3815ICT – Software Engineering

Minesweeper - Milestone 2

Student: Zak Barker

Student#: S5085150

Subject: 3815ICT Software Engineering

Email: Zak.Barker@Griffithuni.edu.au

1. Analysis of Requirements

1. Complete

At this stage, functionality has been implemented for the completion of a standard minesweeper game in an isolated environment. Previous listings of functional requirements detailed the following key aspects of the game:

- Implementation of Title Page Complete with Navigation
- Implementation of Gameplay Grid
- Implementation of Gameplay Cells with Mechanics
- Implementation of Gameplay Flags
- Implementation of End Game

All gameplay elements have been completed in regards to the standard minesweeper game. Aside from this, non-functional requirements have been implemented including:

- GUI
- Performance – No lag
- No install

2. Ongoing

Previously, functional requirements pertained to a single instance of the minesweeper game – being the standard minesweeper. An overhaul of functional requirements will be necessary as previously, they only described a single game mode. New functional requirements will need to be created to accommodate for the additional game modes being: hex and colour. In terms of previous functional requirements, the only one which is still ongoing is the Implementation of a Title Page.

For prioritization, the following list describes the important tasks to complete and the order in which they will be prioritized.

- **Title Page Complete with Navigation to all Screens** – The reason this is prioritized first is the ease of implementation vs the structure it will provide to the subsequent requirements. It will be important to place the already completed Minecraft game within the context of the entire project. Furthermore, it will be useful to develop the subsequent game types within the context of the entire game as it will give a good feel for the flow of the UI as a whole while developing. It will also aid in testing as each additional piece of the program can be

tested against the entire project instead of being created separately and hoping for a smooth merge.

- **Prototyping of Hex Minesweeper –** Hex Minesweeper will be the next challenge and the same functional requirements used within the standard minesweeper game can be translated as a subset for this version. Very few changes should be necessary, however, the GUI will need to be re-themed.
- **Prototyping of Colour Minesweeper –** Once Hex has been implemented, a new game type will need to be developed. At present, the concept for this game is confusing so the details of the precise functional requirements involved will be necessary. This will be prioritised, mapping the functional requirements for the colour game and developing a prototype.
- **Cross-Platform Capabilities –** This final requirement will be important to address once the initial Minesweeper project has been completed. At present, the goal is to provide functionality across web browsers and for Microsoft, Windows and Linux devises running web browsers.

2. Analysis of Use-Cases

Originally, a single use cased was developed which described the process of the user reading the game instructions and proceeding to win a game of minesweeper. There was an alternate flow of events documented in which the user would not win their first game. The user would lose the game and then decide to restart the game – thus exploring majority of game functionality. The following historical use case diagram described the user interacting with all aspects of the game.

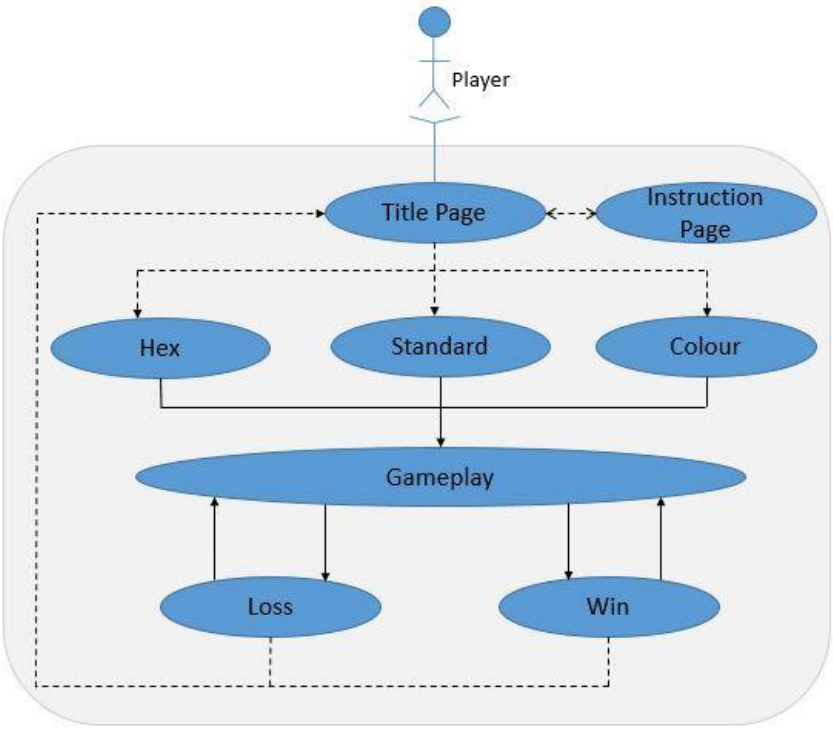

Figure 2.1 – Use Case Diagram

For an overview of the entire game structure, this use-case is fine, however, in analysing this use of a use-case, it is clear that the methods applied were misguided. A series of shorter use-cases should be employed to further explore more niche aspects of the game. Instead of a large use case involving everything, smaller use cases should be developed to gain insight into how a user might interact with the application. This can only improve the quality of the game as a whole.

The use-case which documented the case of a user winning the game is still pending. This is because the use-case follows the user's interaction with instructions and with a title screen. This has not been developed. Besides this, a user is able to play a game through to completion either winning or losing. In this case – the alternate flow in which the user loses and replays is near-completion, as is the primary flow. Future use-cases will involve user involvement with different types of games.

3. Analysis of Software Architecture

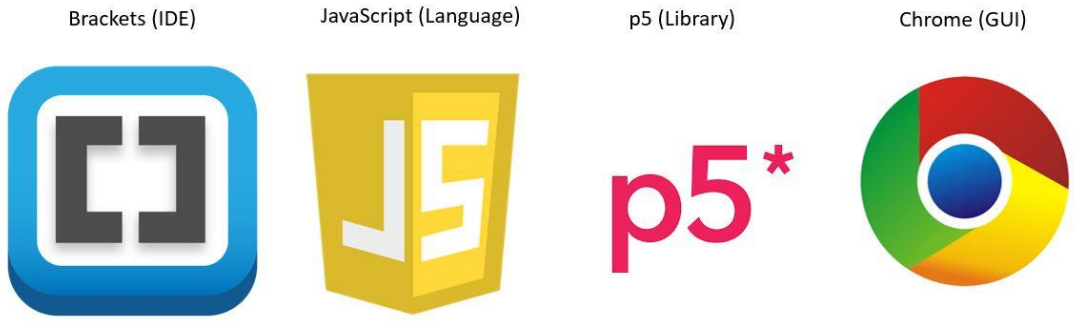

Figure 3.1 – Software Architecture

The current IDE being utilised to implement minesweeper is brackets. This is free software and relatively simple to use. The language being used is JavaScript. Though there are no class models implementable in JavaScript, there are ways in which one may mimic them. Finally, the p5.js library is being used to display the front end GUI for the project

For a small project such as minesweeper, this software architecture is surprisingly efficient. There is unlimited room for GUI customization with the p5 library and JavaScript has more than enough power to deliver the required functionality. “Classes” may be written into modules and methods may be applied via ‘prototyping’ functions, these modules may be called from the main script to display components to the user with inbuilt logic.

In terms of feasibility, this architecture is easy to use, costs no money, is flexible and is able to get the job done. All things considered, this architecture may be basic but it is more than feasible for this particular project. This architecture will remain throughout project completion.

There will be modules to govern the separate functionalities of each type of cell based on the game type. For instance, a hex tile acts slightly different to a colour tile. These three modules will be built upon a collection of modules which are common to the functionality of all types of cell. Ie, the ability for a cell to be revealed.

4. Summary of Design

1. Design Inconsistencies & Goals

At present, there exists only a single module for the Cell Class. This class is exclusively for the standard square minefield and does not take low coupling/high cohesion in to consideration from a

design perspective. The main design goal in future iterations of the project is to break this class down into its modular portions and prepare it for reuse. Ahead is a static model of the minesweeper game.

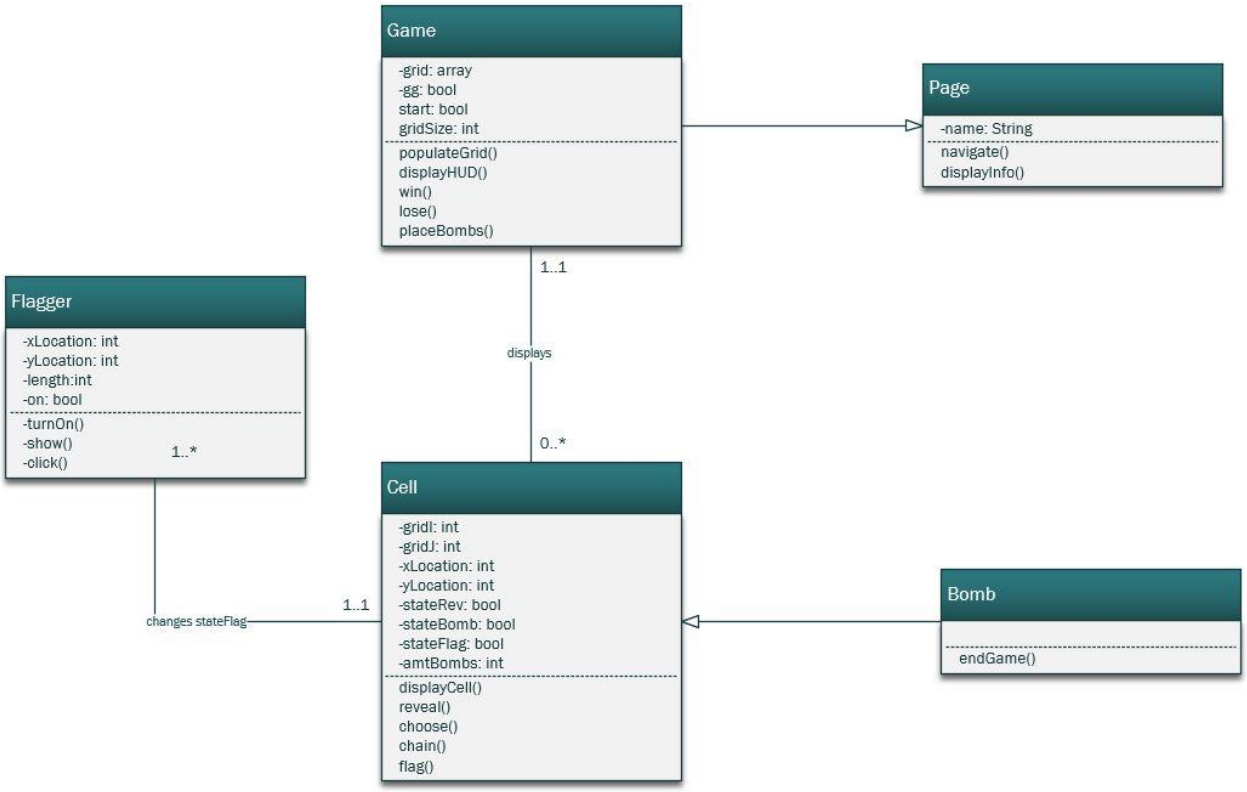

Figure 4.1 – Minesweeper UML Class Diagram

The **Game** class has not yet been implemented, nor the **Flagger**. The main index file which contains the functionality for the game contains all of the functionality for both of these classes. This must be broken down. **Flagger** has been implemented as an object, this will need to be implemented as a separate class and imported for each separate game type.

```
flagger = {
    img: treetree,
    x: 90,
    y: 40,
    l: 50,
    on: false,
    turnOn: function () {
        if (this.on) {
            this.on = false;
        } else {
            this.on = true;
        }
    },
    show: function () {
        imageMode(CENTER);
        if (!flagger.on) {
            image(flagger.img, flagger.x, flagger.y, flagger.l, flagger.l);
        } else {
            fill('#008e54');
            ellipse(flagger.x, flagger.y, flagger.l + 20);
            image(flagger.img, flagger.x, flagger.y, flagger.l, flagger.l);
        }
        imageMode(CORNER);
    },
    click: function () {
        if (mouseX > flagger.x - flagger.l / 2 && mouseX < flagger.x + flagger.l / 2 && mouseY > flagger.y - flagger.l / 2 && mouseY < flagger.y + flagger.l / 2) {
            flagger.turnOn();
            print(flagger.on);
        }
    }
}
```

Figure 4.2 – Minesweeper Flagger Class

The **Cell** class contains a multitude of functions which will be transferable to other game types. These functions will need to be modularized to promote low coupling. The modules will be accessible from all variations of type **Cell** in each separate game. The class **Cell** will need to be tweaked for each variation of minesweeper so this may come down to separate **Cell** classes or perhaps a new set of subclasses for the **Cell** class – each subclass being a different cell type(standard, hex, colour). Functionality which can be modularized for reuse are:

- reveals() – reveals a cell and triggers subsequent game states.

```
Cell.prototype.reveals = function () {
    if (!gg) {
        if (!this.reveal) {
            revealCount++;
        }
        this.reveal = true;
        if (this.amtCrabs == 0) {
            this.chain();
        } else if (this.crab) {
            lose();
        } else if (revealCount + allCrabs == 100) {
            win();
        }
    }
}
```

Figure 4.3 – Minesweeper reveals() Function

- chain() – Provides flow-on effect when revealing tile not adjacent to bomb

```
Cell.prototype.chain = function () {
    for (let xoff = -1; xoff <= 1; xoff++) {
        for (let yoff = -1; yoff <= 1; yoff++) {
            let i = this.i + xoff;
            let j = this.j + yoff;
            if ((i > -1 && i < cols && j > -1 && j < rows)) {
                let neigh = grid[i][j];
                if (neigh.reveal == false) {
                    neigh.reveals();
                }
            }
        }
    }
}
```

Figure 4.4 – Minesweeper chain() Function

- flag() – locks Cell based on Flag state

```
Cell.prototype.flag = function () {
    print('works');
    if (!this.reveal && !this.flagged && flagger.on) {
        this.flagged = true;
    } else if (this.flagged && flagger.on) {
        this.flagged = false;
    } else {
        this.reveals();
    }
    print(!this.reveal && !this.flagged && flagger.on);
    print(this.flagged);
}
```

Figure 4.5 – Minesweeper flag() Function

Some of the other functions are specific to the shape of the Cell they are attached to. This is why a subclass design might be more suitable as these Cell type specific functions can simply be overridden in that case. Currently modularization is the goal, however; subclassing will be prototyped in future.

2. Dynamic Design Model

Provided is a sequence diagram modelling the interaction between the user and the systems within the game:

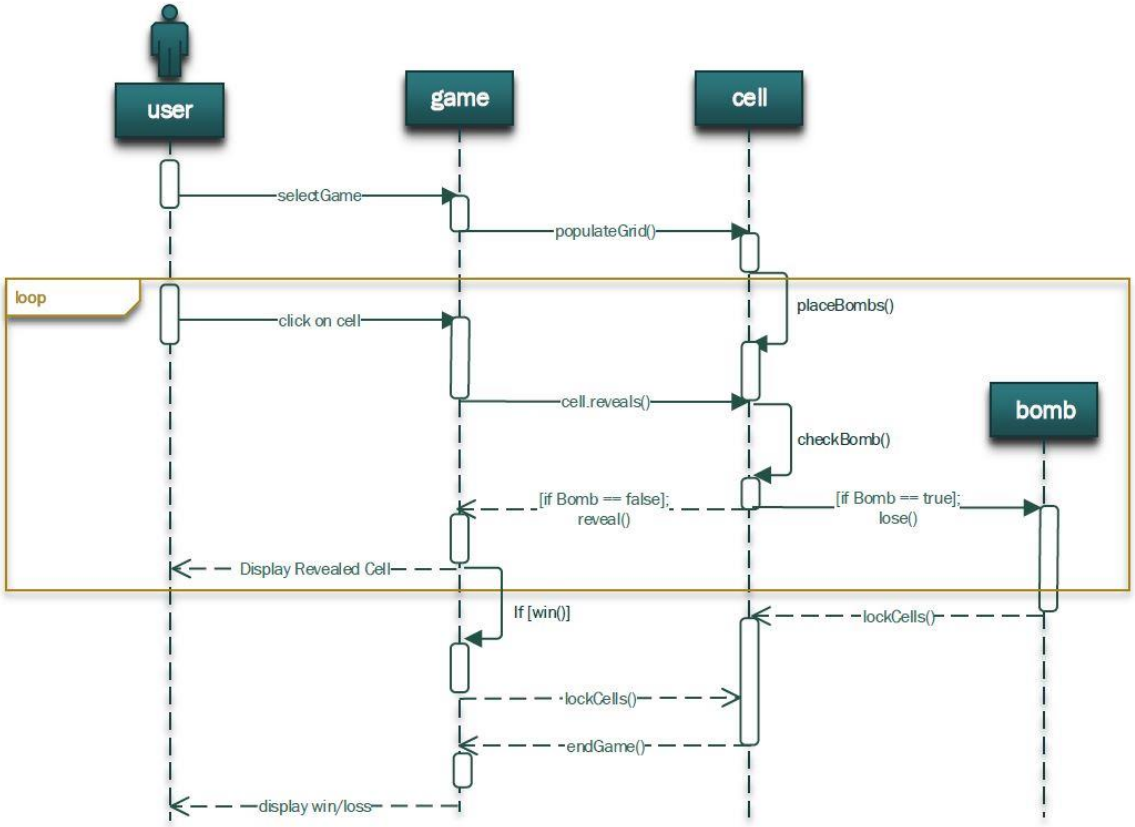

Figure 4.6 – UML Sequence Diagram

The user interacts with the GUI and based on this interaction, the systems within the game return

visual feedback to the user about the current state of the game. The user begins on a title screen (yet to be implemented) and proceeds to choose a game type. The game responds by populating the grid with cells and calling on the cells to count the amount of adjacent bombs for display. From this point, the user enters a basic loop until either he/she wins or loses. The end game state is then returned to the user and the user may make a choice to return to the beginning(title) or restart the game. This loop begins when the user clicks on a cell and calls the cell.reveals() function (figure 4.3). This calls on the cell.show() function which shows the contents of the cell to the user.

```
Cell.prototype.show = function () {
    fill('#1034a6');
    stroke('#f05e23');
    rect(this.x, this.y, this.l, this.l);
    image(this.wave, this.x, this.y, this.l, this.l);

    if (this.flagged) {
        fill('#1034a6');
        rect(this.x, this.y, this.l, this.l);
        image(this.tree, this.x, this.y, this.l, this.l);

    }
    if (this.reveal) {
        if (this.crab) {
            fill('#008ecc');
            rect(this.x, this.y, this.l, this.l);
            image(this.crabs, this.x, this.y, this.l, this.l);
        } else {
            fill('#008ecc');
            rect(this.x, this.y, this.l, this.l);
            fill('orange')
            textAlign(CENTER);
            textSize(28);
            if (this.amtCrabs > 0) {
                text(this.amtCrabs, this.x + this.l / 2, this.y + this.l / 2 + 10);
            }
        }
    }
}
```

Figure 4.7 – Minesweeper cell.show() function

If the User Reveals a bomb, the cell.reveals function calls the lose() function, if the user has successfully cleared all of the cells expect the mines, the cell.reveals() function calls the win() function. This locks all cells and displays the end game state to the user. If neither of those events occur, the user returns to the top of the loop in which he/she chooses a cell.

This structure is mirrored in the following activity diagram:

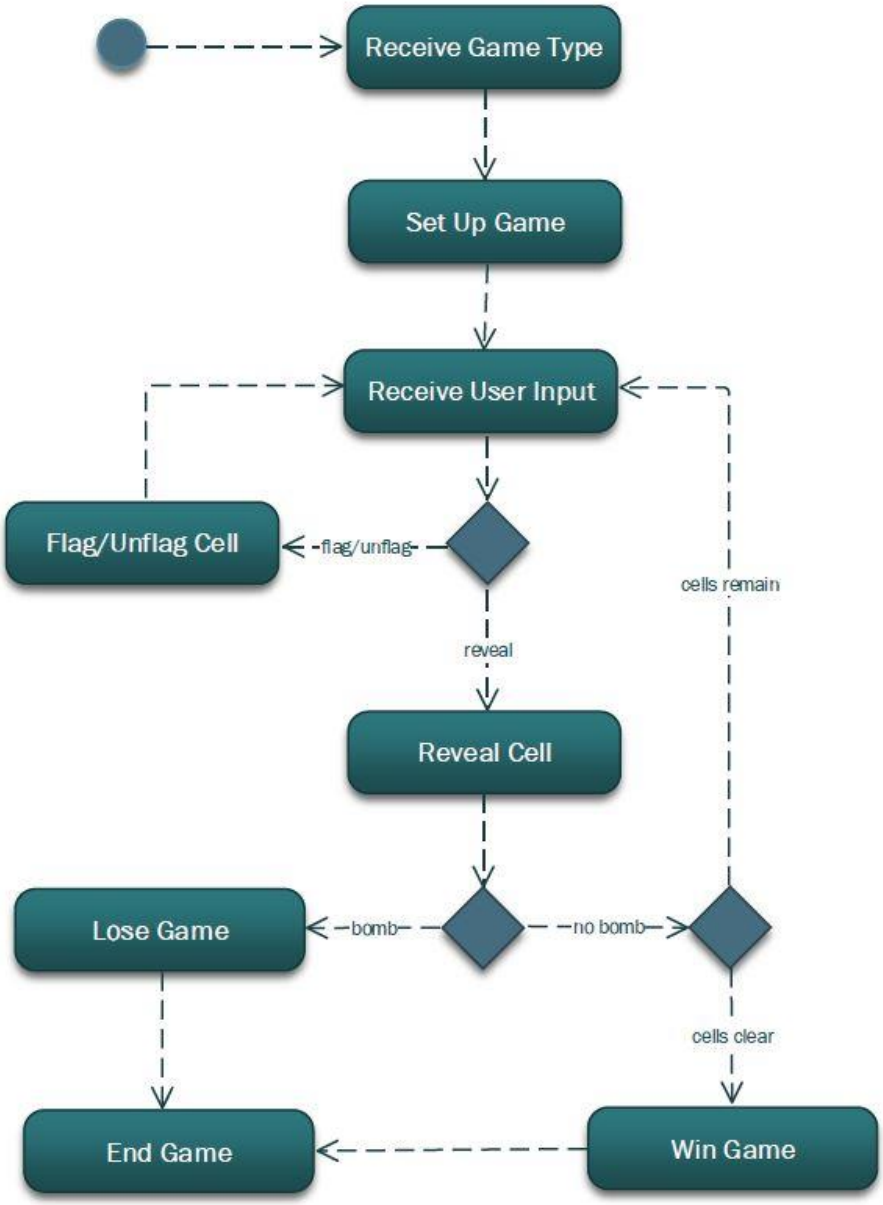

Figure 4.8 – UML activity Diagram

The type of diagram that is an activity diagram allows for different functionality to be observed. The same loop is observable in which the player interacts with the UI, a cell is reveal and the cells are either locked or the user begins a new loop. This diagram however, includes the use of the flag which leads to a separate loop within the original one. The user may opt to enter a *flag* state in which the cell.reveals() function is locked. This state is illustrated above. The cell.flag() Function (figure 4.5) is responsible for locking the cells while in a *flag* state.

5. Persistent Data Management, Security & UI

1. Persistent Data Management

Persistent Data Management is not utilized in this project. Data which might be useful to storage would potentially be high scores or something of the like. In this case, a JSON file could be included along with the implementation. This would guarantee persistent storage across sessions; however, the data would not necessarily be secure. The level of sophistication in this case is exceptionally minimal but it is a thought.

2. Security

Access control and security features have not been included. The code for this implementation will in fact, be free for anyone to browse, alter and study if they so choose. Implementing security and access control for such a basic application is excessive as the application itself lacks sophistication.

3. User Interface

As for User interface, the sophistication of the intended UI design for this project in comparison to the original minesweeper on Windows computers is objectively on par. It doesn't look amazing but neither does the original. The intention is to give the design an interesting theme and charm and this is possible with p5. Presently, the standard minesweeper game has an ocean them and the hex game will likely be a bee hive. Colour sweeper will need some more thought on design once the gameplay mechanics become clearer.

6. Testing

Up until this point, the type of testing utilised has not been formalised. It is essentially trial and error. The entire standard minesweeper game has been tested for errors. Each function has been tested throughout and upon completion and when a completed function or element is introduced to the overall game, the game is tested to ensure successful integration. If something does not work or a problem is caused, the cause of that problem is determined and a solution is developed. This solution is then tested against the entire functionality of the program and once everything is working as intended, a new function will be created and introduced. In this way, new solutions will not affect the efficiency of previous ones.

This is how testing will take place for the future of the project. Perhaps a more formal testing method should be implemented in order to test every possible case within the game project and perhaps

this would make development more efficient. At the rate however, it is reasonable to believe that a durable application will be produced utilising current methods. If it comes to light that current testing methods are unable to fulfil the requirements in terms of playability for the minesweeper game, other testing methods may be implemented to account for this.

7. Analysis of Version Control

The following histogram refers to GitHub Version Control Activity and more specifically, commits throughout time. The histogram illustrates major periods of productivity towards project completion over time. The histogram begins pre-July- 25th and follows through post-August-2nd as this is the full history of GitHub committal.

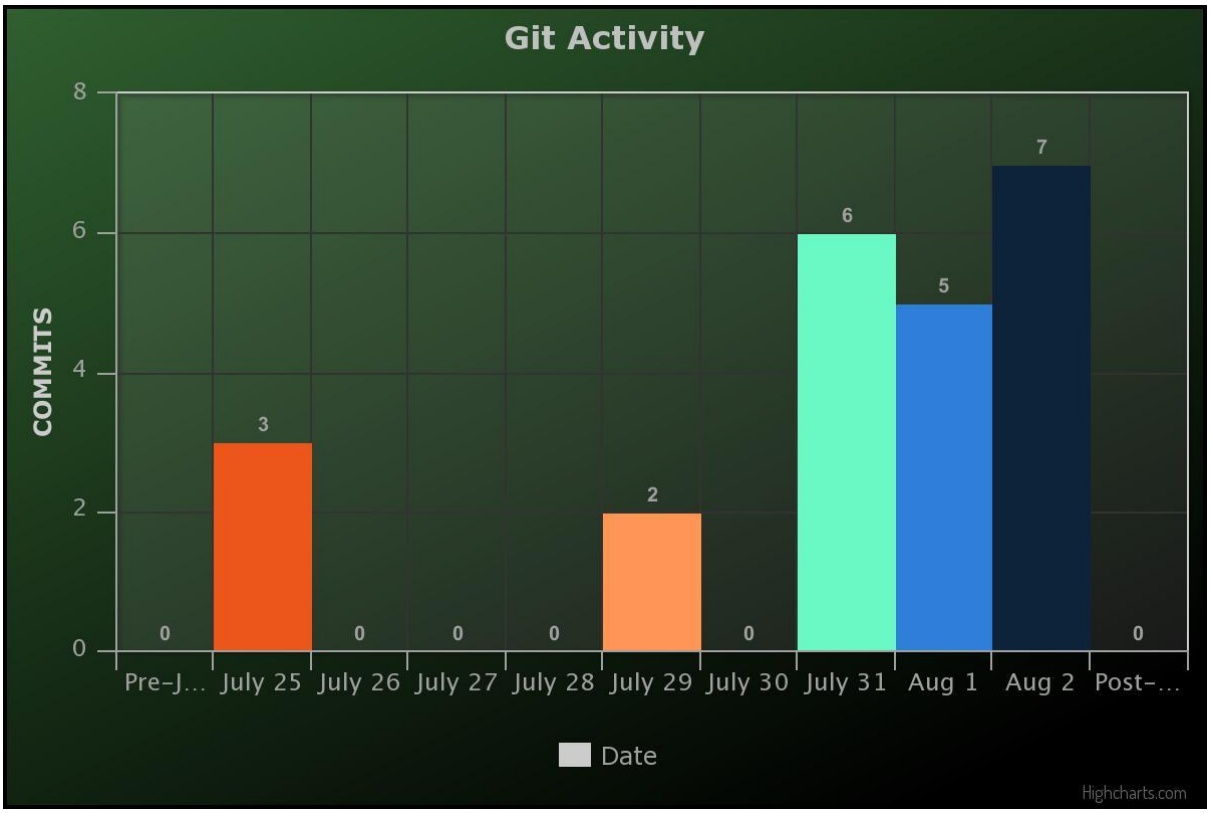

Figure 7.1 – History of GitHub Activity

There appears to be a significant spike in effort according to the history of commits documented. Development of the standard Minesweeper game took place over the space of time between July 25 and August second. No progress toward further development has taken place since this point. That is not to say that significant effort has not gone into software engineering as a whole, simply, other aspects of Engineering have been prioritised over the minesweeper implementation up to this point. This effort has gone towards understanding various Engineering concepts such as modelling, various architectures, many separate development environments and tasks.

With the production deadline approaching, a fresh spike of productivity dedicated to the project is likely to occur. There is still a plethora of effort which will be expended on other concepts surrounding Software Engineering during this development period, however, that will take a back seat to the development of this particular application. Upskilling in software engineering concepts will only benefit the overall project so a lack of time spent on the project specifically does not translate to a lack of productivity overall.

8. Further Modelling & Software Patterns

The following diagrams refer to the design of the project

1. State Machine Diagram

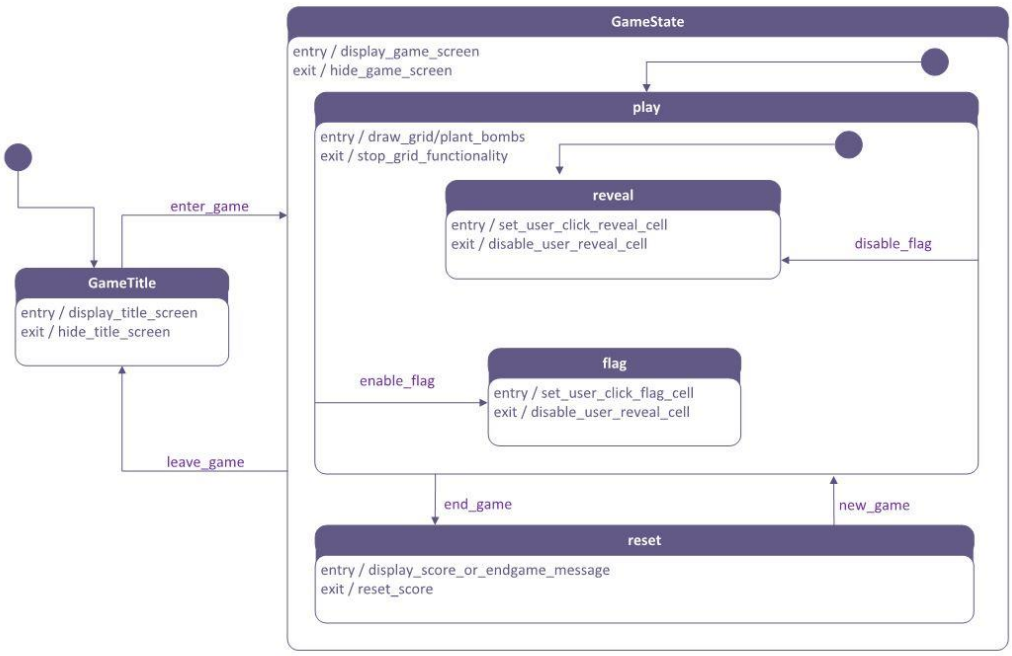

Figure 8.1 – UML State Machine

This state machine refers to the same set of interaction as the previous activity diagram (figure 4.8) and sequence diagram (figure 4.6). This diagram illustrates different states within the game. The overarching states within the application are the game title and the actual game. The title simply offers navigation to the game state. Each game state (standard, hex, color) follows the same structure. The game can be in either a state of play or reset(The time between games). While playing the game, the game can be in one of two states, reveal or flag. These two states influence the way in which a user may interact with the game itself, either revealing cells on click or flagging them

respectively.

2. Collaboration Diagram

The following collaboration diagram illustrates the order in which different objects interact with each other. The interactions are simple and only a few elements are truly interacting.

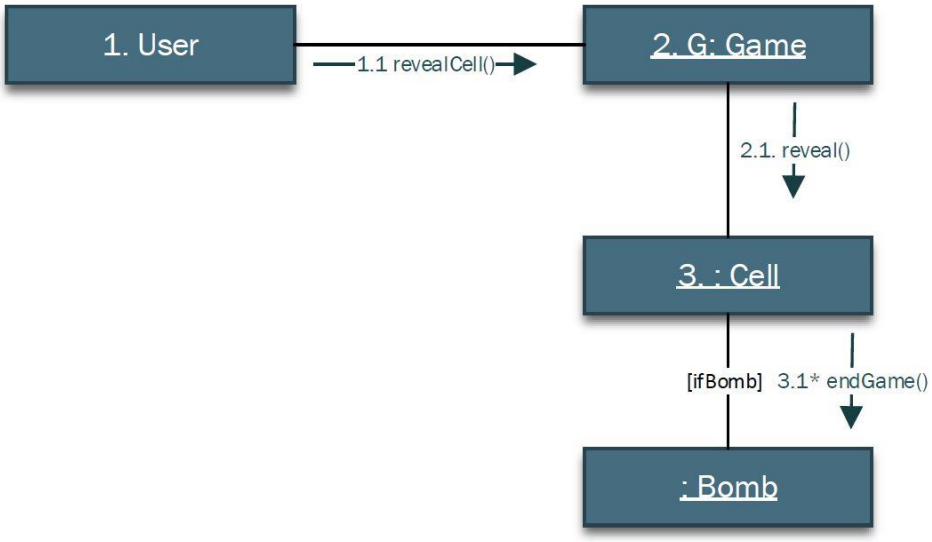

Figure 8.2 – Collaboration Diagram

The collaboration diagram illustrates the order in which different objects interact with each other. The interactions are simple and only a few elements are truly interacting. The user interacts solely with the **Game** class. This class acts as an intermediary between the user and the **Cell** and the **Bomb** class is simply a child of the **Cell.**

3. Relation to Software Patterns

Throughout development (and as referred to in section 7) and through studying Software Engineering Concepts, it is clear that the Software Pattern employed within this project at present is a mess. As described above, The **Game** class does act as an intermediary between the User and the **Cell** class. The game class is displaying a user interface (view) to the User and each cell stores its own data structure. The elements of a Model, View, Controller pattern are evident, however; it is not implemented correctly, nor was it intentional. The **Game** class contains elements of a model, a view and a controller in itself, hence referring to the current software pattern as a mess. There is however, potential within that mess.

```
function Cell(i, j, l) {
    this.crabs = crabcrab;
    this.tree = treetree;
    this.wave = wavewave;
    this.i = i;
    this.j = j;
    this.x = (i * l) + 20;
    this.y = (j * l) + 90;
    this.l = l;
    this.crab = false
    this.reveal = false;
    this.flagged = false;
    this.amtCrabs;
}
```

Figure 8.3 – Minesweeper Cell Class

Each Cell stores all of the data necessary for it to be displayed, changed its state, have knowledge of its own position in the grid along with the contents of neighbour cells and also know the content of itself. It also stores a range of methods to be called upon.

```
let flagger;
let cols;
let rows;
let l = 50;
let grid;
let allCrabs = 20;
let gg = false;
let score = 0;
let start = false;
let revealCount = 0;
let gridSize = 0;
```

```
cols = 10; //floor(width / l);
rows = 10; //floor(height / l);
grid = makeField(cols, rows);
for (let i = 0; i < cols; i++) {
    for (let j = 0; j < rows; j++) {
        gridSize++;
        grid[i][j] = new Cell(i, j, l);

    }
}
```

```
    for (let i = 0; i < cols; i++) {
        for (let j = 0; j < rows; j++) {
            grid[i][j].show();
        }
    }
}
```

Figure 8.3 – Minesweeper Game Class

The **Game** class stores assorted pieces of data related to the game board and creates a grid within it and then populates that grid with Cells. The **Game** class then calls upon the method of cell.show() (figure4.7). This means that each individual cell is displaying itself. Furthermore, when the user clicks on the game board, the **Game** function receives that click and alters the data within the **Cell** which then alters the view in real time.

```
function mouseClicked() {
    start = true;
    flagger.click();
    for (let i = 0; i < cols; i++) {
        for (let j = 0; j < rows; j++) {
            if (grid[i][j].click(mouseX, mouseY)) {
                grid[i][j].flag();
            }
        }
    }
}
```

Figure 8.4 – Minesweeper Game Click Function

If user clicks on a particular grid location, cell.flag() function (figure 4.5) is called. This checks if Cell is locked by a flag and if not, changes the state of that cell to revealed.

This is not a true MVC pattern, however, there are many elements of MVC there. If the UI acts as a view, the **Game** Class acts as the controller and the **Cell** class as the model which stores the data structure about itself, a pattern does begin to emerge. With more refinement, it will be possible to make this into a true Model, View, Controller pattern.