Overall Project Documentation

2805ICT Principles of Software Engineering

Damon Murdoch (s2970548)

Click Link for Demonstration Video

Table of Contents

Video link

Video link

1.0 Application of Software Principles

1.1 Principle of Least Privilege

This software project enforces the principle of least privilege by ensuring that at no point is any user, or any software component accessing a section of the application which is not critical to that component's purpose. This is critical to the operation of the program, as it provides a greater level of security to the application, and ensures that users and functions cannot see or make changes to variables or objects they should not. One key software example of this is reflected in is the 'check_solution' function in ms.py, which is used to check if a board is in a valid solution state.

1.1.1 Source Code Example - check_solution() from ms.py

```
def check_solution(grid):
   for i in grid:
       for j in i:
           if j.get_bomb():
               if j.get_flagged() == False:
                   print("Bomb not covered!",i,j)
                   return False
   return True
```

This function only accesses functions and data which it needs to perform its required task, and as a result follows the principle of least privilege.

1.2 Principle of Economy of Mechanism

This project follows the principle of economy of mechanism to ensure that the project does not become unnecessarily complicated, and so that it can be interpreted by other programmers without unreasonable difficulty. In addition to this, the more complex a system becomes the higher the risk of security vulnerabilities are. One way this is implemented into the project is by breaking reusable sections of the application up into functions or classes - so that they can be reused or abstracted away to make code easier to understand or modify. One example of how this is implemented in the final project is the left_click_grid function, which is utilised when a block is clicked to uncover all surrounding blocks recursively if it has no adjacent bombs. This algorithm can be implemented using an iteration structure, however it is much simpler to implement and understand if it is written using a recursive method, as seen below.

Video link

1.2.1 Source Code Example - left_click_grid() from ms.py

```
def left_click(grid,x,y):

   grid_size = len(grid)

   if grid[y][x].get_covered():
       if grid[y][x].get_bomb():
           return -1
       if grid[y][x].get_adjacent() > 0:
           grid[y][x].uncover()
           # implement game winning code here

       if grid[y][x].get_adjacent() == 0:
           grid[y][x].uncover()

           if (y > 0):
               if(not grid[y-1][x].get_flagged()):
                   left_click(grid,x,y-1);

           if (x > 0):
               if (not grid[y][x-1].get_flagged()):
                   left_click(grid,x-1,y);

           if (y < grid_size - 1):
               if not grid[y+1][x].get_flagged():
                   left_click(grid,x,y+1)

           if (x < grid_size - 1):
               if not grid[y][x+1].get_flagged():
                   left_click(grid,x+1,y)

           if (y > 0 and x > 0):
               if not grid[y-1][x-1].get_flagged():
                   left_click(grid,x-1,y-1)

           if (y < grid_size - 1 and x < grid_size - 1):
               if not grid[y+1][x+1].get_flagged():
                   left_click(grid,x+1,y+1)

           if (y < grid_size - 1 and x > 0):
               if not grid[y+1][x-1].get_flagged():
                   left_click(grid,x-1,y+1)

           if (y > 0 and x < grid_size - 1):
               if not grid[y-1][x+1].get_flagged():
                   left_click(grid,x+1,y-1)
```

Video link

1.3 Principle of Information Hiding

The principle of information hiding was implemented in this project by ensuring that all communication between classes is performed using functions such as getter-setter functions, or other functions related to the interface objects. This is to ensure that modifications to the backend implementation of the application do not have unintended effects on the rest of the application, and to ensure that when any variable from the objects is accessed, any additional code which is required to run in the function is performed. One key pieceof software where information hiding is implemented is the colour_square class, for which the source code is displayed below.

1.3.1 Source Code Example - colour_square() from cs.py

```python
class colour_square():
   def __init__(self):
       r = rand.randint(0,2)
       if r == 0:
           self.colour = "red"

       if r == 1:
           self.colour = "yellow"

       if r == 2:
           self.colour = "blue"

       self.is_flagged = False
       self.is_triggered = False
       self.is_covered = True
       self.adjacent = 0

   def get_triggered(self):
       return self.is_triggered

   def set_triggered(self):
       self.is_triggered = True

   def get_covered(self):
       return self.is_covered

   def uncover(self):
       self.is_covered=False

   def get_flagged(self):
       return self.is_flagged

   def set_flagged(self):
       self.is_flagged = not self.is_flagged
```

Video link

```
    def set_adjacent(self,adj):
        self.adjacent=adj
        pass

    def get_adjacent(self):
        return self.adjacent

    def get_colour(self):
        return self.colour

    def __str__(self):

        if self.get_flagged():
            return "F"

        if self.get_covered():
            return " "

        return str(self.get_adjacent())

    def __repr__(self):

        if self.get_flagged():
            return "F"

        if self.get_covered():
            return " "

        return str(self.get_adjacent())
```

Video link

1.4 Principle of Reducing Coupling

The principle of reducing coupling is implemented in this project by ensuring that functions which serve multiple purposes are split into separate functions, and values subject to change are stored in variables rather than hard-coded into the program. This is to ensure that program code is reusable, and software components do not have to be frequently rewritten as the application is developed. One example of how this has been implemented into the project can be seem in the popup_window function. The function originally had quite high coupling, and was not reusable for other purposes. However, it was later modified so that it could be reused for multiple purposes without needing to be rewritten.

1.4.1 Source Code Example - popup_window() (original)

```
def popup_window(text,time):
   toplevel = tk.Toplevel()
   toplevel.focus_force()
   # Lost the Game
   if wl == False:
       label1=tk.Label(toplevel,text="You Lose! Time Taken: "+str(time) + "
seconds.",height=0,width=50)
       label1.pack()
   # Won the Game
   else:
       label1 = tk.Label(toplevel, text="You Win! Time Taken: " + str(time) + "
seconds.", height=0, width=50)
       label1.pack()
```

1.4.2 Source Code Example - popup_window() (implementation)

```
def popup_window(lbl,time=None,):
   toplevel = tk.Toplevel()
   toplevel.focus_force()
   # Lost the Game
   msg = lbl
   if time != None:
       msg += " Time Taken: " + str(time) + " seconds."
   label1 = tk.Label(toplevel, text=msg, height=0, width=50)
   label1.pack()
```

Video link

The first version of this piece of software could only be implemented for two purposes, and this was displaying whether or not the player won or lost. The second version can be reused to display other messages to the screen, and only displays a time variable if one is submitted by the user. This update also ensures that if minor changes are made to the operation of the message function, the basic usage will stay the same and its usage will not need to be modified in other places of the code.

1.5 Principle of Maximising Cohersion

There are a number of strong examples of cohesion in this application. All of the interface objects in the application are contained in a ‘master’ class for their window, and these classes contain all of the functions which are required to make the interface for that window operate. In addition to this,the game algorithm codes are stored in their own files, although Standard Minesweeper and Hexagon Minesweeper share the same file as their algorithms are functionally similar. Colour-Based minesweeper is stored in its own file, as its data structures and gameplay is too different to be cohesive. The source code displayed below is the basic structure of the ‘Vanilla’ class, which contains the interface resources for the Standard Minesweeper game.

1.5.1 Source Code Example - vanilla() from main.py

```
class Vanilla(Page):

  def decide_sprite(self,y,x):
      ...

  def __init__(self, *args, **kwargs):
      ...

  def start_timer(self):
      ...

  def get_time(self):
      ...

  def start(self,grid,bomb):
      ...

  def load_game_board(self):
      ...

  def reload_game_board(self):
      ...

  def handle_left_click(self,j,i):
      ...
```

Video link

```
  def handle_right_click(self,j,i):
      ...

  def gameover(self,j,i):
      ...
```

All of the other interface objects follow a similar pattern, however the specific functions perform different tasks for each implementation of the game. This demonstrates high cohesion as all of the software components in each interface class follow a common purpose, which is to manage the interface and game state for each different implementation of the game.

1.6 Principle of Separation of Concerns

The principle of separation of concerns is implemented to ensure that all separate objects are designed to perform a separate purpose. This principle is also demonstrated in the implementation of the interface objects, as all of them are designed to operate on a different form and their functions and objects do not interact. Outside of the main form object, all of the other interface objects are completely separate and have no interfaces with each other. This is to ensure that all objects are self-contained, and any changes made to those specific objects do not have adverse effects on the other game implementations. The following section of code will display a stripped version of the interface objects source code to demonstrate this principle.

```
class Vanilla(Page):

  def decide_sprite(self,y,x):
      ...

  def __init__(self, *args, **kwargs):
      ...

  def start_timer(self):
      ...

  def get_time(self):
      ...

  def start(self,grid,bomb):
      ...

  def load_game_board(self):
      ...

  def reload_game_board(self):
      ...

  def handle_left_click(self,j,i):
      ...
```

Video link

```
  def handle_right_click(self,j,i):
      ...

  def gameover(self,j,i):
      ...

class Hexagon(Page):

  def start_timer(self):
      ...

  def get_time(self):
      ...

  def __init__(self, *args, **kwargs):
      Page.__init__(self, *args, **kwargs)
      ...

  def decide_sprite(self, y, x):
      ...

  def __init__(self, *args, **kwargs):
      ...

  def start(self, grid, bomb):

      ...

  def load_game_board(self):
      ...

  def reload_game_board(self):
      ...

  def handle_left_click(self,j,i):
      ...

  def handle_right_click(self,j,i):
      ...

  def gameover(self,j,i):
      ...

class Colours(Page):

  def start_timer(self):
      ...

  def get_time(self):
```

Video link

```
        ...

    def decide_sprite(self, y, x):
        ...

    def __init__(self, *args, **kwargs):
        Page.__init__(self, *args, **kwargs)
        ...

    def start(self, grid):
        ...

    def load_game_board(self):
        ...

    def reload_game_board(self):
        ...

    def handle_left_click(self, j, i):
        ...

    def handle_right_click(self, j, i):
        ...

    def gameover(self, j, i):
        ...

class HighScores(Page):
    def __init__(self, *args, **kwargs):
        Page.__init__(self, *args, **kwargs)
        ...

    def search(self,gamemode,grids,bombs):
        ...

class MainView(tk.Frame):
    def __init__(self, *args, **kwargs):
        tk.Frame.__init__(self, *args, **kwargs)
        p1 = Vanilla(self)
        p2 = Hexagon(self)
        p3 = Colours(self)
        p4 = HighScores(self)

       ...

        p1.place(in_=container, x=0, y=0, relwidth=1, relheight=1)
        p2.place(in_=container, x=0, y=0, relwidth=1, relheight=1)
        p3.place(in_=container, x=0, y=0, relwidth=1, relheight=1)
        p4.place(in_=container,x=0,y=0,relwidth=1,relheight=1)
```

Video link

```
b1 = tk.Button(buttonframe, text="Standard", command=p1.lift)
b2 = tk.Button(buttonframe, text="Hexagon", command=p2.lift)
b3 = tk.Button(buttonframe, text="Colours", command=p3.lift)
b4 = tk.Button(buttonframe, text="High Scores",command=p4.lift)

b1.pack(side="left")
b2.pack(side="left")
b3.pack(side="left")
b4.pack(side="left")

p1.show()
```

This section of code demonstrates that all of the interface objects have separated concerns, and the only interface object which references the others is the MainView object, which is utilised as a main page to display the other interface forms. Many sections of code in these interface objects are reused, and while this introduces a certain level of redundancy it is important that different implementations of each game have separate implementations, so that alterations made to the code for one version of the game do not cause adverse side effects on other implementations.

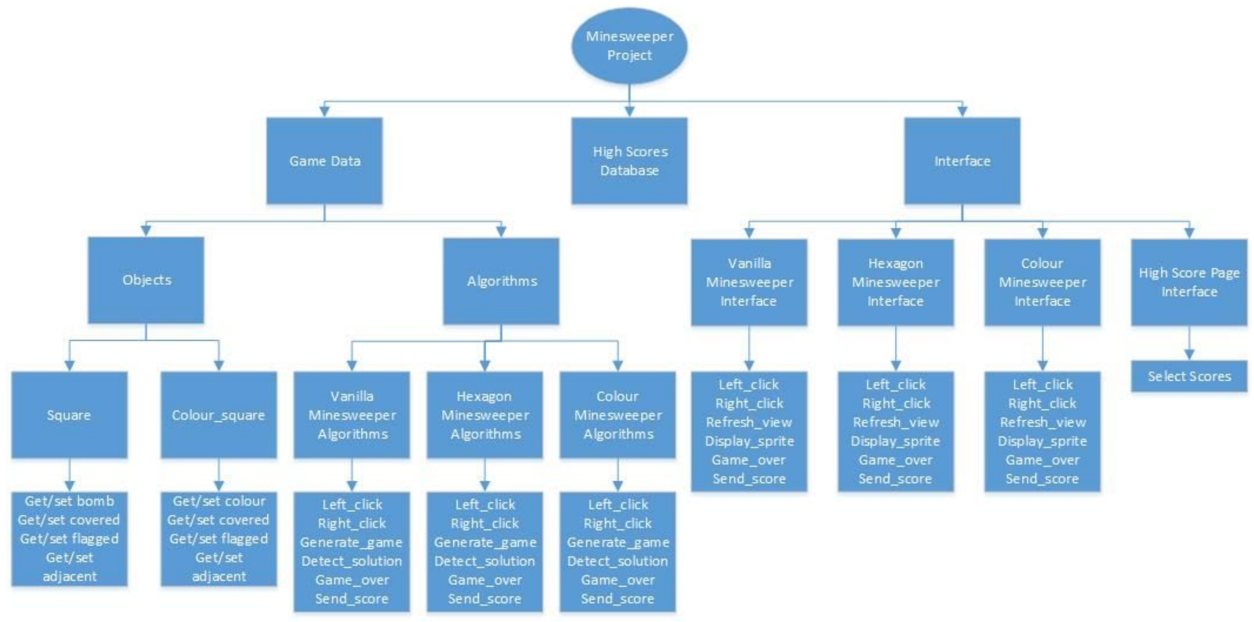
1.7 Principle of Psychological Acceptability

The principle of psychological acceptibility is implemented in this project to ensure that the project is as simple to use as possible. Interface objects are designed to be simple to be user-friendly and simple to interact with, so that users do not feel like tasks are being made more difficult than necessary. There are no specific snippets of code which can be used to demonstrate this, however this principle will be demonstrated in the video accompanying this report.

Video link

2.0 Design Methodology

The design methodology utilised for this project was the agile design methodology, as the project was broken down into sections and developed incrementally. This is due to the fact that agile development supports requirement changes mid-project, and allowed for changes to be made to older sections of the software due to requirements from newer sections. See below for a basic diagram of all the subsystems which were identified for this project.


Minesweeper Project
Game Data
High Scores Database
Interface
Objects
Algorithms
Vanilla Minesweeper Interface
Hexagon Minesweeper Interface
Colour Minesweeper Interface
High Score Page Interface
Square
Colour_square
Vanilla Minesweeper Algorithms
Hexagon Minesweeper Algorithms
Colour Minesweeper Algorithms
Left_click Right_click Refresh_view Display_sprite Game_over Send_score
Left_click Right_click Refresh_view Display_sprite Game_over Send_score
Left_click Right_click Refresh_view Display_sprite Game_over Send_score
Select Scores
Get/set bomb Get/set covered Get/set flagged Get/set adjacent
Get/set colour Get/set covered Get/set flagged Get/set adjacent
Left_click Right_click Generate_game Detect_solution Game_over Send_score
Left_click Right_click Generate_game Detect_solution Game_over Send_score
Left_click Right_click Generate_game Detect_solution Game_over Send_score

2.1 Development Process Outline

See below for the basic progress breakdown for each iteration followed during the development of the software project.

Iteration 1 - Vanilla Minesweeper Game Code Implementation

Section: Game Data

Objects: Square
Algorithms: Vanilla Minesweeper Algorithms

Iteration 2 - Vanilla Minesweeper Interface Basic Implementation

Section: Interface

Objects: Vanilla Minesweeper Interface
Functions: Left_click,right_click,refresh_view,display_sprite,game_over

Video link

Iteration 3 - Hexagon Minesweeper Game Code Implementation

Section: Game Data

Algorithms: Hexagon Minesweeper Algorithms

Iteration 4 - Hexagon Minesweper Interface Basic Implementation

Section: Interface

Objects: Hexagon Minesweeper Interface
Functions: Left_click,right_click,refresh_view,display_sprite,game_over

Iteration 5 - Colour Minesweeper Game CodeImplementation

Section: Game Data

Objects: Colour_Square
Algorithms: Colour Minesweeper Algorithms

Iteration 6 - Colour Minesweeper Interface Implementation

Section: Interface

Objects: Hexagon Minesweeper Interface
Functions: Left_click,right_click,refresh_view,display_sprite,game_over

Iteration 7 - High Scores Database Implementation

Section: High Scores Database

Functions: Insert_Score,Select_Scores,Create_Connection

Section: Interface

Objects: High Score Interface
Functions: Select_Score

Video link

2.2 Design Patterns

There were several design patterns utilised in the development of this application.

2.2.1 Creational Patterns

There are several creational patterns implemented in this project. The first creational pattern is the lazy initialisation pattern, which involves delaying the creation of an object or calculation of values until it is needed. This is performed in several parts of the application, where a global variable is initialised with an arbitrary value and then assigned an actual important value the first time it would be needed. A sample of code where this can be seen is in standard minesweeper's init and start functions.

2.2.1.1 Lazy Initialisation Source Code Example - main.py

```
def __init__(self, *args, **kwargs):
   Page.__init__(self, *args, **kwargs)
   ...
   self.game = None
   ...
def start(self, grid):
   if self.game != None:
       self.gamewindow.pack_forget()
       self.gamewindow.destroy()

   self.game = cs.generate_board_colours(grid)
   ...
```

This creational pattern is used to ensure the reusability of the start function so that it knows when the game has been just started, or when the game is being restarted after a previous session. The next creational pattern included in this project is the singleton class structure which is utilised for all of the interface menu forms such as Vanilla, Hexagon, Colours, High Scores and MainView. Only a single instance of these classes is ever created, and they can be referenced from any location in the program. The following section of source code demonstrates how singleton patterns are utilised in this project.

2.2.1.2 Singleton Source Code Example - Main.py

```
class Vanilla(Page):
  ...

class Hexagon(Page):
  ...

class Colours(Page):
  ...

class HighScores(Page):
   ...

class MainView(tk.Frame):
   def __init__(self, *args, **kwargs):
       tk.Frame.__init__(self, *args, **kwargs)
       p1 = Vanilla(self)
       p2 = Hexagon(self)
       p3 = Colours(self)
       p4 = HighScores(self)
       ...

if __name__ == "__main__":
   ...
   main = MainView(root)
   main.pack(side="top", fill="both",expand=True)
   ...
```

Video link

3.0 Model-View-Controller Pattern

The model-vew-controller pattern is utilised for this application to separate presentation and interaction from the actual system data. This is important because it allows for data and representation to change independently. The application components have been separated into categories as follows:

Model - Backend Software Operations

Inputs:

Game state update data, Menu select data, Score select data

Outputs:

Game state data, Scores data

View - User Interface Display

Inputs:

Data: Game state data, Scores data

Outputs:

Display: Game state data, Scores data, Menu UI

Controller - User Input

Inputs:

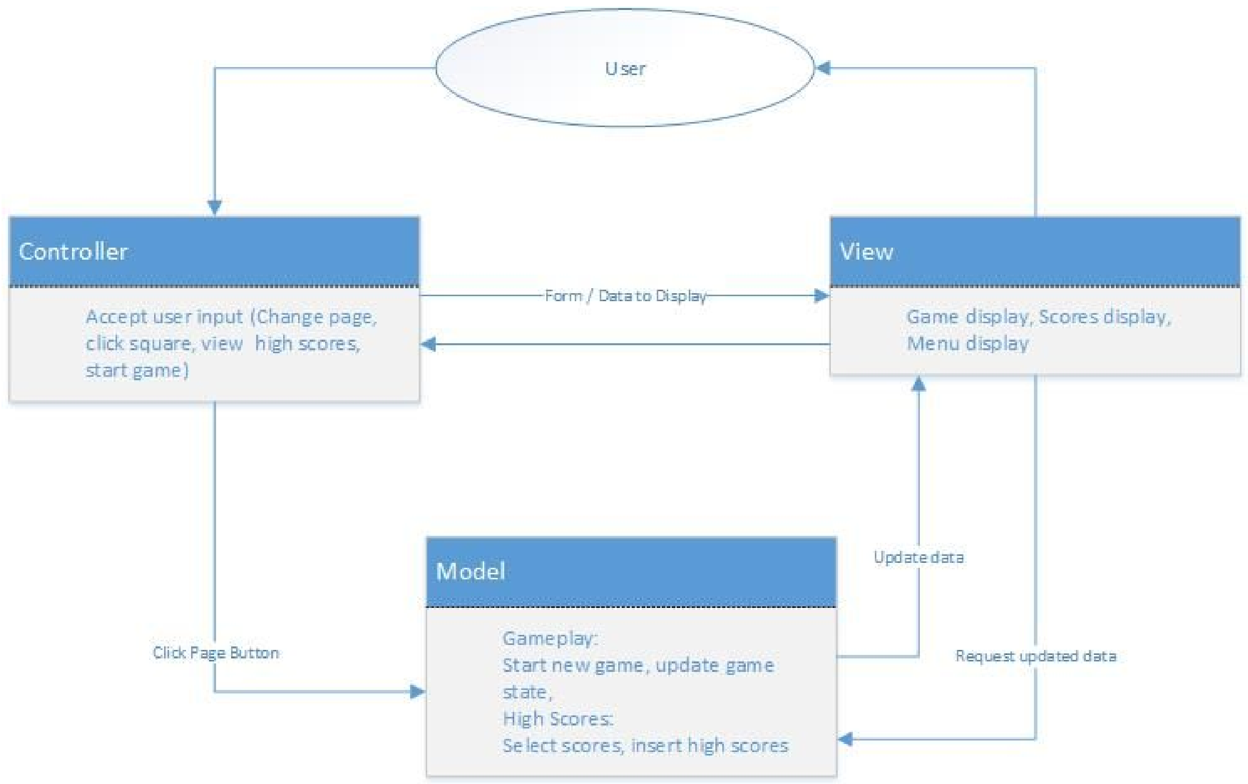
Buttons: Standard, Hexagon, Colours, High Scores, Start Game, Search Scores, Grid Square
Text Fields: Grid Size, Mines, Game Mode (High Scores Page)

Outputs:

Data: Button click data submitted to Model

A simple diagram of the interactions between each section can be viewed below.

Video link


User
Controller
Accept user input (Change page, click square, view high scores, start game)
Form / Data to Display
View
Game display, Scores display, Menu display
Update data
Request updated data
Click Page Button
Model
Gameplay:
Start new game, update game state,
High Scores:
Select scores, insert high scores

3.1 Interface Design Implementation

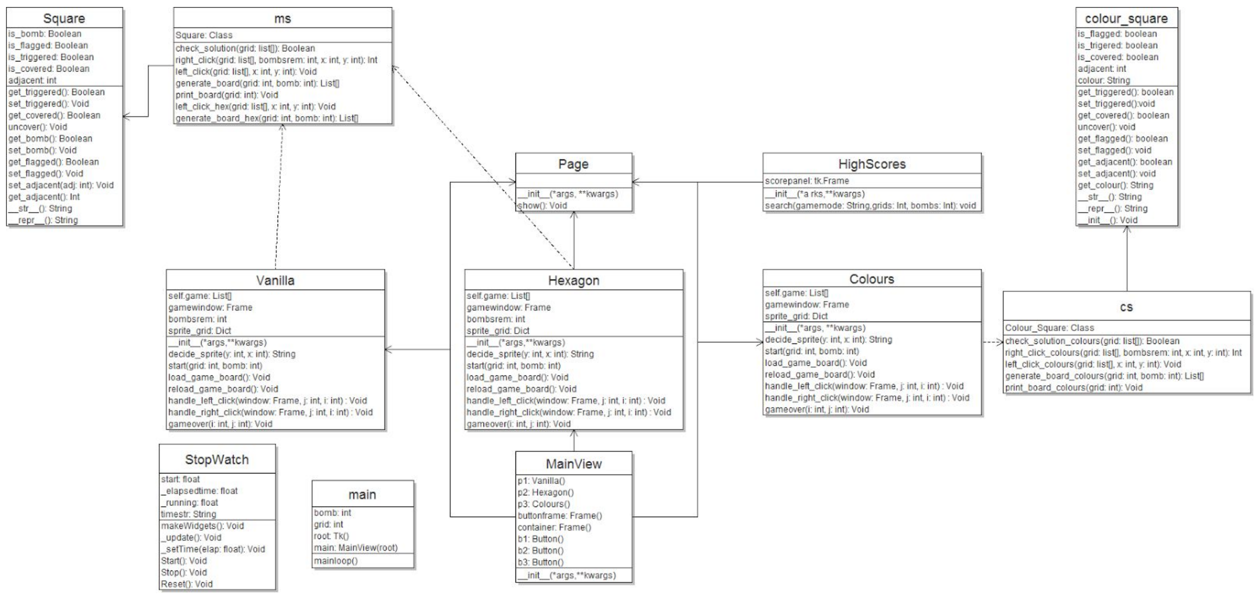
The interface utilised for this application has been implemented using an event driven design methodology. This implementation was chosen over callback driven design because it has extremely low coupling and allows lower levelled subjects to interact with higher level observers without impacting the layering of the system.

[Video link]

4.0 Design Documentation

4.1 Class Diagram

The following class diagram demonstrates all of the components of the software project, and their interactions.

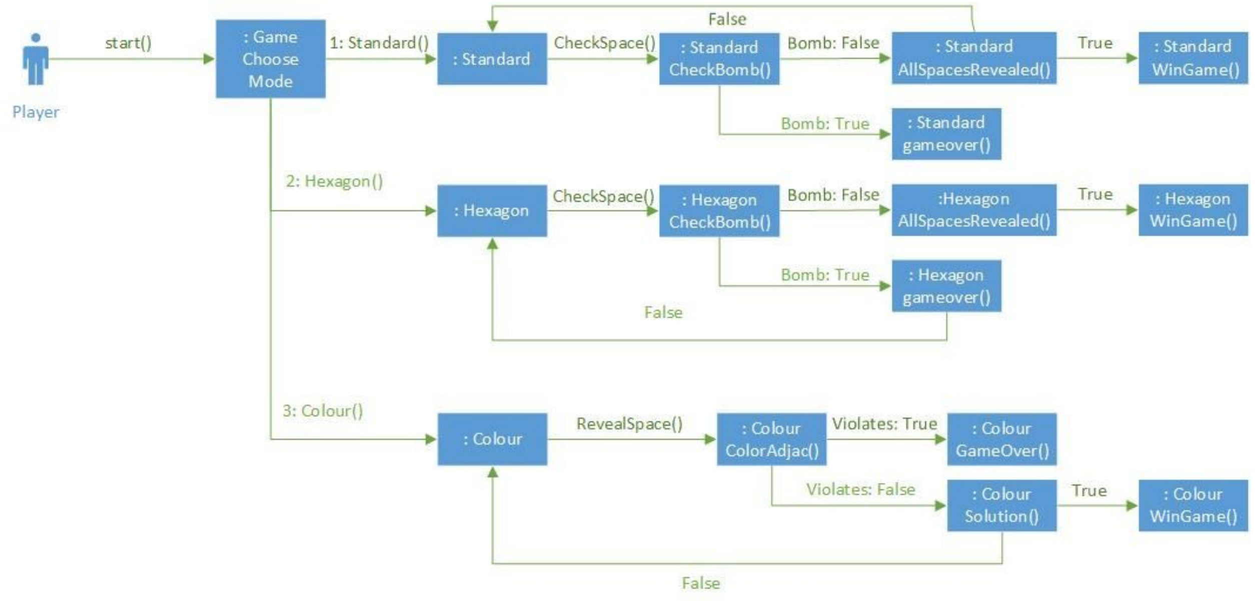

Square
is_bomb: Boolean
is_flagged: Boolean
is_triggered: Boolean
is_covered: Boolean
adjacent: int
get_triggered(): Boolean
set_triggered(): Void
get_covered(): Boolean
uncover(): Void
get_bomb(): Boolean
set_bomb(): Void
get_flagged(): Boolean
set_flagged(): Void
set_adjacent(adj: int): Void
get_adjacent(): Int
__str__(): String
__repr__(): String
ms
Square: Class
check_solution(grid: list[]): Boolean
right_click(grid: list[], bombsrem: int, x: int, y: int): Int
left_click(grid: list[], x: int, y: int): Void
generate_board(grid: int, bomb: int): List[]
print_board(grid: int): Void
left_click_hex(grid: list[], x: int, y: int): Void
generate_board_hex(grid: int, bomb: int): List[]
Page
__init__(*args, **kwargs)
show(): Void
HighScores
scorepanel: tk.Frame
__init__(*a rks,**kwargs)
search(gamemode: String,grids: Int, bombs: Int): void
colour_square
is_flagged: boolean
is_trigered: boolean
is_covered: boolean
adjacent: int
colour: String
get_triggered(): boolean
set_triggered():void
get_covered(): boolean
uncover(): void
get_flagged(): boolean
set_flagged(): void
get_adjacent(): boolean
set_adjacent(): void
get_colour(): String
__str__(): String
__repr__(): String
__init__(): Void
Vanilla
self.game: List[]
gamewindow: Frame
bombsrem: int
sprite_grid: Dict
__init__(*args,**kwargs)
decide_sprite(y: int, x: int): String
start(grid: int, bomb: int)
load_game_board(): Void
reload_game_board(): Void
handle_left_click(window: Frame, j: int, i: int) : Void
handle_right_click(window: Frame, j: int, i: int) : Void
gameover(i: int, j: int): Void
Hexagon
self.game: List[]
gamewindow: Frame
bombsrem: int
sprite_grid: Dict
__init__(*args,**kwargs)
decide_sprite(y: int, x: int): String
start(grid: int, bomb: int)
load_game_board(): Void
reload_game_board(): Void
handle_left_click(window: Frame, j: int, i: int) : Void
handle_right_click(window: Frame, j: int, i: int) : Void
gameover(i: int, j: int): Void
Colours
self.game: List[]
gamewindow: Frame
sprite_grid: Dict
__init__(*args, **kwargs)
decide_sprite(y: int, x: int): String
start(grid: int, bomb: int)
load_game_board(): Void
reload_game_board(): Void
handle_left_click(window: Frame, j: int, i: int) : Void
handle_right_click(window: Frame, j: int, i: int) : Void
gameover(i: int, j: int): Void
cs
Colour_Square: Class
check_solution_colours(grid: list[]): Boolean
right_click_colours(grid: list[], bombsrem: int, x: int, y: int): Int
left_click_colours(grid: list[], x: int, y: int): Void
generate_board_colours(grid: int, bomb: int): List[]
print_board_colours(grid: int): Void
StopWatch
start: float
_elapsedtime: float
_running: float
timestr: String
makeWidgets(): Void
_update(): Void
_setTime(elap: float): Void
Start(): Void
Stop(): Void
Reset(): Void
main
bomb: int
grid: int
root: Tk()
main: MainView(root)
mainloop()
MainView
p1: Vanilla()
p2: Hexagon()
p3: Colours()
buttonframe: Frame()
container: Frame()
b1: Button()
b2: Button()
b3: Button()
__init__(*args,**kwargs)

Video link

4.2 Collaboration Diagrams

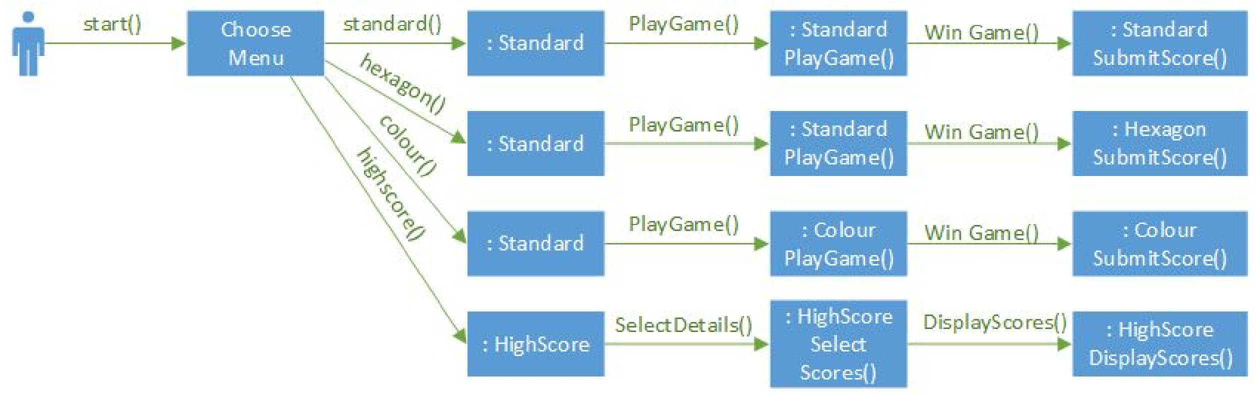
4.2.1 Collaboration Diagram 1: Basic Gameplay

The following collaboration diagram demonstrates the interaction between objects during standard, hexagon and colour game modes.



4.2.2 Collaboration Diagram 2: Insert Score / View Scores

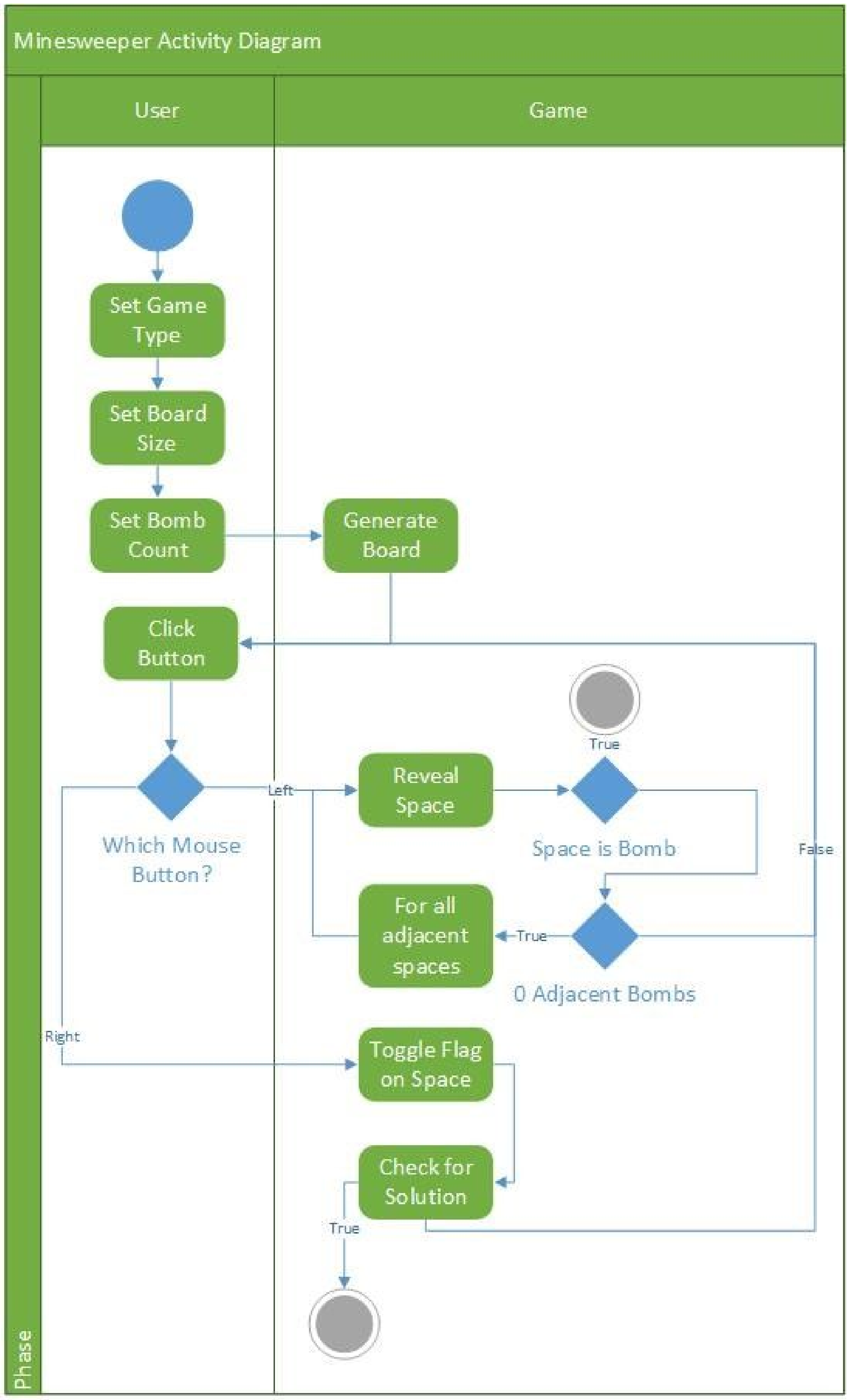
The following collaboration diagram demonstrates the interaction between objects when submitting and viewing scores.



Video link

4.3 Sequence Diagram

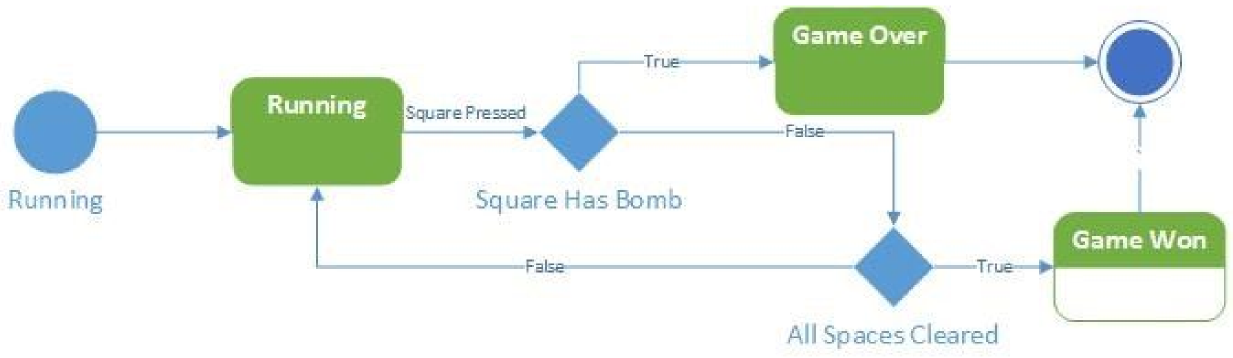
This sequence diagram demonstrates the flow of tasks between the game and the user during a standard game of minesweeper.



Video link

4.4 State Chart

The following state chart represents the shift between the running, game won and gameover states during the gameplay of a standard or hexagonal game of minesweeper.



5.0 Application Testing and Version Control

5.1 Version Control

Version control for this project has been managed using Git and GitHub. This platform of version control was selected as it is convenient to store program files in a centralised location, and it provides a convenient user-interface for checking commit update history and file changes. The repository for this project can be found here.

5.2 Testing Methodology

Testing for this project has been performed using the python unittest module, or by manually testing the application features. Features which can be tested quickly and easily using unittest, such as the square and colour_square getter and setter functions are tested using the unittest module, however for features which are more complicated to perform structured testing on it is performed manually by compiling and running the program, then testing the feature.

Video link

5.2.1 Example Source Code

5.2.1.1 'square' class data from ms.py

```python
class square:
   def __init__(self):
       self.is_bomb=False
       self.is_flagged=False
       self.is_triggered=False
       self.is_covered=True
       self.adjacent=0

   def get_triggered(self):
       return self.is_triggered

   def set_triggered(self):
       self.is_triggered = True

   def get_covered(self):
       return self.is_covered

   def uncover(self):
       self.is_covered=False

   def get_bomb(self):
       return self.is_bomb

   def set_bomb(self):
       self.is_bomb = True

   def get_flagged(self):
       return self.is_flagged

   def set_flagged(self):
       self.is_flagged = not self.is_flagged

   def set_adjacent(self,adj):
       self.adjacent=adj
       pass

   def get_adjacent(self):
       return self.adjacent

   def __str__(self):

       if self.get_flagged():
           return "F"

       if self.get_covered():
```

Video link

```
                return " "

            if self.get_bomb():
                return "B"
            return str(self.get_adjacent())

    def __repr__(self):

            if self.get_flagged():
                return "F"

            if self.get_covered():
                return " "

            if self.get_bomb():
                return "B"
            return str(self.get_adjacent())
```

5.2.1.2 ‘TestSquare’ class data from test.py

```
class TestSquare(unittest.TestCase):
    def test_square_set_get_triggered(self):
        s = ms.square()
        self.assertEqual(s.get_triggered(), False)
        s.set_triggered()
        self.assertEqual(s.get_triggered(),True)

    def test_square_set_get_covered(self):
        s = ms.square()
        self.assertEqual(s.get_covered(), True)
        s.uncover()
        self.assertEqual(s.get_covered(), False)

    def test_square_get_set_bomb(self):
        s = ms.square()
        self.assertEqual(s.get_bomb(),False)
        s.set_bomb()
        self.assertEqual(s.get_bomb(),True)

    def test_square_get_set_flagged(self):
        s = ms.square()
        self.assertEqual(s.get_flagged(),False)
        s.set_flagged()
        self.assertEqual(s.get_flagged(),True)

    def test_square_get_set_adjacent(self):
        s = ms.square()
        self.assertEqual(s.get_adjacent(),0)
        for i in range(0,10):
            s.set_adjacent(i)
            self.assertEqual(s.get_adjacent(),i)
```

Video link

```
def test_square_repr_(self):
    s=ms.square()
    print(s)

def test_square_str_(self):
    s=ms.square()
    print(str(s))

pass
```

6.0 Software Architecture

6.1 Requirements

1.1 Functional Requirements

1. The user must be able to left click a block to reveal it.
2. When a block is revealed, all of the blocks immediately surrounding it must display how many bombs are within 5 spaces from them.
3. The user must be able to right click a block to 'tag' it as a bomb
4. The game must be lost when a bomb is uncovered
5. The game must be won when all non-bomb spaces are uncovered
6. The game must be playable in standard, hexagon and colours mode

1.2 Non-Functional Requirements

1. The user must be able to select a difficulty
2. The user must be able to select a grid size
3. Scores must be decided by completion speed for a specific board size
4. Scores will be stored in a local sqlite3 database

1.3 Constraints

1. The program must be portable and upgradeable
2. The program must be modular
3. The program must be multi-platform

Video link

6.2 Software Architecture Justifications

6.2.1 Language Implementation

Most of the requirements specified are available in any programming language. Left block clicking, revealing space recursion, right click tag, game loss, game win and multi-gamemode can be implemented regardless of implemented language. Difficulty, grid size, and scores can also be implemented using most languages. However, the best portable and multi-platform languages are Java, Python, or a web-based language such as Javascript or PHP. The original version of this project was written using PHP on a web server, however the PHP server could not handle the level of recursion required for the revealing algorithm so the platform was changed to Python, which is a language which is portable, multi-platform and provides all of the features required to implement the requirements listed above.

6.2.1.1 Graphics Library

The python Tkinter graphics library was chosen over other libraries, as it comes standard with all 3.6.x python compilers and can be used across platforms.

Video link