*3813ICT – Software Frameworks*

*Milestone 3 - Minesweeper*

Student: Zak Barker

Student#: S5085150

Subject: 3815ICT Software Engineering

Email: Zak.Barker@Griffithuni.edu.au

* 1. ***Design Principals***

Least privilege is a software design principal which relates to access rights, as does a fail-safe default. Least privilege sees that any entity should only be granted the least possible privilege necessary for that entity to complete its job. Fail-safe defaults is the idea that an entity must be given explicit access to an object, otherwise it is denied access to this object. The two work in conjunction as a method of preventing unnecessary access to promote the integrity of the system as a whole. Encapsulation is the process of encapsulating separate components of a system so that they are independent of one another. This leads into the ideas of coupling and cohesion. Coupling is the interdependency between two components and this should be minimized as a failure of one component should ideally - not affect the functionality of another. Cohesion describes the effective relationship between operations within a specific module. These concepts encourage information hiding. This is the concept of keeping the internal functions of an interface invisible to an entity that does not require the knowledge. The entity should simply know how to operate the interface as opposed to knowing how it works internally.

* 1. ***Design Process***

The first step in the design process for the Minesweeper implementation was prototyping. This involved created an initial version of the game to facilitate tests, development and functionality. Prototyping is a critical aspect of the design process and a working prototype was the first goal. Initially, Models were created based on conceptual designs for minesweeper, however, over the course of implementation, these models became unsatisfactory for functionality. The process became more of an agile approach in which pieces were modelled, built, tested and then integrated into the main project. This was a small scale implementation but the principals were essentially agile and a new overall model resulted from this along with an improvement in efficiency. The end result was an overall idea for software design and architectural design. This also helped with the process of identifying which pieces could be decoupled and aided in deciding which functions should be related and which functions could be factored out and re-used.

* 1. ***System Models***

Ahead is an image of the original class model used for the creation of this project:

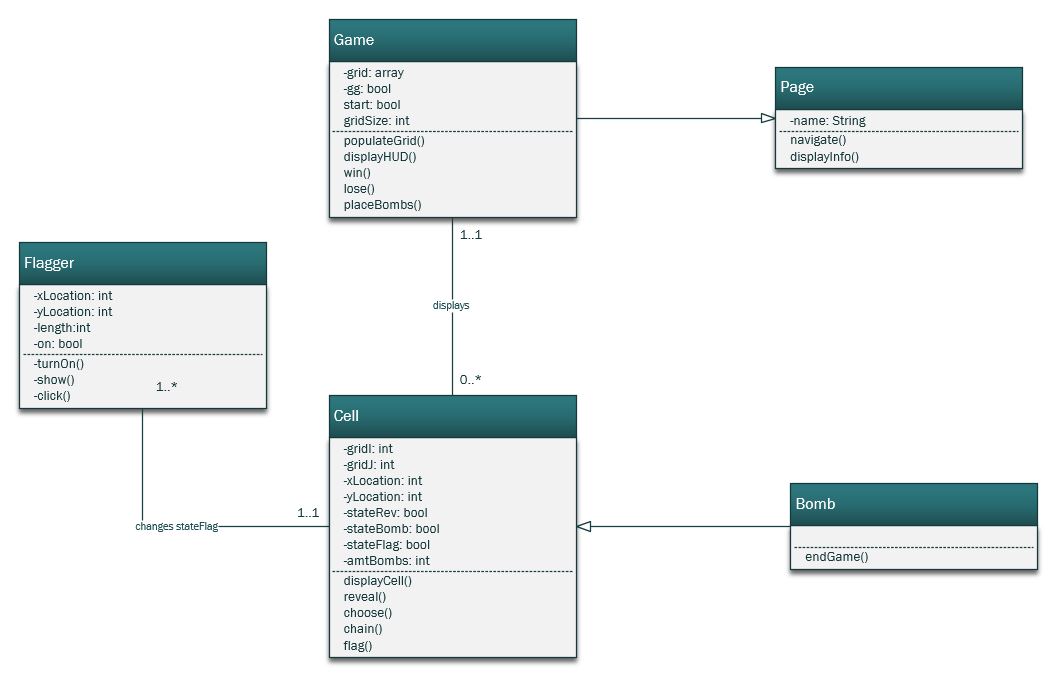


Figure 1 – Original Class Diagram

This was revised throughout the software design process in order to facilitate cohesion and reduce coupling. The design has become slightly more complex but allowed for separation of responsibilities across various aspects of the implementation. The new design is as follows

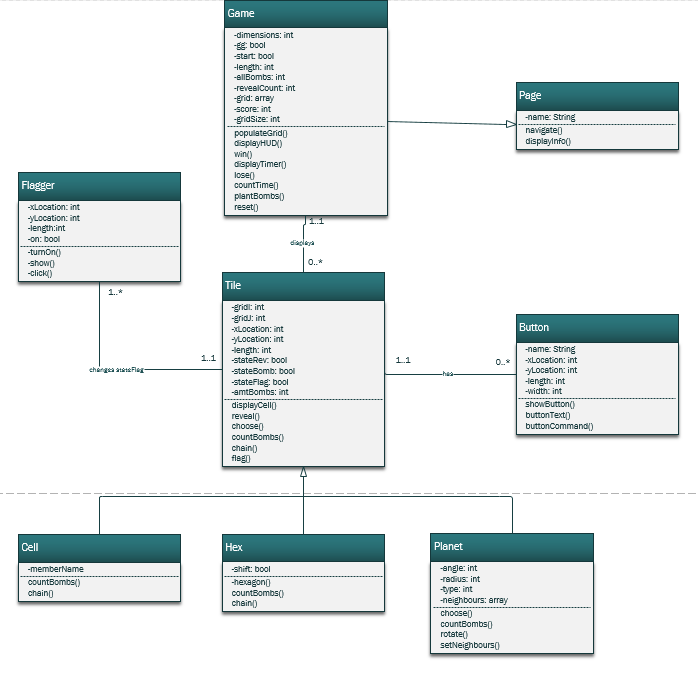


Figure 2 – Revised Class Diagram

The tile types were separated into their respective subclasses of the Tile. The common methods and members were factored out so that the subsequent subclasses would be able to hold information crucial to that type of tile. For instance, in a hex game, each tile row needs to know if it is odd or even. Furthermore, slightly different algorithms were necessary for the same function in different tile types. In this case, the original can be overridden within the subclass. This adds to a limiting of coupling between tiles and the gameState. This design is much more efficient than the original class structure. The Game class has been extended to include more functionality. A button class has been introduced to disconnect the Game class from the navigational aspects of the software. This is now handled separate from the game itself. Finally, the Game itself has been modified to hold all of the rules for the game in separate functions along with all of the information required for a game state. Each method pertains to a rule within the game and the members facilitate this. These changes also promote cohesion amongst each Class.

* 1. ***Design Paradigm***

As illustrated above, the Tile module has been decomposed into separate subclasses. As a design paradigm, the original idea was to decompose the Cell class into separate modules. Each module would be used for functionality which could be used across tiles – ie, the show() function or the reveal() function. This would have achieved a similar purpose as breaking the Tile into three separate cell types for each game. The software has been modularised in other areas for functionality. For instance, there is a module which purely handles the background display of the project and another module which governs routing. Another area for decomposition was in the introduction of the Button Class as previously, the functionality was handled within the pages, this responsibility has been separated so that the Page class needs only to worry about the functionality of the page. These modularisations promote usability in the functions as well, they can be used across pages for the specific requirements of that resource.

* 1. ***Software Architecture***



Figure 3 – Software Architecture

The software architecture chosen to deliver this project is Javascript as the programming language, with p5 library integrated. The IDE chosen was brackets and Chrome Browser has been used as a user interface. This is a free software architecture and ideal for a small project such as a minesweeper implementation, using this architecture is quite suitable. The reason being that minesweeper is not an incredibly taxing project on the CPU. There is not a lot of power necessary to deliver this project so there is no need to use a more powerful architecture to deliver it. The p5 library provides all of the necessary functions to deliver a satisfying GUI and Javascript is able to handle the back-end programming requirements. It is a very simple architecture to deliver a very simple project.

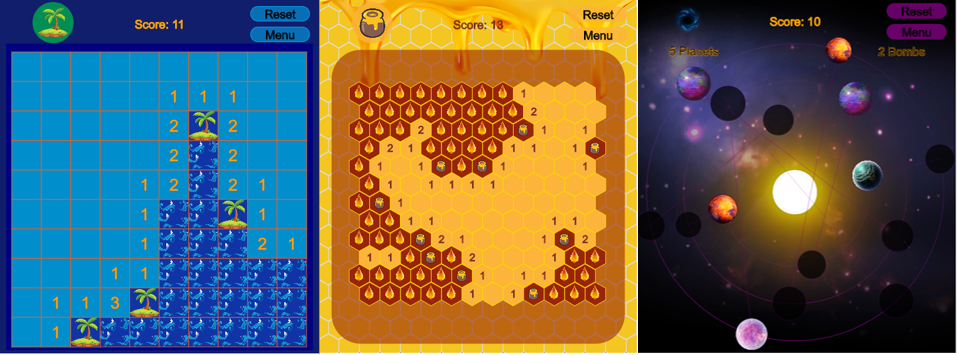


Figure 4 – GUI provided by Architecture

In terms of ease of use, the architecture is simple to implement and simple to use. The advantages over other architectures lies in simplicity. It is simple to develop a satisfying system with this architecture which leaves room for exploration of concepts and the display factor is highly flexible. With the ability to create a UI design easily, it is possible to add gameplay elements which rely on this design – for instance – the rotation nodes in the Planet Minesweeper implementation. This would be much more difficult to implement with other more complex architectures. Selecting the correct architecture for a given project comes down to deciphering the requirements of the project and providing a platform which is powerful enough to meet the requirements as well as flexible enough to explore those requirements.

* 1. ***Software Programs & Design Patterns***

Design patterns such as that of an observer interface are useful while implementing various system architectures such as one with a client – server framework. The minesweeper implementation follows a Model-View Controller pattern. The view in this instance is the GUI and this is provided by a single Javascript file within the project. The functionality of this file is to display the view to the user and there is no data stored. The routes are handled by a controller and these designate which pages will display which data. Implementation of these routes is in a simple form to facilitate ease of navigation and information flow. The Data Structure of the system lives within each class set-up and the controller is used to initialise these classes and store the data for the objects separately. These objects are then called by the controller and displayed by the view.

* 1. ***User Interface***

The user interface for this project uses an event-driven design structure. The system waits for an event (being a mouse click) from the user and enacts on this event. There are multiple possible mouse interactions on each page and the system needs to be able to handle these in any order. As such, the mouse is clicked and if a component registers that it has been clicked on, it registers that event and acts.

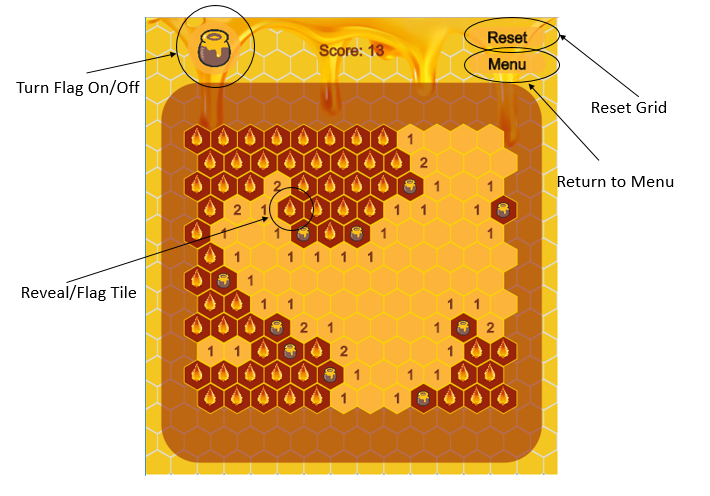


Figure 5 – Events in Minesweeper

There are a few different possible functions a user can interact with on click. The system however, must listen for an interaction with any tile, along with the outside options (reset, menu, flag). The system must wait for user input (an event) before returning the required data in response and updating the view.

* 1. ***Model-View Controller***

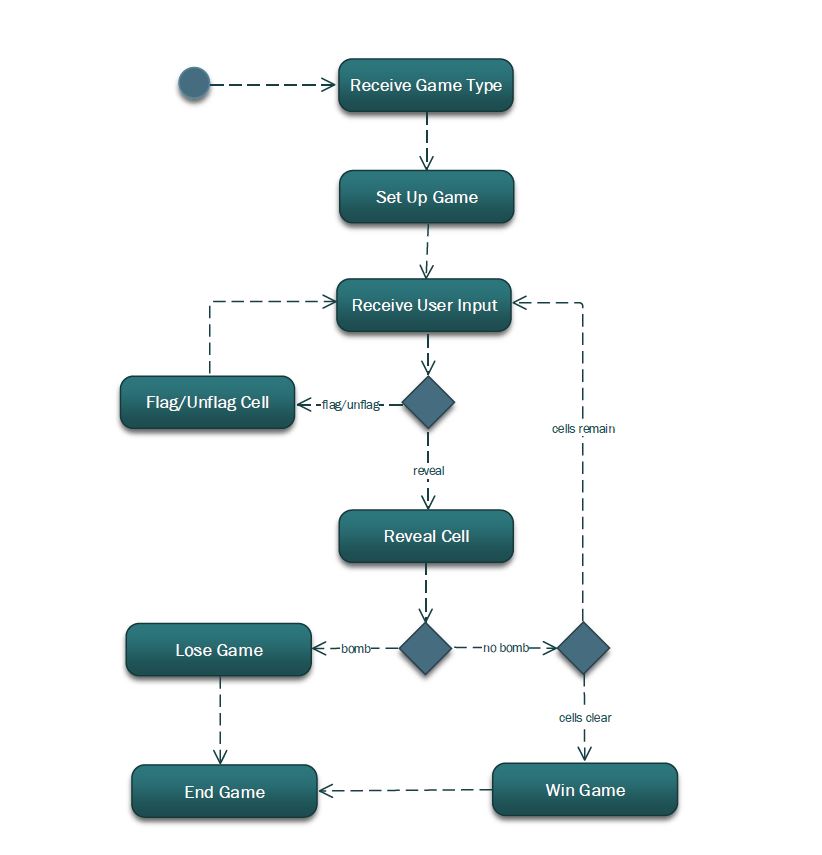
Model, View Controller paradigm is a way to separate responsibilities between separate sections of an interface. The view refers to the Interface which is ‘viewed’ by the user. The model contains the data and the structure of that data which is utilised by the interface. Finally, the controller facilitates the flow of data between both the model and the view. The separation of concerns between different sections of the system means that each element may act more efficiently and when implemented correctly, provides a smooth flow of data to and from the interface. The importance of this is in the decoupling of core components in the architecture which makes code more reusable.

The minesweeper project implements MVC on a small scale. There is a specific file which acts as the view. This is updated based on input by the user and runs through another file which handles routing between screens and transfer of data between the objects stored and the view for the interface. The model in this case would be the class structure (figure 2) and the controller is the portion of code which bridges the two.

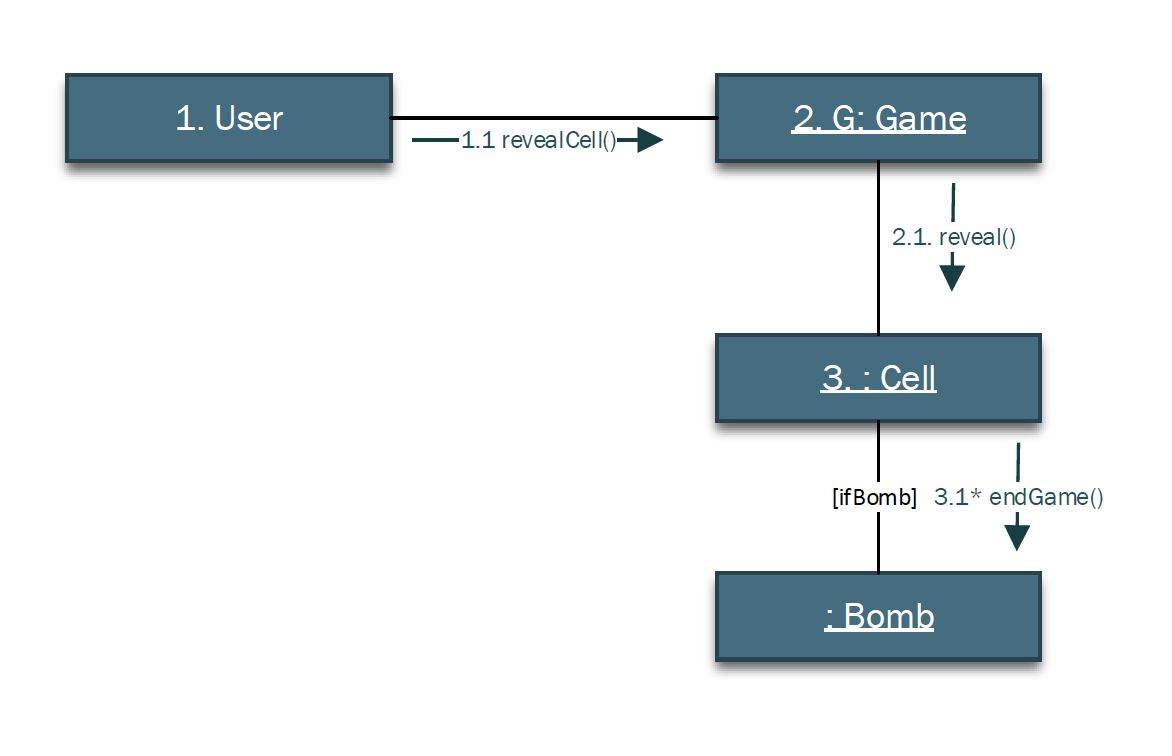
* 1. ***Good Software Design***

The aim of good software design is to provide the correct functional requirements of a project as efficiently as possible. This means minimizing code, structuring code and organising code so that it follows principals described earlier in this document (low coupling, high cohesion, emphasis of design patterns etc). This is one tenet of good software design, however, programming can be a collaborative effort. With this in mind, it is important to emphasise readability into one’s programming so that any would-be collaborator will have a smooth transition into working on that code. Commenting is an essential aspect of this within the code, however, accompanying documentation is one of the most effective ways to clarify the structure and use of a software implementation. Associated documentation serves the purpose of clarifying the structures, requirements, dynamic functionality and processes within a given design. It is a level of abstraction away from the actual implementation and helps to understand the project as a whole. This documentation includes modelling, diagrams, written structures, requirements among other forms.

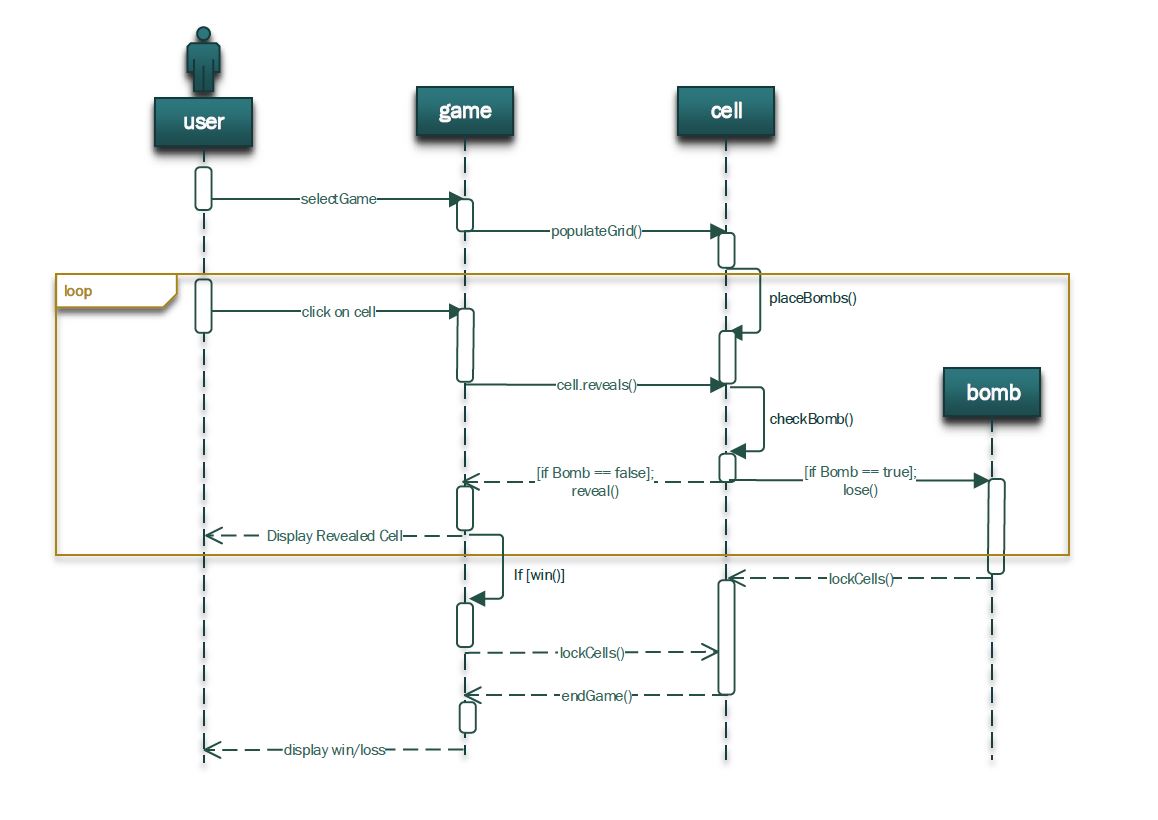
* 1. ***Design Documentation (Appendix)***



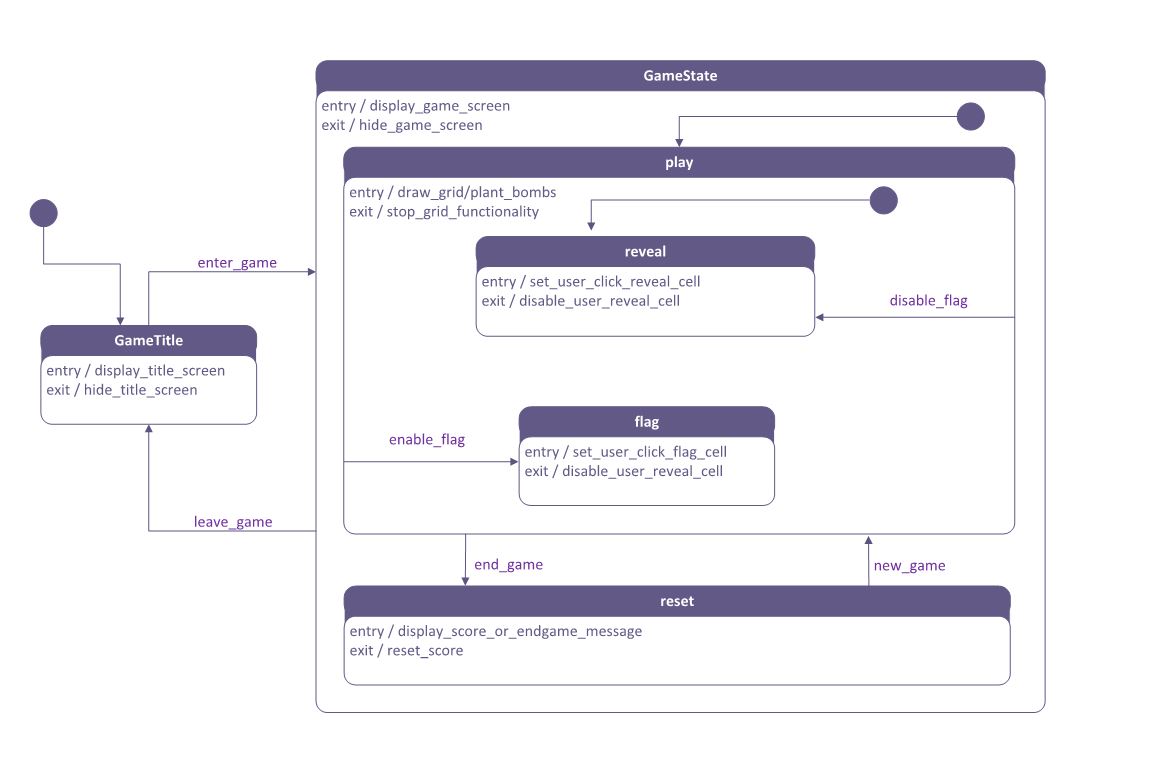
Appendix 1 – Activity Diagram



Appendix 2 – Collaboration Diagram



Appendix 3 – Sequence Diagram



Appendix 4 – State Diagram  
Appendix 5 – Use Case Diagram

