Software Design Patterns

Jordan Stephano Gray 40087220

November 2015

1 Introduction

Throughout the history of computer programming there have been many design patterns that have been realised. These design patterns are very useful for efficient programming. This project will go through some of these patterns and their implementation in a 2D game-like simulation.

The game designed here is called "Space Meat" and features the player character which is an alien-type creature who's goal is to invade a ship and consume anything it can whilst avoiding any dangerous enemies when necessary.

Figure 1: A 5x5 Game Panel

2 Game Mechanics

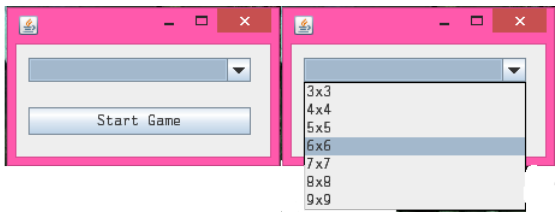

Figure 2: Initial GUI

The game follows a 2D grid structure. When the user runs the application a small launcher GUI will allow the player to choose from a 3x3 grid up to a 9x9 grid as seen in Figure 2. The game will then start up with the selected grid. The "Player" character will load into any tile randomly except from the top left tile. The player controls the simulation through a few buttons on a separate GUI. The main button is the "Move" button. This controls the simulation and when it runs. Every time this button is pressed a few things happen, the player will move to a neighbouring tile randomly (in 8 directions), an enemy will have a 1/3 chance to spawn in the top left tile, a consumable will have a 1/3 chance of spawning in the top left tile and each enemy/consumable entity will move to a random neighbouring tile. The other three buttons have similar functions to each other, they each have a label "Space Rodents", "Space Dogs" and "Enemies" all labelled under diet. These will change the players diet state for each one.

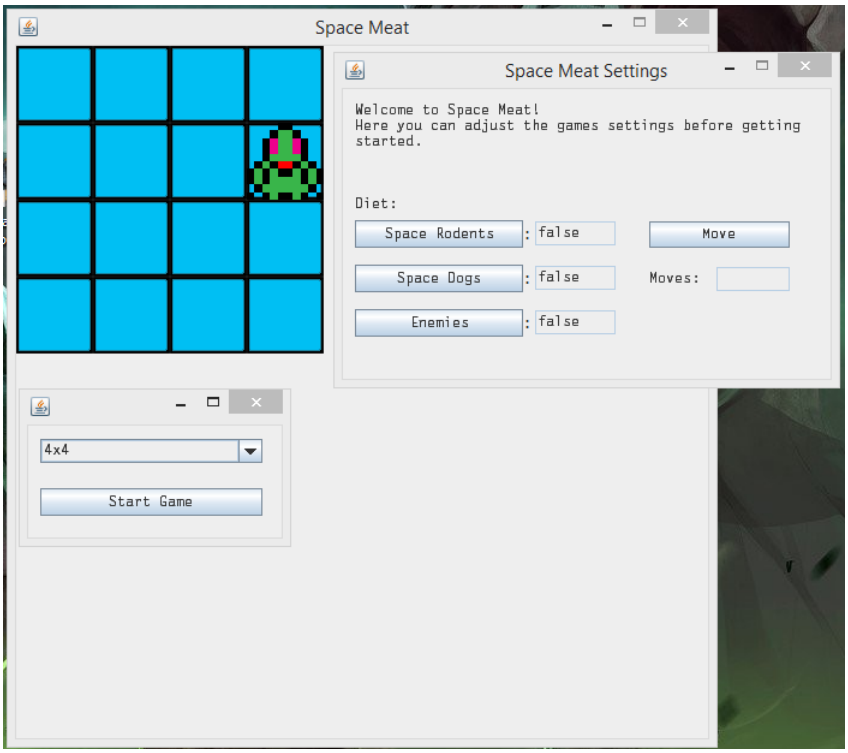

Figure 3: A 4x4 Game Panel with the GUIs

When the player enters the same tile as a consumable two things can happen, if the players diet is set to that creature the player will consume it and it will be removed from the grid as the player takes its place. If the players diet is not set to the consumable then they will occupy the same tile and nothing else will happen.

When the player enters the same tile as an enemy two things can happen. If the players diet is set to the enemy then the player will consume the enemy(s) and occupy that space. If the players diet is not set to the enemy and enters the same space as the enemy then the player will be consumed and this will result in a game over.

Every time the user moves successfully a move counter will count up the total number of times the player has moved and when the player gets consumed and receives a game over a "GAME OVER" message and the number of moves made appears in the console.

3 Recursion

For the random movement and random spawning mechanics recursion was implemented. This is the process of calling a specific method inside of itself, if not used correctly this can be dangerous but in this case it was necessary. Random numbers were being generated to produce movement for each of the 8 ways but if the entity was at an edge and tried to move out of bounds but could not the the move method would call itself again until movement was successful.

```
// Move method generates a random direction to move in.
   public void move() {

      // Initialise new random variable
      rand = new Random();
      // 8 is the maximum and 1 is the minimum returned.
      randMove = rand.nextInt(8) + 1;

      // If random number is 1 move upwards.
      if (randMove == 1 && y != 0) {

         y -= yMove;
      }
      // If random number is 2 then move right
      else if (randMove == 2 && x != tile.TILE_WIDTH * (gridSize - 1)) {

         x += xMove;
      }
      // if random number is 3 then move down
      else if (randMove == 3 && y != tile.TILE_WIDTH * (gridSize - 1)) {

         y += yMove;
      }
      // if random number is 4 then move left
      else if (randMove == 4 && x != 0) {

         x -= xMove;
      }
      // if random number is 5 move up/left diagonally
      else if (randMove == 5 && x != 0) {
         if (y != 0) {
            x -= xMove;
            y -= yMove;
         }
      }
      // if random number is 6 move up/right diagonally
      else if (randMove == 6 && x != tile.TILE_WIDTH * (gridSize - 1)) {
         if (y != 0) {
            x += xMove;
```

```
            y -= yMove;
        }
    }
    // if random number is 7 move down/right diagonally
    else if (randMove == 7 && x != tile.TILE_WIDTH * (gridSize - 1)) {
        if (y != tile.TILE_HEIGHT * (gridSize - 1)) {
            x += xMove;
            y += yMove;
        }
    }
    // if random number is 8 move down/left diagonally
    else if (randMove == 8 && x != 0) {
        if (y != tile.TILE_HEIGHT * (gridSize - 1)) {
            x -= xMove;
            y += yMove;
        }
    }
    // Use recursion to call again if it cant move
    else {
        move();
    }

}
```

4 Design Patterns

4.1 Factory Pattern

The factory pattern involves the use of inheritance. Firstly you create a basic object, in this project the main "Creature" class was used for this. Then more classes are created forming the basic creatures that share the same attributes and methods. Here "DeathClaw", "HumanEnemy" and "SpaceBee" classes were used and the point of using the factory pattern is to create one of these objects using a method which takes in an argument which represents each individual object and set it to a "Creature" object which will create an instance of this specific creature object. For example calling the "createEnemy" method which takes in a string equivalent to each ones name and create an instance of that enemy/creature object.

The below code shows where in the project this is implemented. If not for the factory pattern it would have been more difficult to randomly select an enemy creature to spawn using this method and simplified the coding process.

```
// Factory create enemy method.
    public static Creature createEnemy(Game game, String enemyName) {

        // Declare empty creature object
```

```
    Creature creature = null;

    // If human has been entered then create a human
    if (enemyName.equalsIgnoreCase("human")) {

       creature = new HumanEnemy(game,
           Creature.DEFAULT_CREATURE_WIDTH,
           Creature.DEFAULT_CREATURE_HEIGHT,
            game.gridSize);
    }
    // If deathclaw has been entered create a deathclaw.
    if (enemyName.equalsIgnoreCase("deathclaw")) {

       creature = new DeathClaw(game, Creature.DEFAULT_CREATURE_WIDTH,
           Creature.DEFAULT_CREATURE_HEIGHT,
            game.gridSize);
    }
    // If space bee has been entered create a space bee.
    if (enemyName.equalsIgnoreCase("spacebee")) {

       creature = new SpaceBee(game, Creature.DEFAULT_CREATURE_WIDTH,
           Creature.DEFAULT_CREATURE_HEIGHT,
            game.gridSize);
    }
    // By default creates a human
    if (creature == null) {

       creature = new HumanEnemy(game, 64, 64, game.gridSize);
    }

    return creature;
}
```

4.2 Observer Pattern

The Observer Pattern demonstrates a relationship between two kinds of objects, observer objects and observable objects. An observer object observes observable objects, this means an observer and observable objects can communicate with each other without being directly linked. For example if an observable object changes what state it is currently in it will notify the observer object and the observer object can act on this. In this project the observable object is the "Game" object (Game class) and there are two observers, each of the GUIs. They both observe the state the game is in whether it is running or not and methods are run based on this.

This is carried out using the "GameListener" Interface, "GameEvent" Class, both of the GUI classes and the game class. Two observers to one observable object.

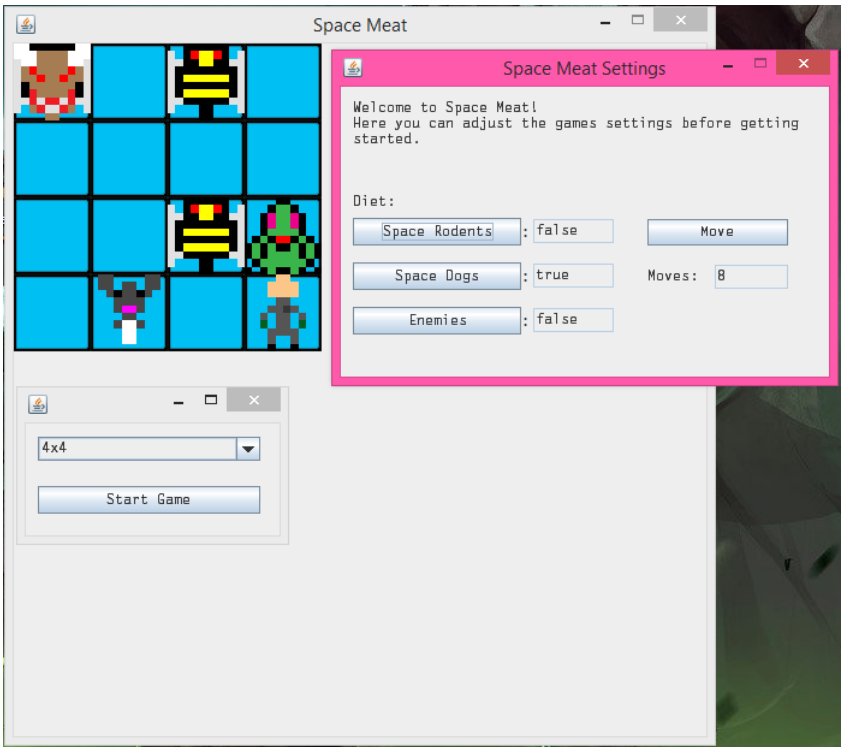

Figure 4: Game after moves on standard grid

5 Threads

Threads are used in this project to run the main bulk of the game. This allows for running the game separately to the rest of the application and continuously at a set frame rate, in this case 60FPS. There is no need in this case to put threads to sleep. This gave control over starting and stopping the game loop whenever needed.

```
// Create new thread object.
// Everything in this class runs separately to rest of app.
private Thread thread;
// Create running boolean for the game loop
private boolean running = false;


// Synchronised to work with threads
public synchronized void start() {

   // The the game is already running then return out of method.
```

```
    if (running)
      return;

    // When started set running to true
    running = true;

    thread = new Thread(this); // Set new thread object to run game
                       // class(this).
    // This calls the run method.
    thread.start();
  }

  // Stop thread safely
  public synchronized void stop() {

    // If game has stopped then return.
    if (!running)
      return;

    running = false;

    try {
      thread.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
```

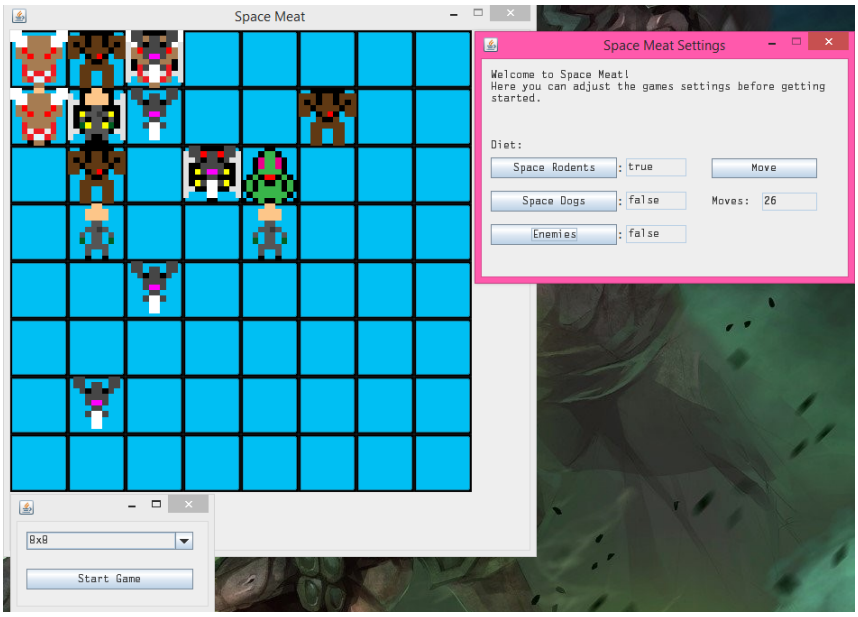

Figure 5: Game on larger variable grid

6 Variable Grid Size

As seen in Figure 2 and demonstrated between Figures 3 and 5 the user can choose from a variety of grid size. The code in this project could technically allow from a 2x2 grid all the way up to any possible number of NxN grid sizes. A range from 3x3 to 9x9 seemed the most optimal way to go for practicality of the game so the user can choose from these in a GUI.

This as easily implemented as each "Tile" was its own object and with the use of a "World" class which could contain a variety of tiles all that was needed was the use of basic OO programming and the communication between these objects.

The movement of each entity was based around whatever size the chosen grid was aswell as player spawning.

```
import java.awt.Graphics;
import java.awt.image.BufferedImage;

//represents in game tiles
public class Tile {

   //STATICS HERE
```

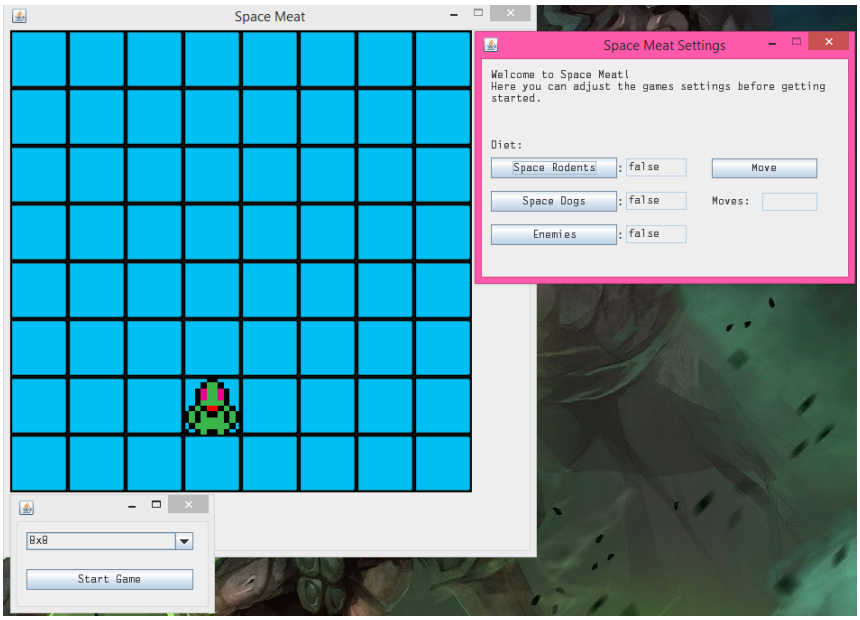

Figure 6: Game on 9x9

```
//Holds 1 instance of every tile
public static Tile[] tiles = new Tile[256];
public static Tile greyTile = new GreyTile(0); //Give grey tile an id
    of 0

//CLASS

//Tile attributes.
//Width and height.
public static final int TILE_WIDTH = 64, TILE_HEIGHT = 64;

protected BufferedImage texture; //Each tile has an image
protected final int id;       //Each tile has an id

//---------------------------------
//---------------------------------

public Tile(BufferedImage texture, int id) {

   this.texture = texture;
   this.id = id;

   //Index is id for tile.
   tiles[id] = this;
```

```
    }

    //--------------------------------
    //--------------------------------

    //If is solid then you cant walk on it.
    public boolean isSolid() {

        return false;
    }

    //Update variables
    public void tick() {


    }

    //Update graphics. Takes in x and y.
    public void render(Graphics g, int x, int y) {

        g.drawImage(texture, x, y, TILE_WIDTH, TILE_HEIGHT, null);
    }

    //--------------------------------
    //--------------------------------

    //Getter
    public int getId() {

        return id;
    }
}
```

```
import java.awt.Graphics;

public class World {

    private int gridSize;

    private int width, height;

    private int[][] tiles;

    //Default constructor
    public World(String path, int size) {

        loadWorld(path, size);
```

```
}

//Update variables.
public void tick() {


}

//Update graphics
public void render(Graphics g) {

   for (int y = 0; y < height; y++) {
      for(int x = 0; x < width; x++) {

         //Call get tile method to render tiles
         //Multiply by Tiles pixels to get actual world size.
         getTile(x, y).render(g, x * Tile.TILE_WIDTH, y *
             Tile.TILE_HEIGHT);
      }
   }
}

//Finds ID in tile array
public Tile getTile(int x, int y) {

   Tile t = Tile.tiles[tiles[x][y]];

   //Only set so many tiles so this does a check on id.
   if(t == null)
      return Tile.greyTile;

   return t;
}

//
private void loadWorld(String path, int gridSize) {

   width = gridSize; //Set width of level
   height = gridSize; //Set height of level
   tiles = new int[width][height]; //Set to new 2D array

   //For every tile position in this size world
   for(int x = 0; x < width; x++) {
      for(int y = 0; y < height; y ++) {

         tiles[x][y] = 0; //Sets every tile to id 1 tile.
      }
   }
}
```

```
   public int getWidth() {
      return width;
   }

   public void setWidth(int width) {
      this.width = width;
   }

   public int getHeight() {
      return height;
   }

   public void setHeight(int height) {
      this.height = height;
   }

}
```

7 Conclusion

Software design patterns are important to identify when receiving a design document as they make the designing of the code and its structure easier and more structurally sound. It also makes for reusable code and allows for the code to be expanded upon more easily when necessary i.e for client updates etc. Reusable, reliable and robust code are three very important attributes to have in a project and using these design patterns help with that

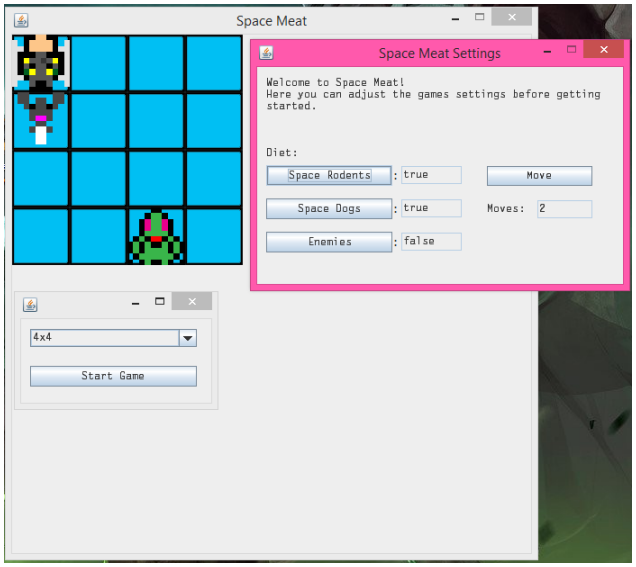

Figure 7: Game on 4x4 Early Stages