# 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 UI 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 completed in the prototype stage (milestone 1) and only needs some tidying up and more testing done before it is completely 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 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 revealed cells are now monitored as a count to enable easier checking for a 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 (previously it was middle click but that was fixed during testing it 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, 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 are 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 the focus was to make sure the game was robust before adding another game variant. But due to the way the code has been factored it should be quite 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 is scored 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 to 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 has been revised during every major requirement addition. This is something hard to test until a rough Alpha game is completed. But a lot of design choice were made with this in mind.

#### 2.2 “Customisability”

Because design isn’t 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 and the variants are in place.

#### 2.3 “High Scores”

This has been implemented but only to a very basic point. Currently the user can play the game and a high score will be stored but the method for view is somewhat basic. Currently all it shows is a complete list of all the high scores in all the different game variants. This will be later changed to filter and search different levels.

#### 2.4 “Visually pleasing”

This requirement has not really been the main focus and is why it is last on the list to finish. It will be the final polishing up when the game is done and will and a nice edge to the game. As described before it should look a bit more modern but also keep older features of the old game.

#### 2.5 “Colour coding”

This was added quite early on and was in the prototype. Currently it works as intended but will be revisited during completion of the game similar to requirement 2.4. Small tweaks on the colours will be added to make it seem more modern and slight changed to the style of the fonts are planned.

## Product Use Cases

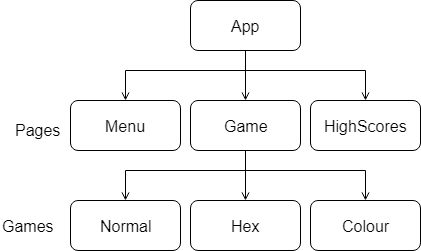
|  |  |  |  |
| --- | --- | --- | --- |
| Use Case Detail | Requirement  Relation | Status | Description |
| **1.0 Menu Screen** |  |  |  |
| 1.1 User clicks on play | 1.5, 2.4, 2.1 | Implemented | This should navigate the user to the game menu screen. |
| 1.2 User clicks on High Scores | 1.5, 2.3, 2.4 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 and architecture of the software as a whole. 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 clicks in a certain way. This allowed the program not waste processing 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 needing an external server.

## Design

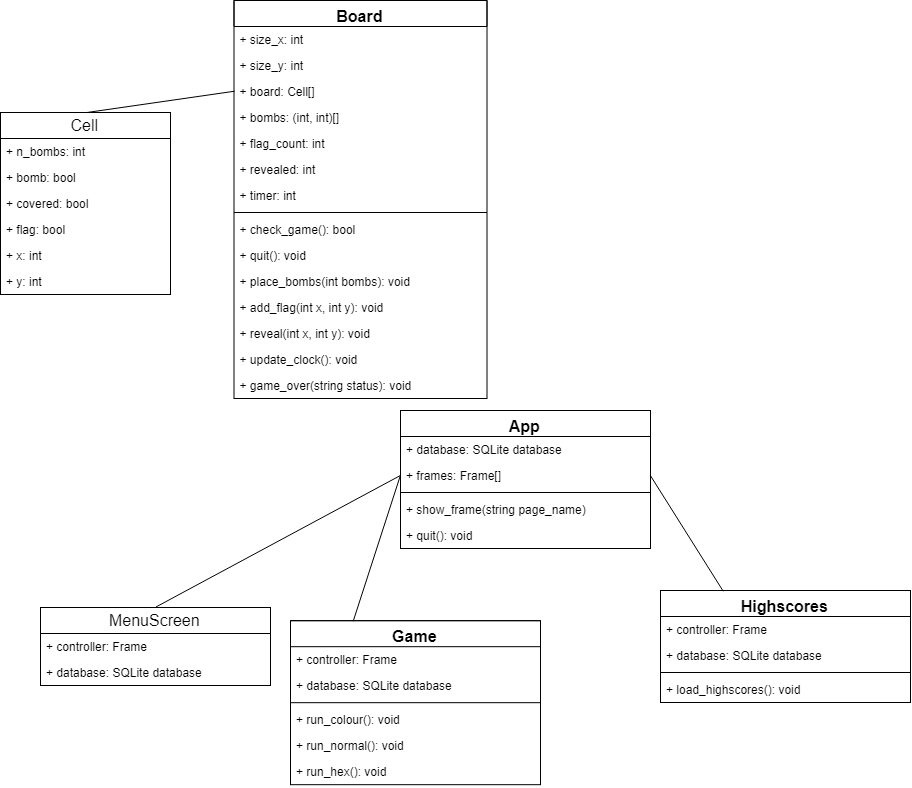
### Summary

The main design goal was to make sure the code could be split up into sections. This was ideal because each game variant could be developed independently 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 and games variants.

The GUI consists of three pages: “MenuScreen” or just “Menu”, “Game” and “HighScores”. The Menu screen is there to be an entry point of the Application and allows users to navigate between high scores and the games themselves. As far as UI design goes most of it was outlined in the non-functional requirements. But it should be a modernise version of Minesweeper but also have some relations to the older game.

### Class Diagrams

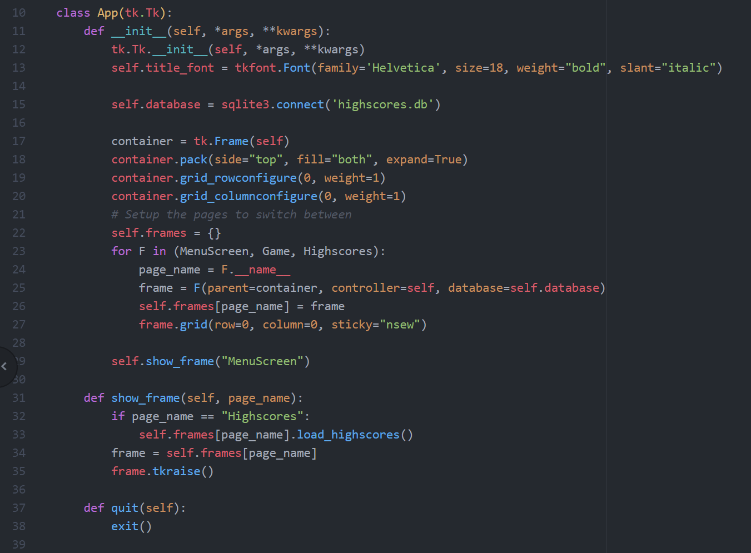
The following class diagrams have been updated from milestone 1 because of feedback given. More classes were added and more functions to make sure they are more relevant and cleaner. This also allows for future improvements to be made to the overall structure.



### Main App

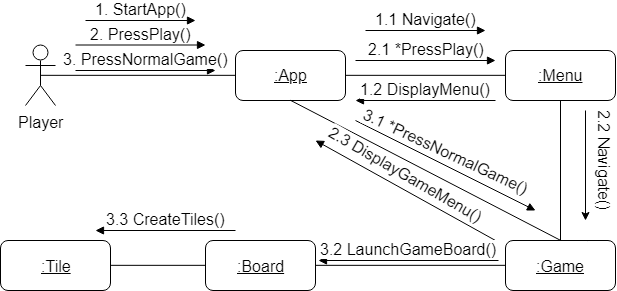
The following code is the main setup for the GUI and pages. Using tkinter and frames that overlap it allows for a nice navigation flow by using the function show\_frame(“Page to display”). It does this by bringing the page to the front which forces that page to be active.

#### GUI code



#### Collaboration Diagram

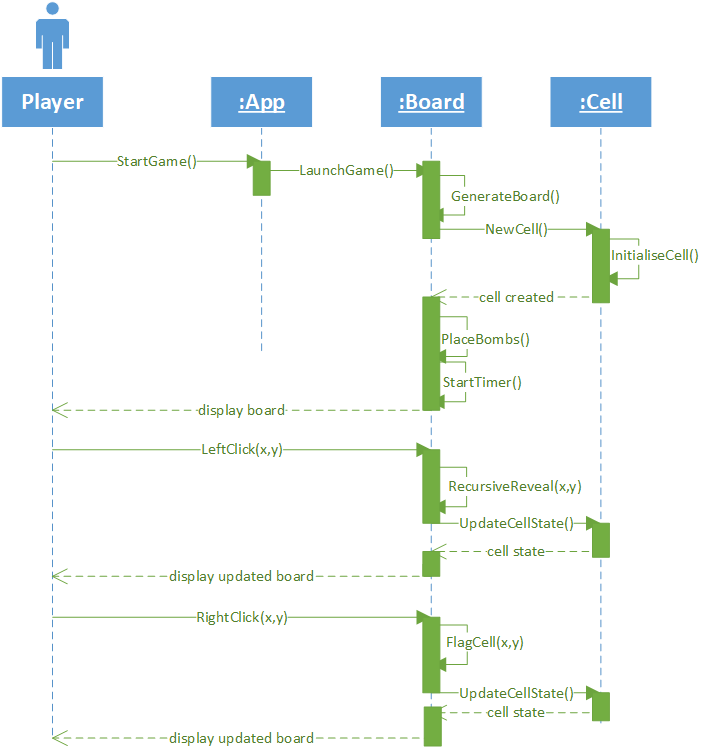
This collaboration diagram helps to show the flow of the objects as different use cases are performed. The connecting lines show how the objects are related to each other and what action links them. The main aim for this diagram was to show how the user would navigate to a new game using the GUI.



### New Game setup

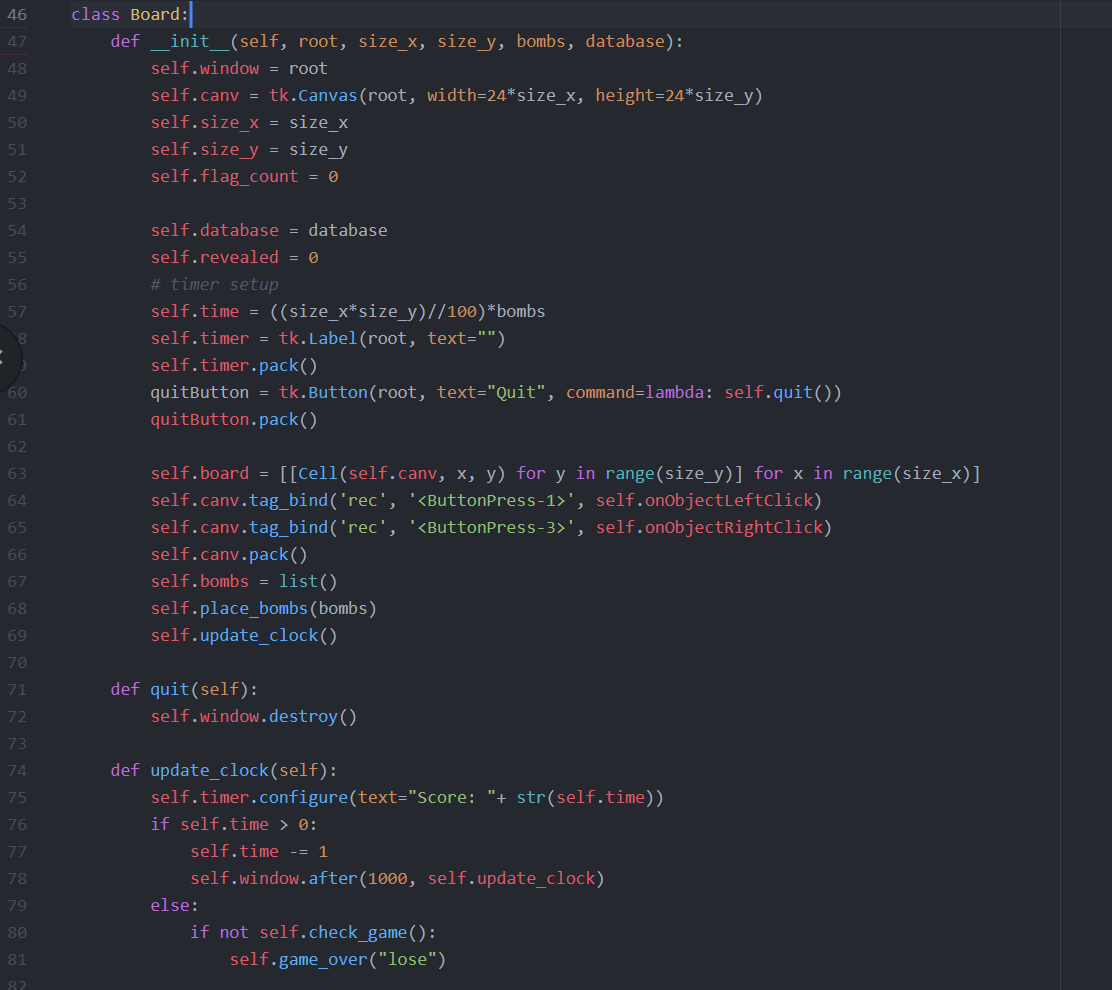
The game mainly consists of a Board which is the where all the logic for the game is processed and game itself is setup. Once the user clicks on start game, an instance of the game will launch in a new window. All the initial setup for the game is run once the board is created. The following sequence diagram outlines the flow of starting a new game.

#### Sequence Diagram



#### Board code

The following code is just to illustrate how the board is setup using python and Tkinter. Some of the code was left out and comments were removed to shorten the image size.



### Playing the Game

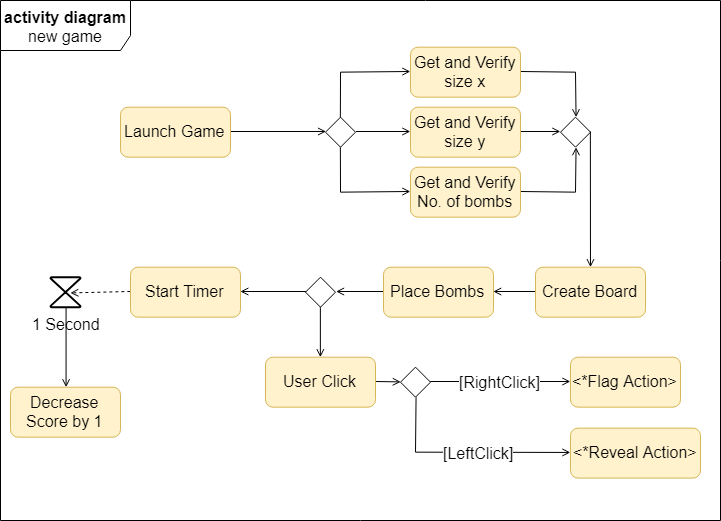
The flow of playing any variant of the games should be all the same or very similar. Other than the initial setup of the game there are three main functions that allow the game to work. These three functions are:

* Reveal
* Flag
* Place Bomb

The following section will outline how the game is played through its life.

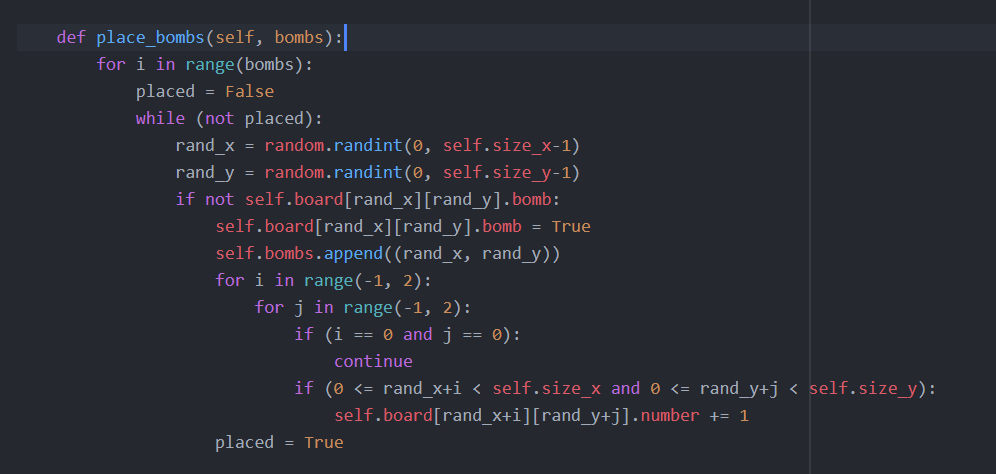
### Activity diagram showing basic game flow

The activity diagram below shows what happens when the game starts up and the flow of the board generation. This is a basic overview of how the game is played and the function will be described later. This should outline what the main activities of playing the game are.



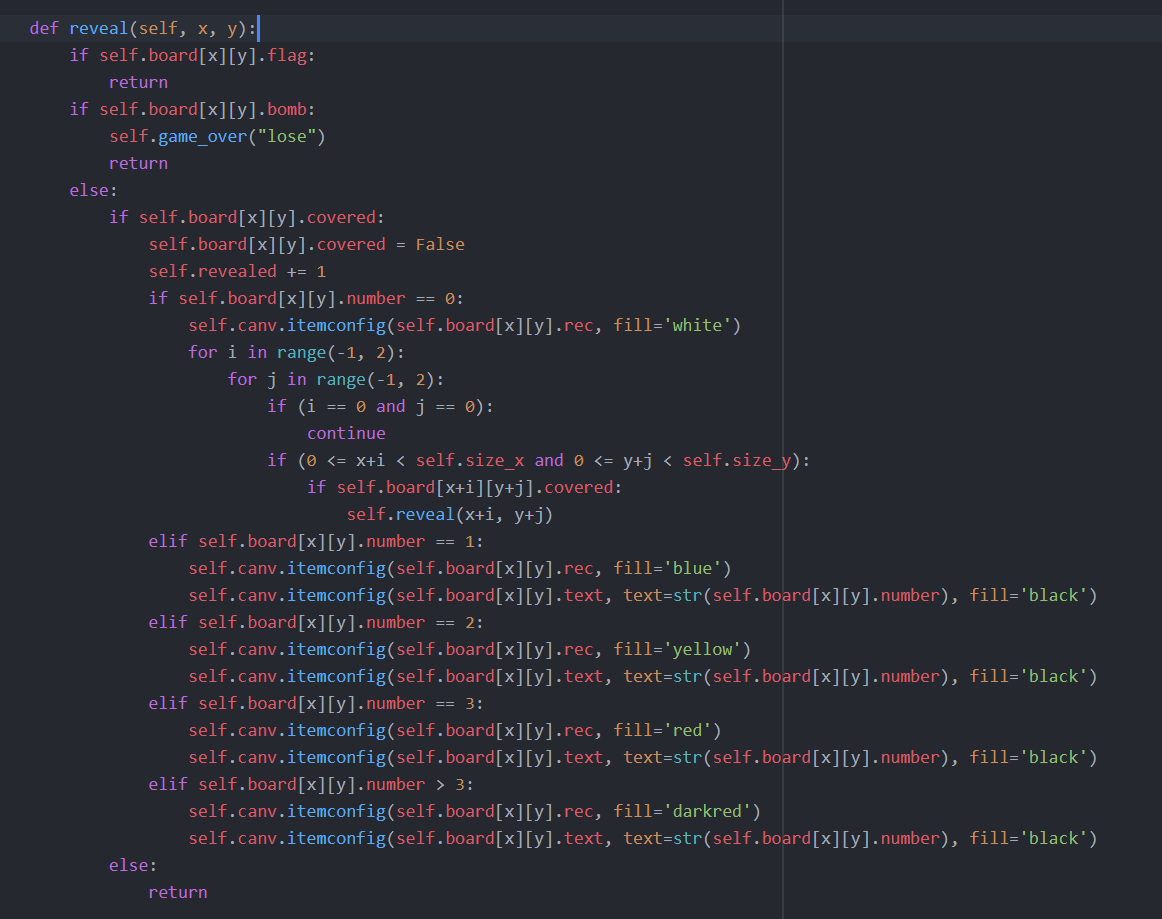
#### Place Bomb

In order for there to be any game at all there needs to be a goal. Like the original minesweeper the game consists of finding where bombs are and flagging them. The bombs are first hidden and could be anywhere on the board. But after the user clicks on a cell to reveal it, the board shows all the cells that are not bombs and shows numbers of how many bombs are near each cell. In order for this to work each bomb is placed around the board randomly, then each neighbours “neighbouring bombs” count is incremented to save time later. Example of the code below:



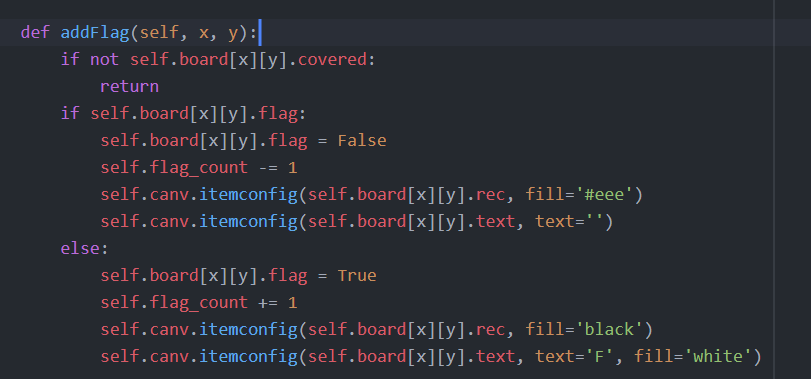
#### Reveal

Once all the bombs are place the user can then start revealing bombs by clicking on cells. If the user clicks on a cell containing a bomb the game is over. Else the board is recursively revealed from that point. The following code shows how the cells are revealed and how they appear to the user using Tkinter:



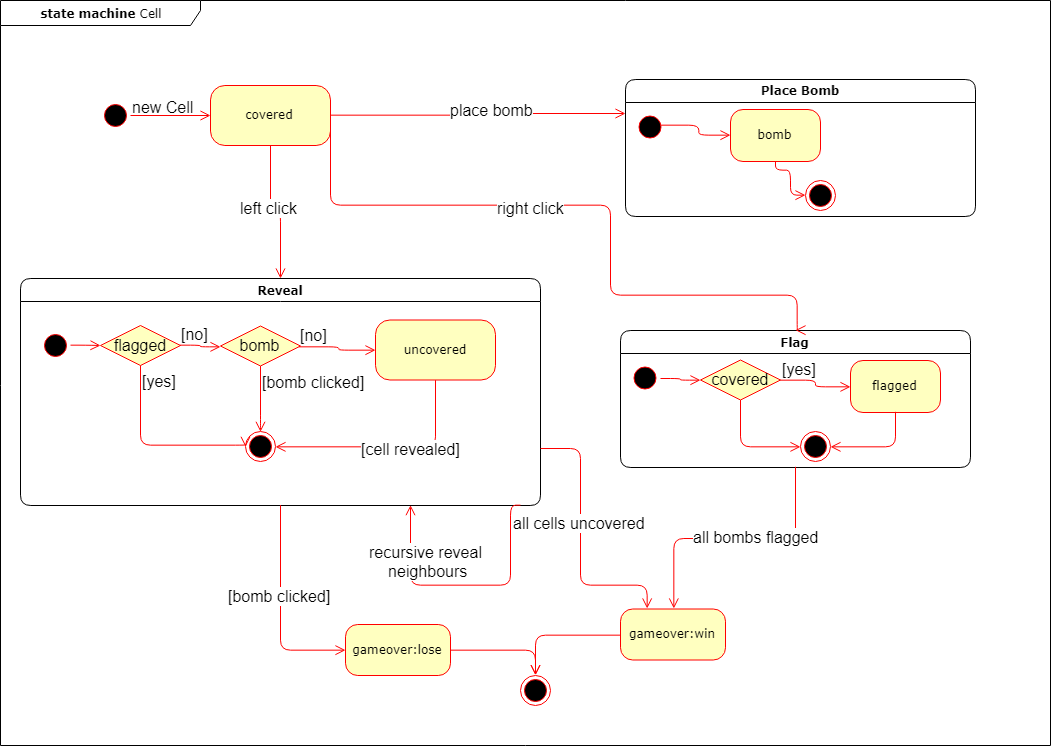
#### Flag

There are two ways for the user to win, if the user flags all the bombs or if the user reveals all the non-bomb cells. The user can flag a cell meaning it cannot be revealed and is marked so they can keep track of where they think the bombs are. The following code outlines how this is done:



#### State machine for a cell

While looking at the code it is hard to see what state the cell is in. The following state machine diagram should illustrate how the cells states are changed throughout the life of the game.



## 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 database consists of one table shown 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 when filtering)
* 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 the new 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 thing 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. This was tested by simply running the application and making sure the user 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 visually display the board. One issue found was that Tkinter does not support hexagonal buttons, so the program uses a clickable canvas instead. 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, the plan is to use it for developing the colour variant and testing out other menus or just altering the current ones. Test driven development seems like the best way to go since it enables consistent progress and has an agile like development feel. The plan is to make sure all functional requirements are done before any progress to tweaking the design and optimising code.

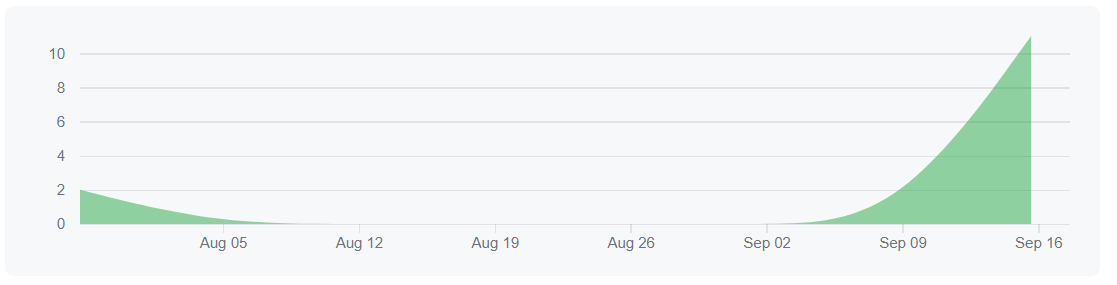
Version control

The version control used two methods to make sure nothing was lost, and progress can be made. During the start of development there was a lot of changes and commits to get the prototype working. Then a USB was used to back up even more.

As seen in the histogram there is a large gap in development. This is due to other classes taking time and no progression made to the code base until the start of September.  
Below is the raw commit history.

* 688b145 - (HEAD -> master, origin/master, origin/HEAD) slight changes and more documents added <Carl Humphries>
* dc63e3d - draft of progress report about half way done, also some updates to database <Carl Humphries>
* b9eb0da - Defining how the pages are setup <Carl Humphries>
* ab6bbf0 - Added quit on main menu <Carl Humphries>
* 168545d - fixed scores not showing up hiding the score <Carl Humphries>
* 3d87ed0 - added highscore page to view highscores <Carl Humphries>
* 0612637 - Added basic databse structure for highscores not fully implemented yet opr tested <Carl Humphries>
* e5d2024 - Added timer to use as score it is defined as x by y divide 100 times bombs <Carl Humphries>
* ea33f8f - added documentation to git repo from milestone 1 lost progress on work <Carl Humphries>
* d10d98d - Added basic menu to launch different versions of the game <Carl Humphries>
* 97d6533 - refactored code to have hex and normal minesweeper in separate files <Carl Humphries>
* b39da7c - Rough prototype for hex and square based minesweeper <Carl Name>
* b81c831 - Initial commit <Carl Humphries>

Histogram of commits:



Software patterns

### Information expert (GRASP)

Context:

Because the game heavily relies on the information about the cells, a choice was needed to be made about where to store this information. This would allow the delegation of responsibility for what is happening to each cell.

Problem:

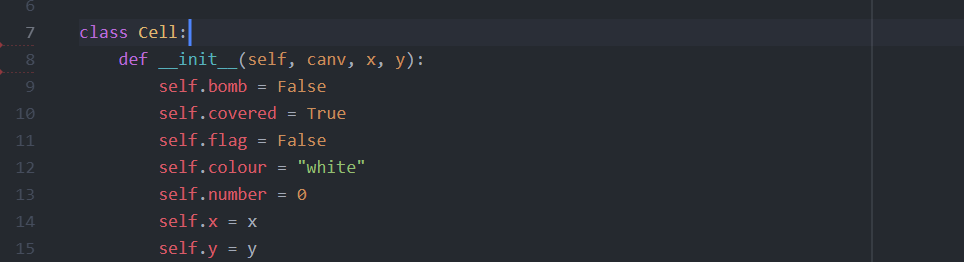
Find the best way to store this information and the best way to delegate the responsibility to somewhere. Represent the relevant information in order to fulfil the responsibilities needed.

Solution:

Create a Cell class that will store the relevant information to fulfil the responsibilities. And place the responsibility of dealing with the cell states onto the cell. This is because the cell will contain the most information required to fulfil the responsibility.

Code:

Below is a snippet of code to outline how the information is stored within the cell giving it the responsibility of the information about its current state.



### Creator

Context:

Since each game is independent from one another they rely on being created by something in order to be able to play each game inside the same app.

Problem:

What class should be responsible for the creation of the Boards? The class should contain and instances of the board. The class should be able to record instances of the boards. The class should closely use the board and the class should be able to contain the setup information used to create the board.

Solution

Using the Creator pattern in GRASP creating a class called Game and having it create each instance of the boards using input from the GUI. This will allow one class to ultimately deal with each instance of the boards and deal with the creation of them.

Code:

