

Ant Farm - SOFT152

Aden Webb - 10574860

Jack Brewer - 10575202

Table of Contents

Solution to the Problem	3
PseudoCode	3
Passing information between ants	3
Passing information between ants - more detailed plan	3
Interacting with nest	4
Interacting with food	4
Aggressive ant pseudo code	5
Matching vectors pseudo code	5
Knows destroyed food	6
Class Relationships	7
Class Descriptions	8
AntFoodForm	8
AntAgent	9
Food	9
Nest	10
Aggressive Ant Agent	10
WorkerAntAgent	11
AggressiveAntNest	11
Data Structures	12
Evaluation	13

Solution to the Problem

PseudoCode

Shown here are simple representations of a few of the interactions and algorithms that will be written in order to create a working simulation.

Passing information between ants

This is a basic demonstration of how the ants will be able to pass information between themselves.

ant1 = ant being checked

ant2 = ant in range

Every tick

For every ant

 Check all other ants to see whether or not they are within a distance of 5

 If any ants are within a distance of 5

 If ant1 doesn't know where food is and ant2 does, share info

 If ant1 doesn't know where nest is and ant2 does, share info

Passing information between ants - more detailed plan

The following method is a slightly more in depth version of the first. It more closely represents what the code will look like when implemented into c#.

Foreach primaryAnt in antList

 Foreach secondaryAnt in antList

 If primaryAnt is within 5 of secondaryAnt

 If primaryAnt does not know where food is and secondaryAnt does

 primaryAnt learns where secondaryAnt thinks food is

 ENDIF

 If primaryAnt does not know where a nest is and secondaryAnt does

 primaryAnt learns where secondaryAnt thinks a nest is

 ENDIF

 ENDIF

ENDFOR

ENDFOR

Interacting with nest

This pseudocode shows how the ants will interact with the nest and when the ant will be allowed to deposit food to the nest.

Every tick

For every ant

 If the ant has come within a distance of 40 to the nest

 Approach

 When the ant reaches the nest - learn location (if not known)

 If ant is carrying food - deposit food and wander again

 Otherwise keep wandering

Interacting with food

The ants will need to interact with the food that the user has placed into the world - this process has been demonstrated in the following pseudocode. It includes when food will be picked up and when the food source will be depleted.

Every tick

For every ant

 If the ant has come within a distance of 40 with a food source

 Approach

 When the ant reaches the nest - learn location (if not known)

 If not carrying food, pick up food

 Reduce food total by 1

 If the new total is 0 then remove the food

 Wander

Aggressive ant pseudo code

The following algorithm shows how the aggressive ants interact with the worker ants and the distances at which events will occur.

```
Foreach aggressiveAnt in aggressiveAntsList
    Foreach workerAnt in workerAntList
        If distance between aggressiveAnt and workerAnt < 5
            aggressiveAnt is carrying food
            workerAnt is not carrying food
        Else if distance between aggressiveAnt and workerAnt < 40 and the aggressiveAnt is
            not carrying food and the workerAnt is carrying food
            aggressiveAnt start following workerAnt
        ENDIF
    ENDFOR

    If aggressiveAnt is carrying food
        aggressiveAnt return to nest

    else
        aggressiveAnt wander
    ENDIF
ENDFOR
```

Matching vectors pseudo code

This code will be used for comparing two vectors and seeing whether or not they are the same - this algorithm will help to reduce the size of some functions as well as minimizing repeated code.

MatchingVector(vectorOne, vectorTwo)

```
    If (vectorOne x value = vectorTwo x value) and (vectorOne y value = vector two y value)
        Return true
    Else
        Return false
    ENDIF
```

While working through implementing the previous algorithms a bug was encountered that caused ants to become stuck in a loop of discovering that a food source had been depleted but then being given that same food source as a new destination by another ant. In order to work around this issue the following algorithm was developed. It essentially allows ants to remember which food sources have run out and compare this to the new information they have just been given. If the new destination is a known empty food source - the information is disregarded.

Knows destroyed food

usedUpFoodList contains all food sources an ant knows has been used up
there is also a vector being passed in which is the vector an ant is trying to learn

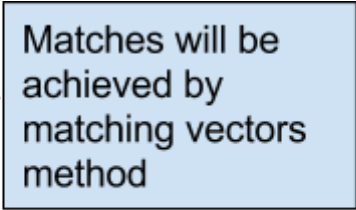
Foreach vector in an ants usedUpFoodList

 If the passed vector matches vector

 return true

 ENDIF

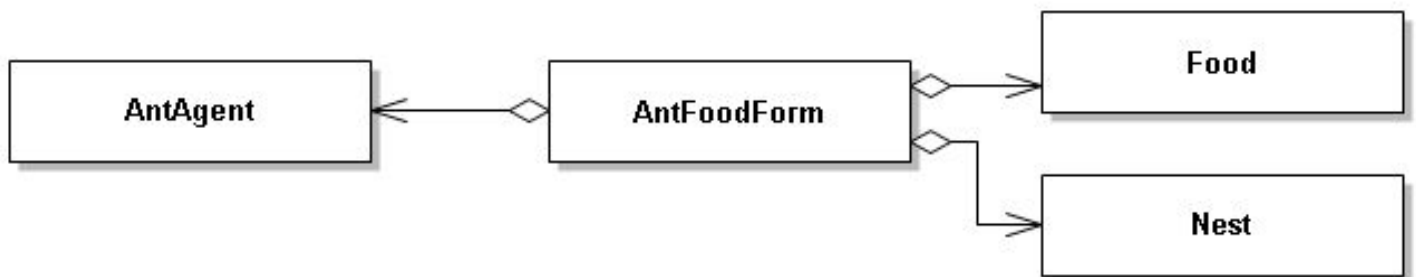
return false



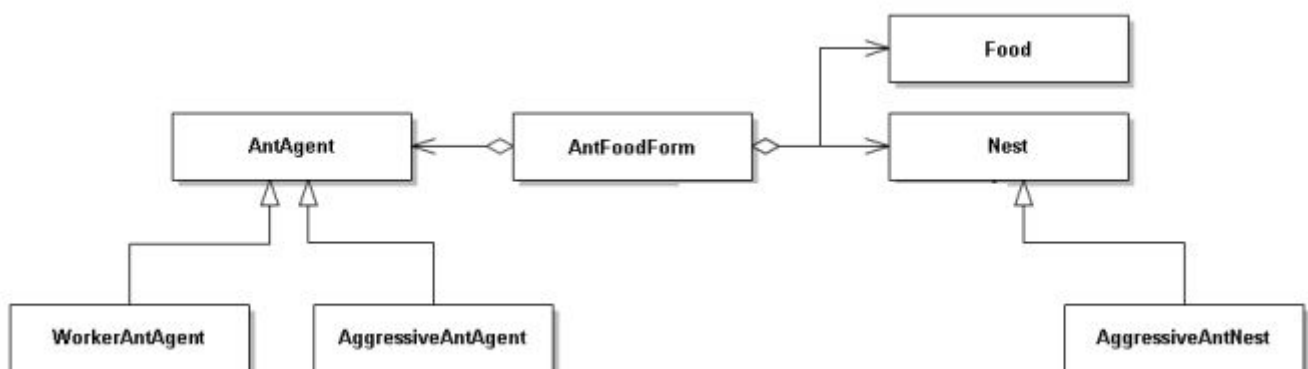
Matches will be
achieved by
matching vectors
method

Class Relationships

Shown below is the relationship between classes prior to adding the aggressive “robber” ants. All that is demonstrated here is aggregation. This is because the form contains all the other elements and they rely on the form to exist.



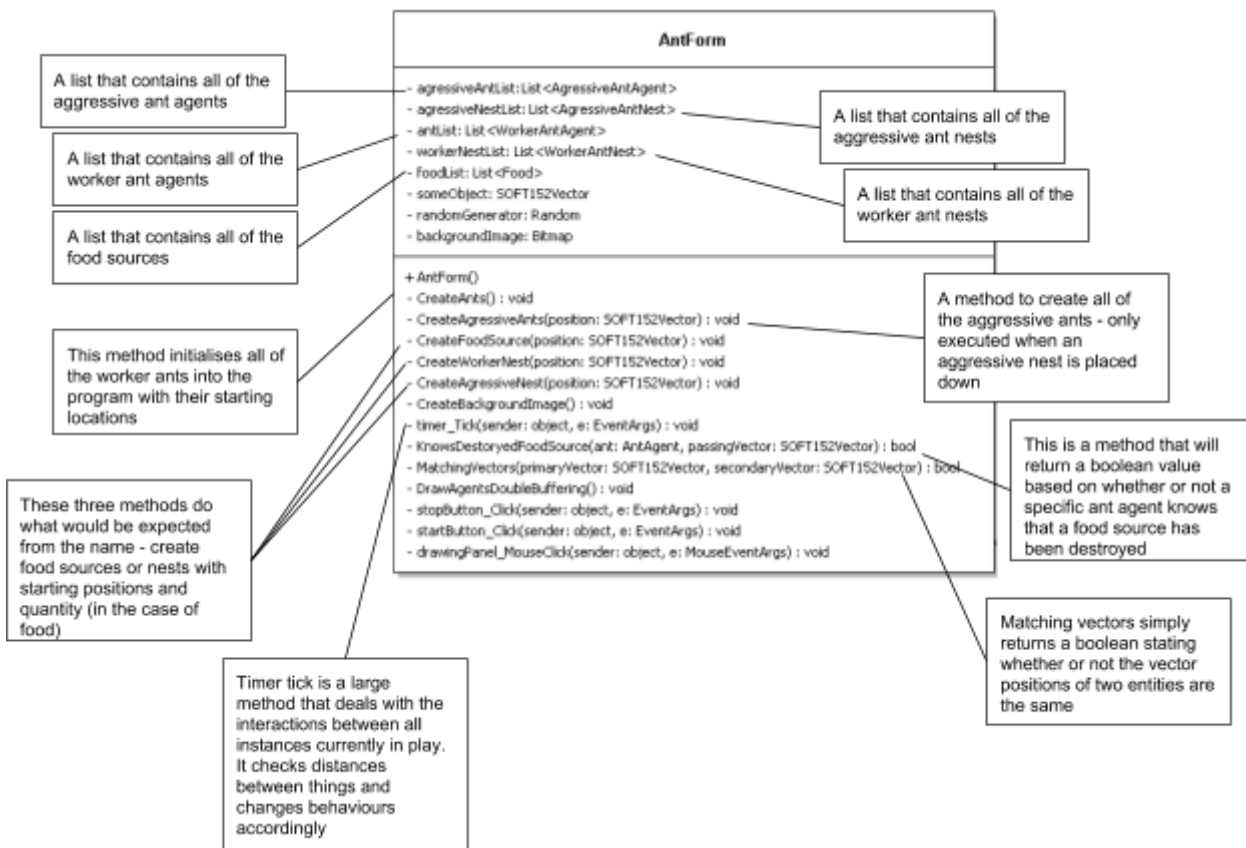
This diagram shows the class relationships after having added the aggressive ants and their respective nests. This means that not only is there aggregation being shown - but also inheritance. The use of inheritance allows shared methods and properties between the two types of ants but also unique behaviors that are individual.



Class Descriptions

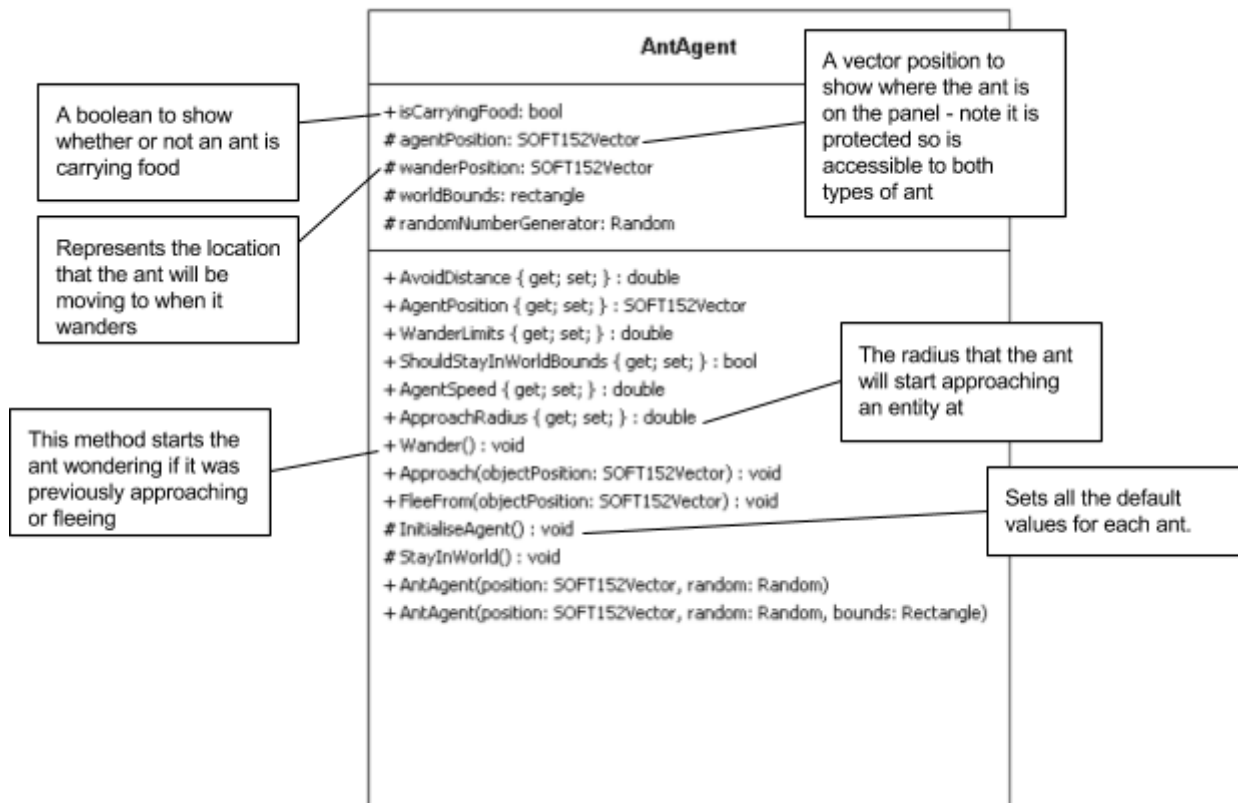
AntFoodForm

This class is used to contain all of the graphical elements as well as the core logic for the game. It will include methods for creating all the ants, nests and food sources on the users action. Another primary purpose for this class will be the actions that occur on each tick. The event handlers for the buttons on the form will also be found here.



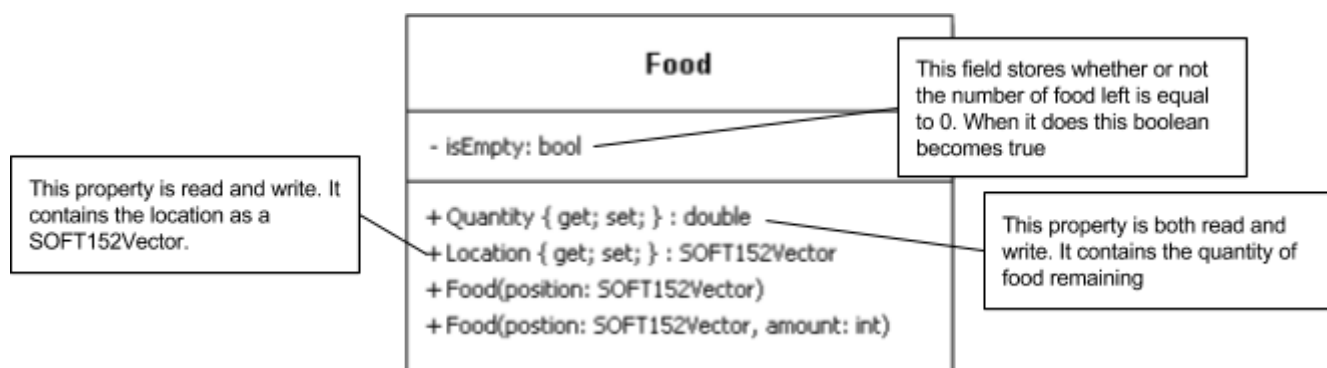
AntAgent

This class has the purpose of containing properties and methods that are shared between both types of ant. It is the base class to WorkerAntAgent and AgressiveAntAgent.



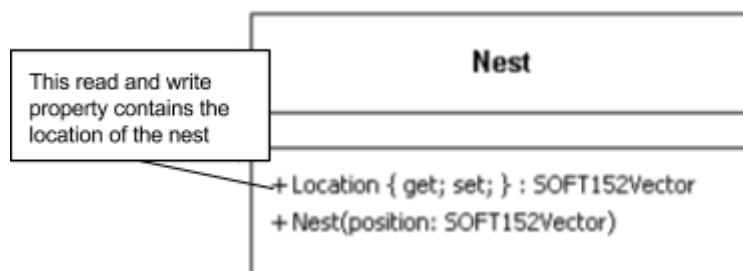
Food

The food class is a simple one thats purpose is just to contain the position and the amount of food left within it that can still be collected by worker ants.



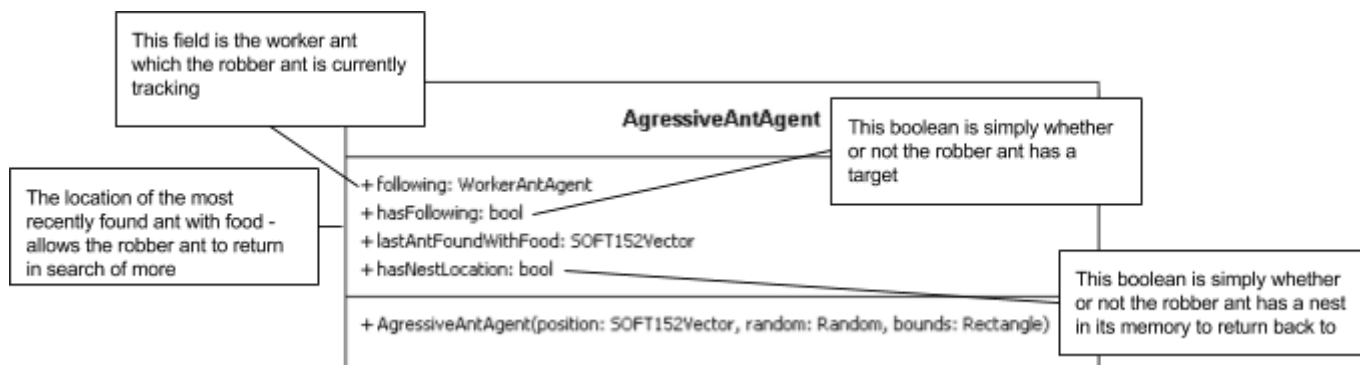
Nest

The nest class is responsible for the location of the nests that the user places. This is a very simple class as the nests are only ever really interacted with by other entities (hence the get accessor) and do not need to store any information of their own. It acts as the base class for the specific versions of the nest.



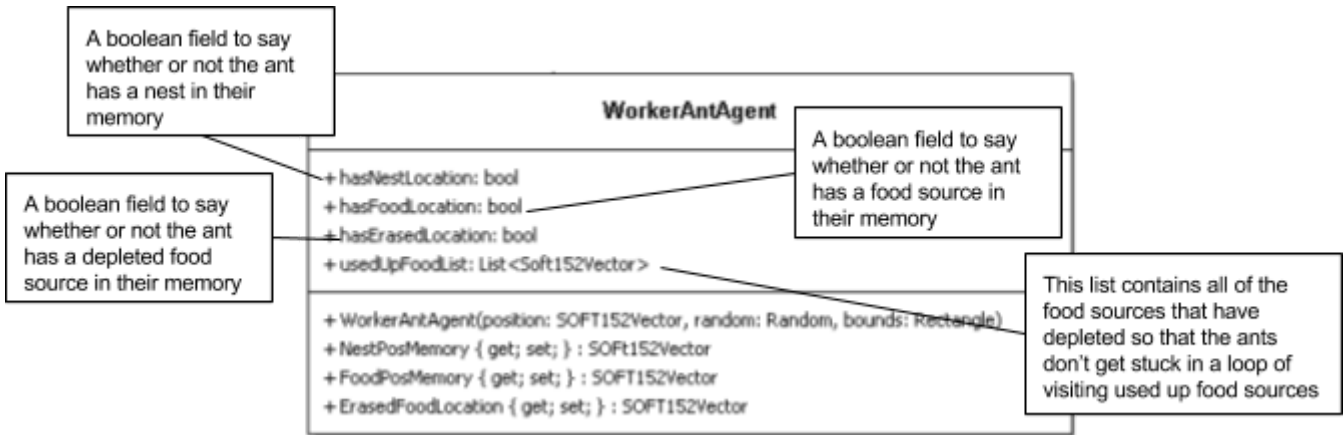
Aggressive Ant Agent

This class is a subclass of the `AntAgent` class. It is responsible for dealing with the data associated with the aggressive ant agents which will track worker ants and steal the food that they are currently carrying. They have fields that are unique to the `AggressiveAntAgent` class that are designed to deal with tracking worker ants.



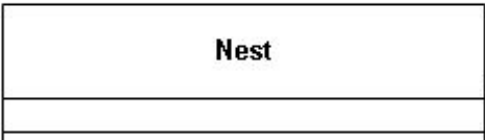
WorkerAntAgent

The worker ant class is the second subclass of AntAgent. This is the normal ant type. They will wander around and discover the location of food sources and nests as they come across them or are informed by other ants of important locations. They will then transport the food from the food source back to the nest and continue until the food source has been depleted. They are targetted by the aggressive ants and can have their food stolen.



AggressiveAntNest

This is the subclass of the AntNest class and deals with the nests created for the aggressive ants to return their collected food to. This is a separate class so that it is easier to reference from within the main body of the code.



Data Structures

List<WorkerAntAgents>

This list is used to store all of the worker ants that are created when the start button is pressed.

List<AgressiveAntAgents>

This list is added to when an aggressive ant nest is created - it then stores all aggressive ants associated with that nest.

List<WorkerAntNest>

This list is used to hold all of the nests that belong to worker ants. It demonstrates why a nest is useful in this situation as the user can place as many nests as they like. All of these nests are added into the list. If an array had been used here a hardcoded limit would have been needed - thus limiting the number of nests the user could place.

List<AgressiveAntNest>

Similar to the worker ant nest, this list holds all nests associated with the aggressive ants. A list is useful here again so that the user can place any number of aggressive ant nests without the need of a previously stated limit to how many can be stored by the program.

List<Food>

The food list shows that not only is it useful to be able to add entries to lists on the fly but also remove them. The user can click to place food at any time and should not be limited - but also once all the food has been taken away by the worker ants the food source is removed from the list. This shows how efficient the lists are as there is never any assigned data which is empty.

List<SOFT152Vector>

This list, unlike the others which can be found in the form class, is located within the worker ant class and contains all previously known food locations. The purpose of this list is so that ants do not get caught in a loop of trying to go to food, discovering that food has all run out and then immediately wandering into another ant who is also trying to reach the empty food. The original ant then tries to look for the food again and the loop perpetuates. Having this list allows the ants to disregard useless directions provided by other ants.

Evaluation

Overall this project went well and all the functions requested in the specification are present. The result is a fully working model that can simulate a colony of ants and how they interact with food and nests in their immediate vicinity.

Also included was the aggressive “robber” ants which steal food from the worker ants rather than learning where a food source is for themselves. This extra feature helps to add a layer of depth to the project as well as to demonstrate some more complex class relationships; namely inheritance. Adding this functionality required a restructure for the code - leading to a more clear and professional final structure.

One of the bugs noticed was where the ants sometimes appear to struggle with multiple food sources and instead focus on a particular source rather than going for one that may be nearer or more directly in their path. This leads to an odd “swerving” behaviour where the ants appear to deliberately avoid one food source to access another. This does not, however, affect the overall functionality of the program as the ants will always go back and use the previously avoided food source after the preferred one has been depleted.

Another function that would have been nice to have included is the ability to scale the program window to allow a larger or smaller area for the ants to roam within. This functionality was attempted but all the graphical elements appeared to stretch and the mouse clicks no longer created the desired resource at the right location. It was decided that more time should be put into elements that appeared on the specification and presentation type tasks should be left until all other elements were complete.

In terms of class structure the final solution works well - however this was an improvement from an initial idea. Previously the idea was to have the aggressive ants be a direct subclass from the worker ants or just “AntAgents”. This would have been less efficient than the final setup because the worker ants include methods that are entirely unused by the aggressive ants. The solution to this was to create the base AntAgent class and then have two subclasses - both aggressive ant agents and worker ant agents. This allowed all shared methods and properties to be accessed but also for each ant type to have its own unique methods.

In addition to adding the aggressive ants, a few quality of life improvements were also added. Examples of these are a matching vectors method and a knows destroyed food source method. These allow the code in more complex areas to look cleaner and more precise as well as allowing for more efficient and faster expansion if someone was to revisit the project in the future.

In conclusion the project worked well and all necessary features were added - as well as a few extras beyond the baseline standard.