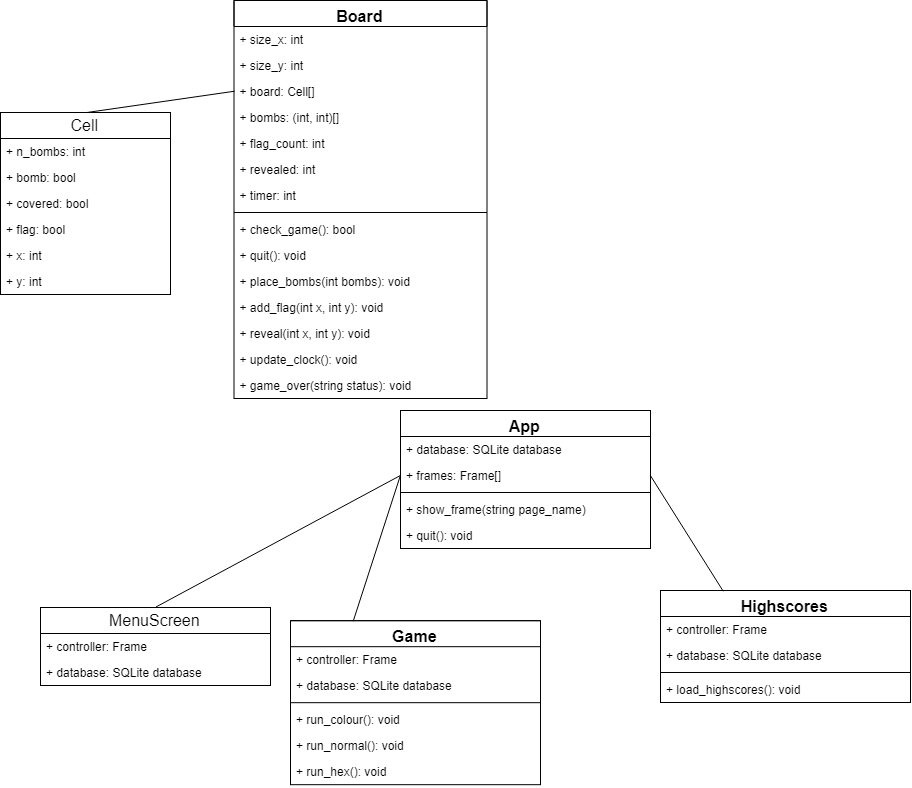
The main design goal was to make sure the code could be split up into sections. The was ideal because each game variant could be developed independent from one another. Then all added into the final project. As outlined in the Software Architecture section the Whole app can be broken down into just pages (for the menus) and games.

I went with a basic menu flow having a main menu, a game selector (games page) and a high scores page. I wanted to add a settings page but couldn’t really think of any reason it may be needed for this particular game.

As far as UI design goes most of it was outlined in the non-functional requirements. I wanted to go for a modernise version of Minesweeper but also have some relations to the older game.

### Class Diagrams

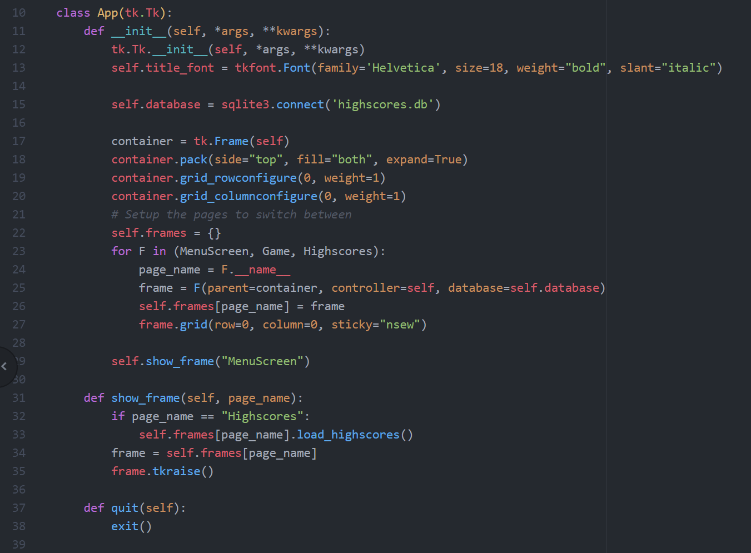
The following class diagrams have been updated from milestone 1 because of feedback given. I added more classes and more functions to make sure they are more relevant and cleaner.



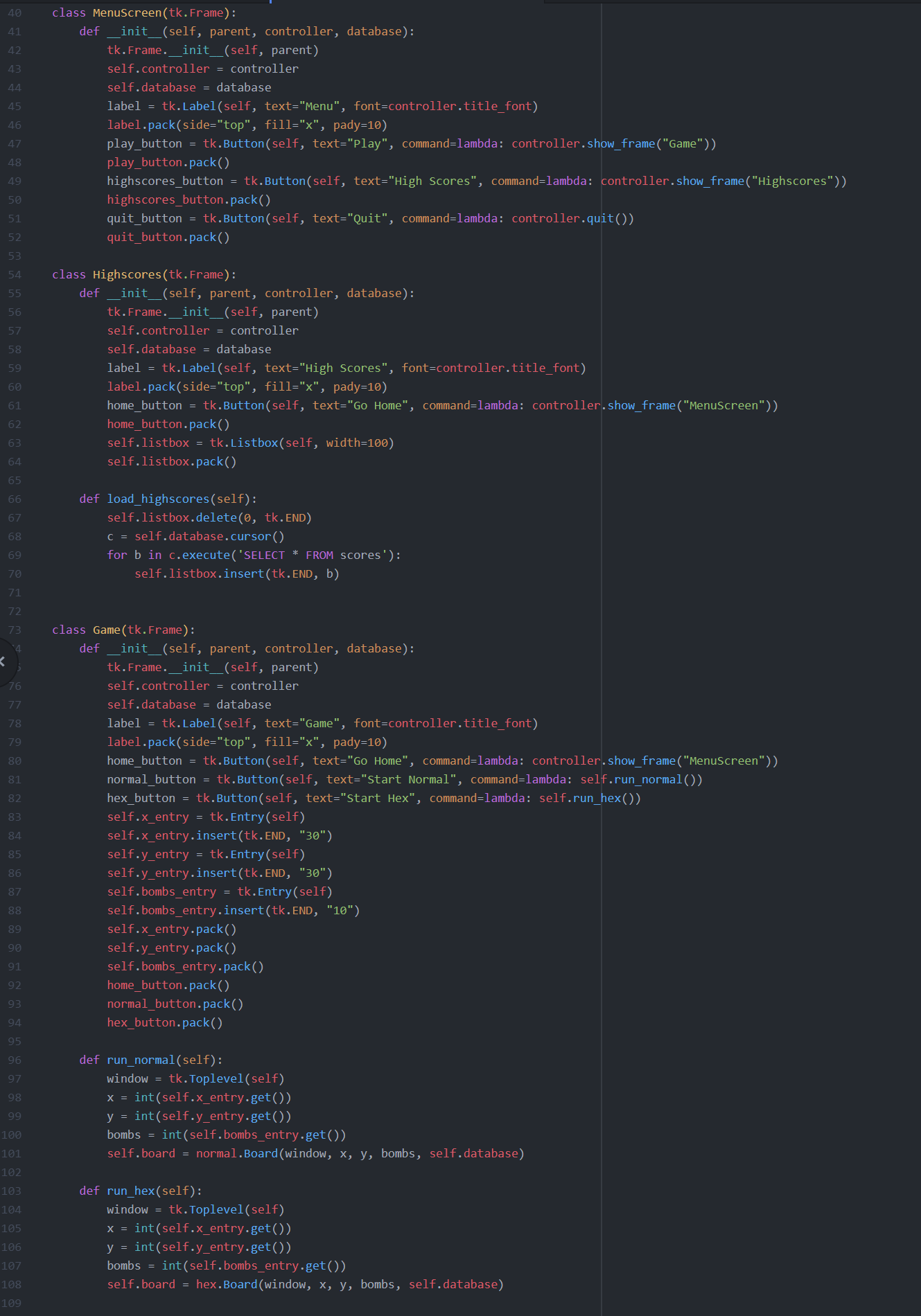
### Main App

Both the sequence diagram and collaboration diagram help show the main flow in the App. Below is the code that uses python and Tkinter to implement the design.

#### App code

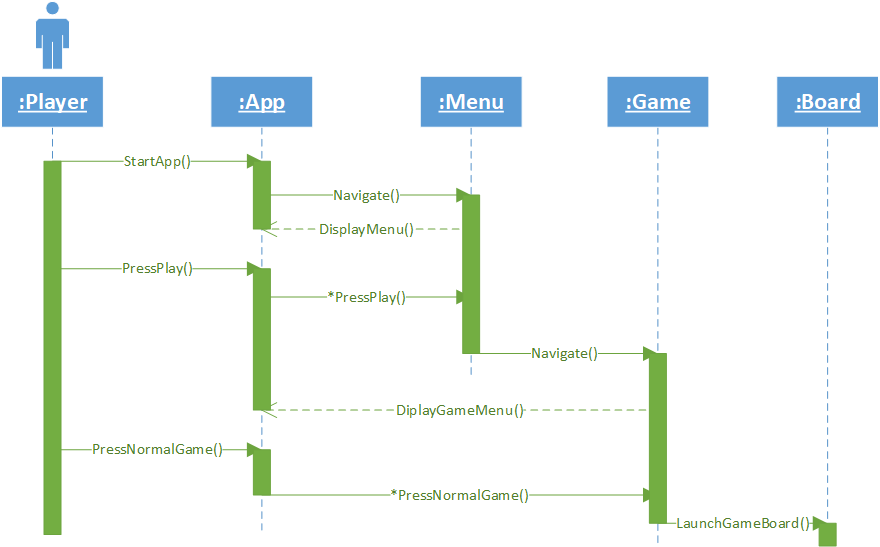


#### Frames code



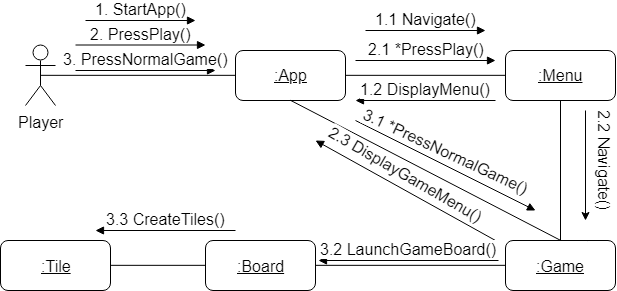
#### Sequence Diagram

This sequence diagram shows how the user would interact with the app in the case they wanted to start a game. While the function names outlined in the sequence diagram are different to the code they are more to illustrate the flow of using the app.



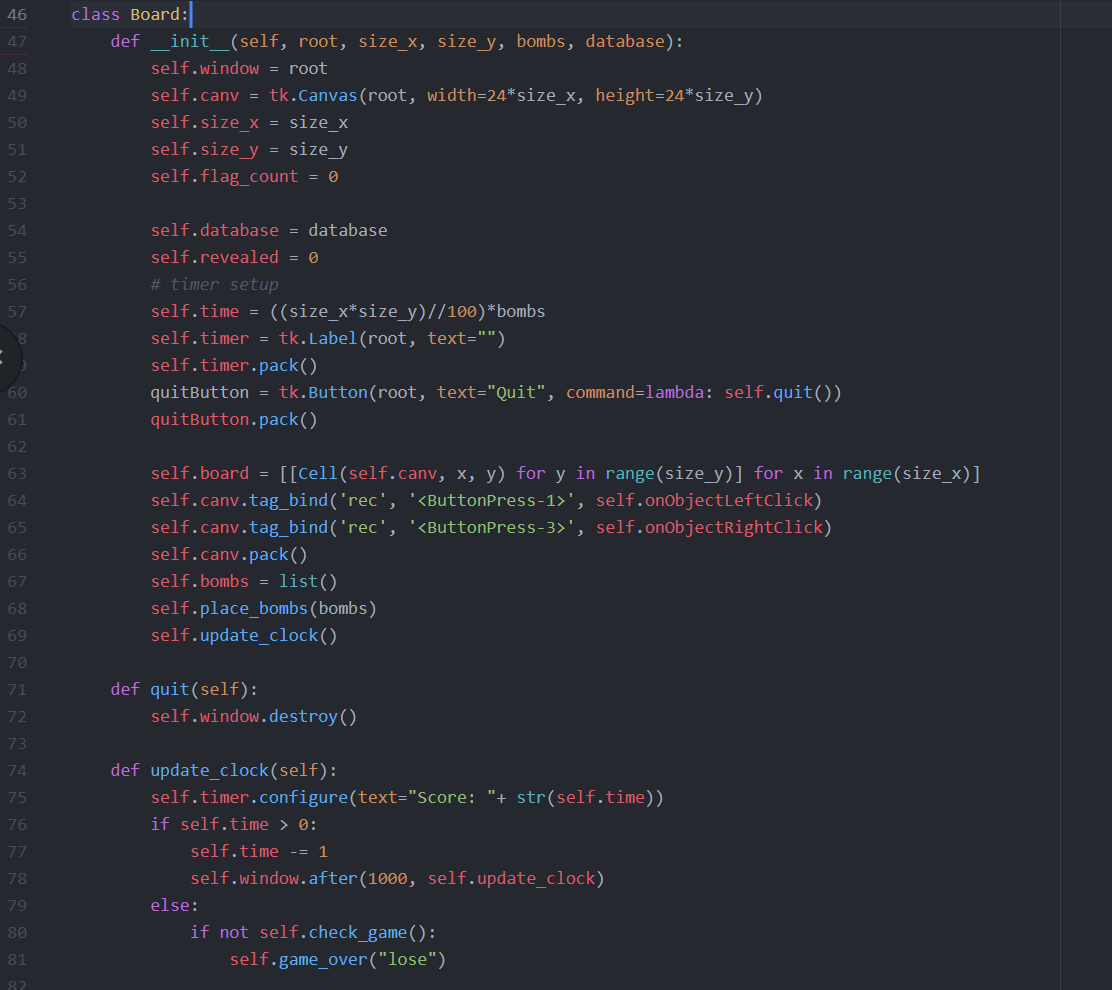
#### Collaboration Diagram

This collaboration diagram helps to show the flow of the objects as different use cases are performed. The connecting lines show how the objects are related to each other and what action links them.



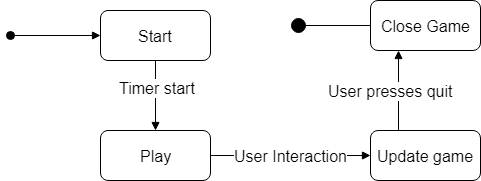
### Normal Game

All of the games variants act very similar, so I will try to outline just the important parts in all of them. The state machine diagram helps to outline the different states the game generally goes through as the player plays the game (focusing on the timer). While the activity diagram shows the flow of the game in terms of actions. The code below is a small snippet outlining the setup for the game and showing how the timer is implemented. (Note comments have been removed to fit inside of image).



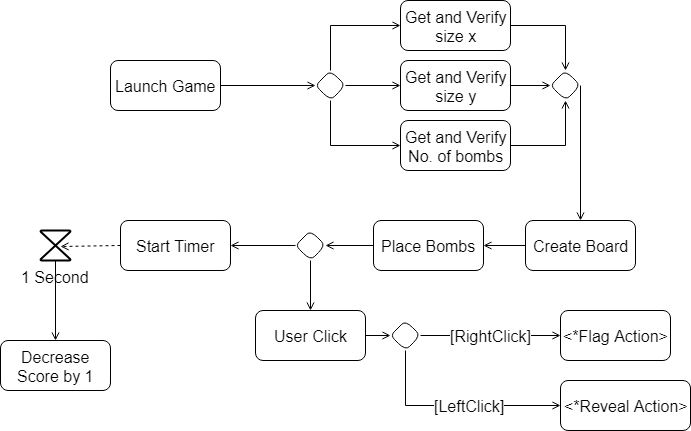
#### State machine for a game instance

This simple state machine outlines the flow of states for the case of user ending the game.



### Activity diagram showing basic game flow

The activity diagram below shows what happens when the game starts up and the flow of the board generation. The “flag action” and “reveal action” are defined in the requirements and there wasn’t a need to reiterate them.



## Persistent Data management

To keep track of high scores some method of persistent data was needed. SQLite was chosen mainly because it is easy to use within python and allows for quick querying of data sets. The schema of the data base was kept as basic as possible but could maybe be improve upon completion of the game. Currently the data base consists of one table show below:

### Scores

* Type: text (defines what game variant the score is for)
* Level: int (defines the predefined level that the score is for)
* Name: text (the name of the player who got the score)
* Size\_x, Size\_y, bombs: ints (they store the size of the board might be useful later on)
* Score: int (the score the player achieved)

## Testing

### Summary

The main testing that was done throughout this project is a similar technique to TDD but somewhat less formal. Each function was implemented then the whole app was run to make sure that, that bit of code didn’t alter anything it wasn’t designed for. There are four major parts of the game:

1. Menus
2. Normal Game
3. Hex Game
4. Colour Game

### Menus

The menus were mainly tested by running the main application. The main things that needed to be working was the navigation. Each page or menu screen needed to smoothly be able to change as the user clicked on the corresponding button. The was tested by simply running the application and making sure you can navigate back and forth without having any issues.

The other aspect of the menus that was tested was opening the games themselves. This was again done by actually running the app, but different values for the board size were hardcoded to make sure it works on different board variations.

The final thing that needed to be tested was making sure the database can be pulled from. This is done again by opening the main app and making sure the GUI displays the correct information. Further plans for testing the database will be making sure the queries work as expected, but since the high scores don’t filter yet this hasn’t been implemented.

### Games

Each of the game variants were developed independently, and they all had smaller features implemented. The variants have followed a very similar testing practice as the original game, so they won’t be described.

The normal game was developed first, and the logic was displayed to the console. This was tested just to make sure the games logic held up showing the mines in CLI and have the user enter x and y moves to simulate clicks. The benefit of doing this was to make sure the logic works before trying out the GUI. Also, certain scenarios could be tested before hiding the bombs.

Once this was working well and tested, it was moved to Tkinter to actually display the board. One issue I had while doing this was that I knew I couldn’t use buttons since Tkinter doesn’t support hexagonal buttons, so I used a clickable canvas. A lot of tweaking went into to make sure it worked without miss clicking but after a lot of testing it ended up working well. Once this was done a simple menu was made and incorporated into it.

### Plans

Using this modular testing method I plan to use it for developing the colour variant and testing out other menus or just altering the current one.

Version control

Software patterns

Describe at least one implementation of a software pattern illustrated by some existing code in the current version of the project or prototype.

OO

State based

Class diagrams