**SOFT152**

**Ant Colony Simulation C# Project**

Reece Davies

## Contents

[Contents 2](#_Toc503628303)

[1. Introduction 3](#_Toc503628304)

[2. Background Information 3](#_Toc503628305)

[3. Method of Approach 3](#_Toc503628306)

[4. Classes Used 4](#_Toc503628307)

[5. Algorithm Design 5](#_Toc503628308)

[6. UML Class Diagram 6](#_Toc503628309)

[7. AntFoodForm 7](#_Toc503628310)

[8. AntFoodForm[Design] 9](#_Toc503628311)

[9. AntAgent 10](#_Toc503628312)

[10. RobberAntAgent 11](#_Toc503628313)

[11. Nest 12](#_Toc503628314)

[12. RobberNest 12](#_Toc503628315)

[13. Food 12](#_Toc503628316)

[14. Evaluation 13](#_Toc503628317)

## Introduction

This document will explain my approach to completing the program for the ‘Option 2 Ant Colony Simulation Project’ of the SOFT152 assignment. It will also contain detailed information of the classes that have been included in this program; as well as how the code has been structured. This will therefore allow the viewer to have a greater understanding of how it functions. The program has been written in C#, and created in Microsoft Visual Studio 2017.

## Background Information

The goal of this project, was to create a two-dimensional simulation of an ant colony collecting food. The user would be able to manually input ‘nest’ and ‘food’ objects into the program, in which the ants would perform certain tasks with each object. By placing food or nest objects, it would then create a graphical depiction of the object within the drawing panel. At the start, there will be no food nor nest. The ants would randomly wander around the drawing panel with the specific aim of coming across a food pile. If the user were to place some food into the drawing panel, the ants would eventually encounter it. However; in the beginning they do not know where the food is; but will learn once they are within a specific radius of the object. There are two ways of each ant learning where the food is: by either wandering close to it; or by coming into contact with another ant that knows the food location. The ants would then carry the food back to one of their nests. Once the ant reaches its nest, it will put down what it was carrying, and return to the food pile to collect some more. In the process of the ants collecting from the food pile, it will decrease in size until there is no food available from within that pile. They would again wander the drawing panel in search for some more food, and thus beginning the process again.

Each ant shares a very simple role; however, when grouped together in a large quantity, they can accomplish extraordinary and complex tasks in order to survive. This is known as “collective intelligence”, and occurs when many individuals collaborate with each other in an effort to achieve a common goal. This program has been developed to imitate this type of behaviour displayed by ants.

In order to achieve a ‘First’ in this assignment, a separate species of ants needs to be implemented into the program. These ‘Robber’ ants are responsible for stealing the food off the other ants, and taking it back to their own nest. They will then remember where this ant is, and will return to this position after delivering the food to their nest. This means that the chances of encountering another ant with food will be relatively high, as they will most likely encounter a chain of ants carrying food. The robber ants will not interact with the food piles, nor the original ant nests. Their primary function is to steal the food that is being carried by the original set of ants.

## Method of Approach

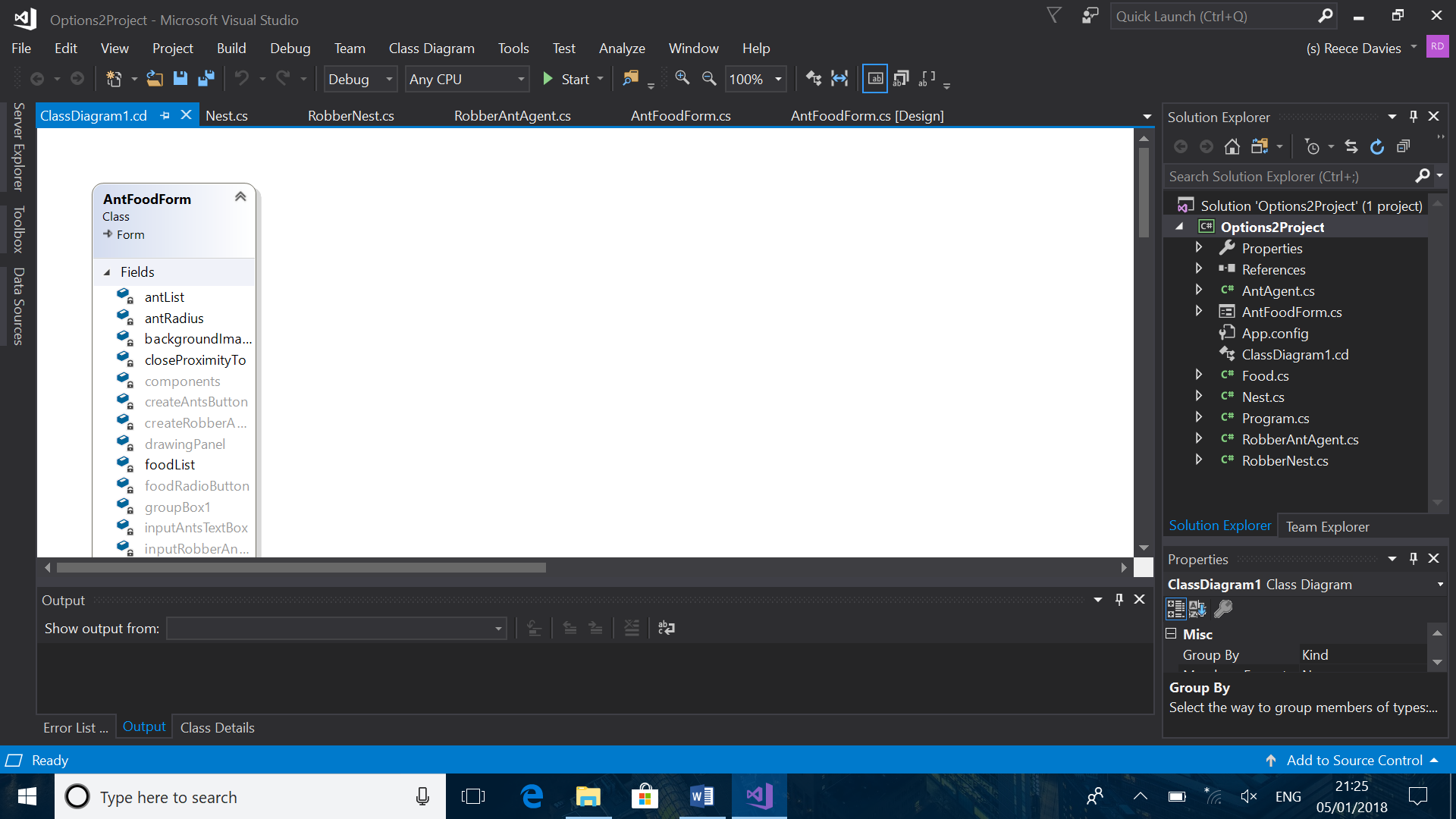
Initially, I had planned to create the ‘Option 1 Data Miner’ program. However, I felt that option 2 was far more interesting, as it contains aspects that may be considered as a simple artificial intelligence system. In the act of choosing option 2, this meant that a Visual Studio project was available on the Plymouth University DLE, containing vital code that allowed me to understand how to fully complete the program. Firstly, I tested the existing code in the program, following with a few alterations. These slight adjustments allowed me to gain an insight of what each method was responsible for, and how they can be affected by specific variables. As a result, I had a greater understanding on how it behaves, and what code can be implemented in order to improve the functionality.

Subsequently, I used pen and paper to plan the next few stages on how I might develop the program, using pseudo code. This specifically related to how items would be added to each list; new methods for certain instances; and the overall behaviour of each ant when encountering another ant, nest or food pile. Consequently; when I began programming, I knew exactly what I was aiming for, and hoped to achieve. Once I had coded the program to a specific point, so it was functioning adequately, I returned to pen and paper to focus on attention to detail. This made the program more visually pleasing to watch.

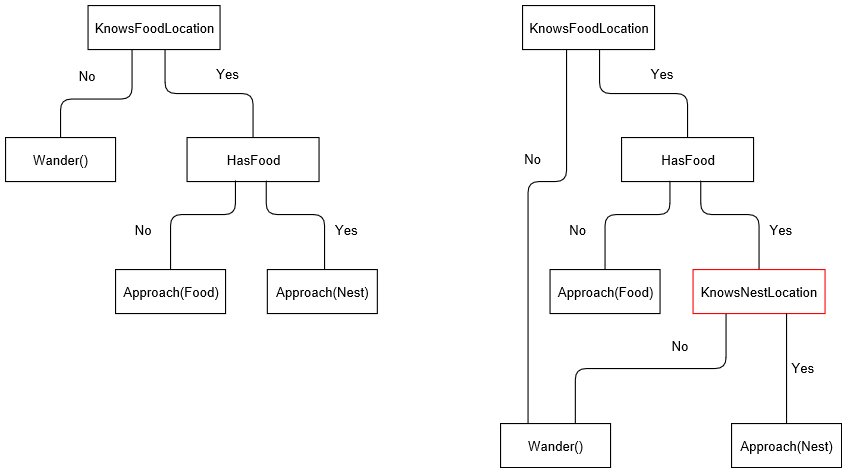
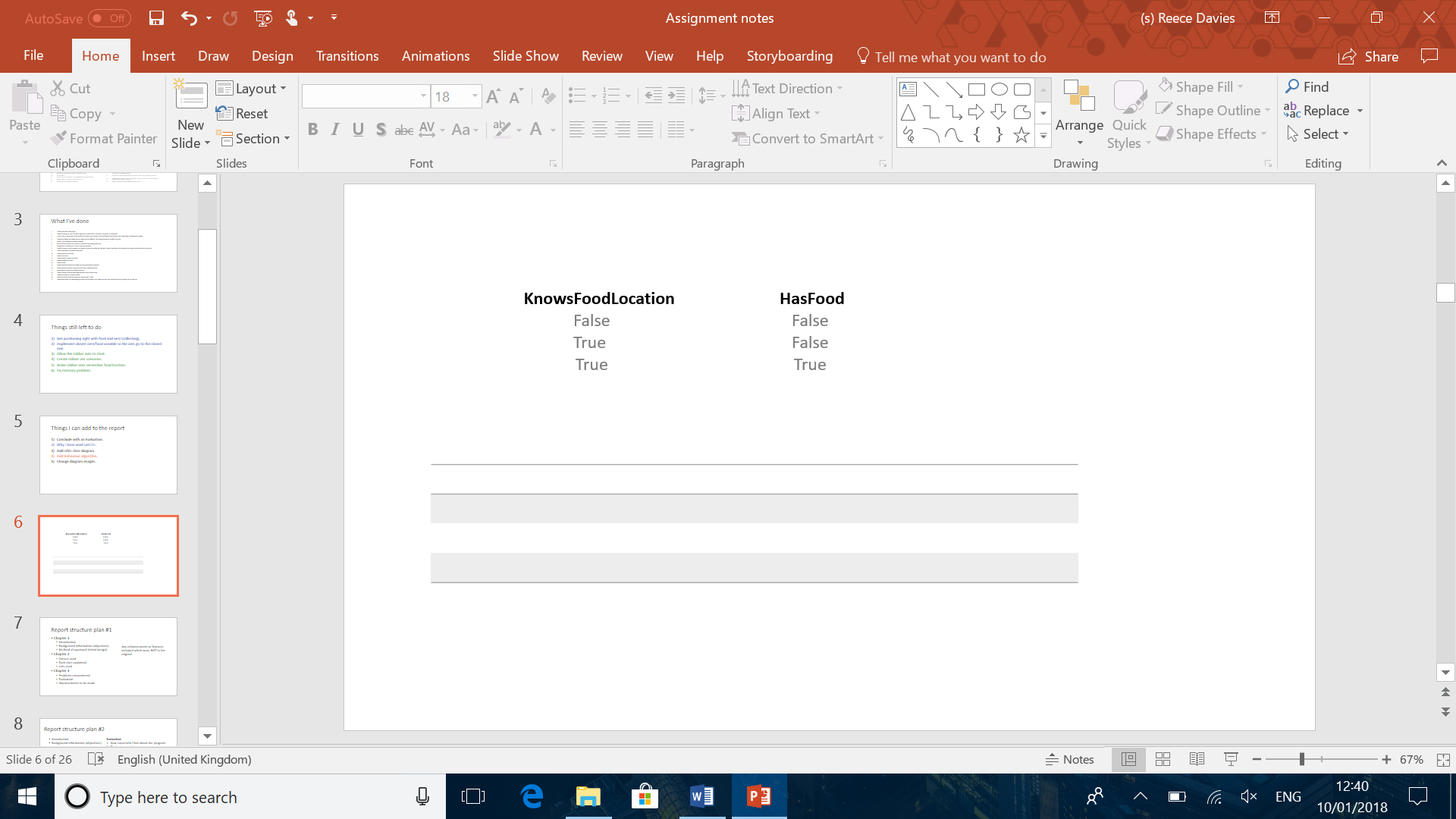
## Classes Used

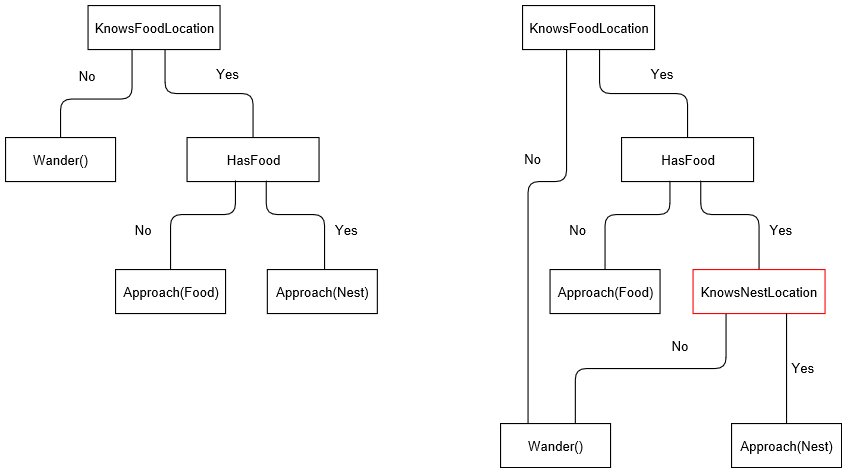
The classes that have been included in this program are:

* AntFoodForm
* AntAgent
* Food
* Nest
* RobberAntAgent
* RobberNest

Each class relates to a specific object that has been used within the program. This is because it has been structured in an Object-Oriented fashion, and therefore will be easier to develop. The program will be structured more professionally, and will leave less room for error. The purpose of these classes, is to store information about each object within the list of their class type. The information that is specifically stored in the ‘AntAgent’ class will dictate what action needs to be taken by that individual ant in the list. For instance; if the Boolean ‘KnowsFoodLocation’ was true, however the Boolean ‘KnowsNestLocation’ was false, the ant would know that it needs to find the nest, so that the food can be delivered to its colony.

## Algorithm Design

This is the original design of the algorithm I had planned to use for each ant. This flowchart shows which Booleans are true, and which are false. Depending on the situation, the ant would react differently and either wander, or approach a specific object. There are three ‘if’ statements that would check the Booleans ‘KnowsFoodLocation’, and ‘HasFood’. There are three different scenarios for this, thus the three ‘if’ statements:



However; after implementing the code, it was revealed that the ants would be required to learn the nest location as well. This therefore required a nested ‘if’ statement in the third scenario, where both ‘KnowsFoodLocation’ and ‘HasFood’ are true. If they know the nest location, they would approach it; and wander if they did not.

This new algorithm design shows this new ‘if’ statement implemented into the third scenario.

Previously, there was an idea to contain four scenarios; however, I soon came to realise that an ant could not be carrying food without knowing the location of the food pile.

## UML Class Diagram

## AntFoodForm

‘AntFoodForm’ is the main code of the form. This means that it controls any tools that have been added into the form, such as buttons and text boxes. As a result, it will allow for user input, therefore the user has more control of how the program functions. The tools and controls implemented into the program are shown in the design. ‘AntFoodForm’ consists of twenty-five Fields, and twenty-one Methods.

‘antList’ is a list solely for the ‘AntAgent’ class, and this allows there to be multiple ants in the program. Having it in a list also makes it easier to add a new ‘AntAgent’ to the program. To prevent complications, any class within a list has a dedicated loop. This loop will continuously repeat until it has reached object within that specific list. This is accomplished by setting the limit to “antList.Count”. It also means it will be easier to manage the behaviour of each ant, as the loop will run through each ‘AntAgent’ in the list, assigning them to a specific task for which Booleans are true and which are false.

‘antRadius’ is an integer, and this will dictate what distance an ant must be to another ant, when they communicate. Decreasing this value will mean the ants have to be closer together in order to exchange information.

‘backgroundImage’ is a bitmap, and has been set so it is the same size as the panel in the program. This is so the ants can be drawn onto the ‘drawingPanel’.

‘closeProximityTo’ is an integer, and functions similarly to ‘antRadius’, although this specifically relates to the distance an ant must be to a ‘Nest’ or ‘Food’, before they know its location. Increasing this value will mean that an ant will learn the locations from a farther distance from each ‘Nest’ or ‘Food’.

‘createAntsButton’ is the button in the form that will register the input from the ‘inputAntsTextBox’, converts it into an integer, and calls the ‘CreateAnts’ method with the argument being the user input integer.

‘createRobberAntsButton’ does the same function as ‘createAntsButton’, although it will register the input from inputRobberAntsTextBox’, following with calling the ‘CreateRobberAnts’ method. The argument is the user input integer.

‘drawingPanel’ is the panel that displays the graphics of each ‘AntAgent’, ‘Food’, ‘Nest’, ‘RobberAntAgent’ and ‘RobberNest’.

‘foodList’ functions the same as ‘antList’, although it is used for the ‘Food’ class, thus allowing the user to input food piles into the program.

‘foodPositionVector’ is a vector, that allows the robber ants to have a position to approach when they do not have any food and know the food location. This is because the ‘Approach’ method has an argument that is required to be a vector, as opposed to a point or integer.

‘foodRadioButton’, ‘nestRadioButton’ and ‘robberNestRadioButton’ are all radio buttons, in which the user can toggle between. When the user clicks on the drawing panel, it will create a new item, depending on which radio button has been checked. These radio buttons are held together in ‘groupBox1’.

‘nestList’ functions the same as ‘antList’, although it is used for the ‘Nest’ class, thus allowing the user to input nests into the program.

‘randomGenerator’ is a Random that allows the ants to spawn at random locations around the drawing panel. As opposed, to all spawning in one location.

‘robberAntList’ functions the same as ‘antList’, although it is used for the ‘robberAntAgent’ class, thus allowing the user to input robber ants into the program.

‘robberNestList’ functions the same as ‘antList’, although it is used for the ‘robberNest’ class, thus allowing the user to input robber nests into the program.

‘AskForDirections’ is called when the ants are within a specific distance of one another. It will allow the ants to exchange information, of where the food and nest locations are. Various ‘if’ statements have been implemented so each ant will only exchange what they know.

‘AskForTheftDirections’ functions similarly to ‘AskForDirections’, although it is a method solely for the robber ants when communicating. This will prevent exceptions from being thrown.

‘CreateAnts’ is called when the ‘createAntsButton’ is clicked. This method contains a loop that will add ants to the ‘antList’. The maximum value of this loop will be the argument, which is derived from ‘inputAntsTextBox’.

‘CreateRobberAnts’ is called when the ‘createRobberAntsButton’ is clicked. This method will add robber ants to the ‘robberAntList’. The argument of the ‘CreateRobberAnts’ method is derived from ‘inputRobberAntsTextBox’.

‘DestroyFood’ is called when the food pile reaches a certain size. It uses the integer ‘foodSize’ for reference. If it is below 1, the object is removed from the list, and replaced with null.

‘DrawAgents’ will contain the graphics for how the ants and robber ants are drawn.

‘DrawAgentsDoubleBuffering’ will contain graphics for how to draw the ants and robber ants, so that they are double buffering, and therefore can be animated.

‘DrawingPanel\_MouseClick’ will check which radio button has been clicked. It will add an item to a specific list, depending on which radio button has been checked.

‘PickUpFood’ is called when an ant is within a specific radius of a food pile. It will change the Booleans ‘knowsFoodLocation’ and ‘hasFood’ to true. The argument of this method is an integer which will locate which ant in the list must have its Booleans changed.

‘PutDownFood’ is called when an ant is within a specific radius of a nest. It will change the Boolean ‘knowsNestLocation’ to true, and ‘hasFood’ to false. The argument of this method is an integer which will locate which ant in the list must have its Booleans changed.

‘PutDownStolenFood’ functions similarly to ‘PutDownFood’, although it relates specifically to ‘RobberAntAgent’ and ‘RobberNest’. This is to prevent exceptions from occurring.

‘StealFood’ is called when a ‘RobberAntAgent’ is within a specific radius of an ‘AntAgent’. The Boolean ‘hasFood’ for the ‘AntAgent’ will change to false; and the Booleans ‘hasFood’ and ‘knowsFoodLocation’ will change to true. It will also add a position for the robber ant, so they can return to the location to steal some more food.

## AntFoodForm[Design]

This is the design of the program. The tools and controls (buttons, text boxes, etc) have been implemented at the top of the form. These include ‘inputAntsTextBox’, ‘createAntsButton’, ‘inputRobberAntsTextBox’, ‘createRobberAntsButton’, ‘label1’, ‘label2’, ‘groupBox1’, ‘nestRadioButton’, ‘foodRadioButton’, ‘robberNestRadioButton’, and ‘stopButton’. The dark grey rectangle is the drawing panel, in which the ants and any other graphics, will appear.

## AntAgent

 ‘AntAgent’ is the class for the ants in the program. It will store information for each ant that is in the ‘antList’. ‘AntAgent’ consists of four Fields, eleven Properties, and six Methods.

‘agentPosition’ are the coordinates of the ant within the drawing panel.

‘wanderPosition’ is a coordinate that is generated using ‘randomNumberGenerator’. This will set a random coordinate, that the agent must approach. This random will also change to different positions ever so often, so the ant appears to be wandering, as opposed to walking in a straight line.

‘AgentPosition’ is the position of the ant within the drawing panel. As the ant wanders around the drawing panel, the ‘AgentPosition’ will change.

‘AgentSpeed’ affects how fast the ant is wandering around the drawing panel. Increasing this value will cause the ants to move quicker.

‘ApproachRadius’ will dictate the distance an ant has to be to a nest or food pile before they call the ‘PickUpFood’ method or ‘PutDownFood’ method.

‘HasFood’ is a Boolean that will affect the behaviour of the ant. Certain ‘if’ statements have been implemented into the program to check its Boolean values, and the ant will respond accordingly. This also applies to ‘KnowsFoodLocation’ and ‘KnowsNestLocation’. These are Booleans that will become true if the ant is within a specific radius of a nest, or food pile.

‘ShouldStayWithinWorldBounds’ will prevent the ant from wandering outside of the drawing panel. If they reach the edge of the panel, their position will be reset to the other side of the panel.

‘SpecificFood’ and ‘SpecificNest’ are integers that will assign the ants behaviour to a specific nest and food pile. This prevents the ants from attempting to approach two or more objects.

‘Approach’ will cause the ant to approach a specific coordinate within the drawing panel. They will approach a specific object, depending on which Booleans are true, and which are false.

‘Wander’ will cause the ant to randomly wander around the drawing panel in no particularly direction. This will occur at the start, when the ant is required to find both the nest, and a food pile.

## RobberAntAgent

‘RobberAntAgent’ is the class for the robber ants in the program. It will store information for each robber ant that is in the ‘robberAntList’. ‘RobberAntAgent’ consists of four Fields, thirteen Properties, and six Methods. This class is very similar to the ‘AntAgent’ class, although there have been some slight adjustments in the properties. It does not have the ‘SpecificFood’ property, as it does not take food from the food piles. And it has three properties that the ‘AntAgent’ class does not contain. These properties are: ‘FoodPosX’, ‘FoodPosY’, and ‘RobberAntCounter’.

‘FoodPosX’ and ‘FoodPosY’ are points that are set to the position of an ‘AntAgent’ when they are within a specific distance of an ‘AntAgent’ that is carrying food. By setting this as a new value; it will prevent the robber ant from approaching the ant. Instead, the robber ant will return to a fixed position to steal some more food.

‘RobberAntsCounter’ is an integer that will gradually increase when the robber ant is approaching the food position. This has been implemented due to the reason that some robber ants will encounter ants that are carrying food, but are not within a chain of ants. If there was no counter, the robber ant would continuously approach this position, and thus would not contribute to its colony. When the counter reaches a specific value, the robber ant will forget the food location, and instead wander in search for another ant that is carrying food.

## Nest

‘Nest’ is a class that is for the nests within the program. It is where the ants would return to, if they are carrying food. It will store information about each ‘Nest’ in the ‘nestList’. It contains two Properties, and one Method which the constructor.

‘NestPosition’ is the position of the nest within the drawing panel. This Property will be set to the mouse cursor’s position when the user clicks on the drawing panel, with the ‘nestRadioButton’ checked.

‘NestSize’ is an integer, and will affect the size of the nest. Increasing this value, will make the nest ellipse larger.

## RobberNest

‘RobberNest’ is a class for the robber nests in the program. It is where the robber ants would return to, if they were carrying food. It will store information about each ‘RobberNest’ in the ‘robberNestList’. It contains two Properties, and one Method which the constructor.

The Properties in the ‘RobberNest’ function the same as the ones in the ‘Nest’ class. This class has been created for the user to add robber nests into the program, for the robber ants.

## Food

‘Food’ is a class for the food piles in the program. It is where the ants would collect their food from. It will store information about each ‘Food’ in the ‘foodList’. It contains two Properties, and one Method which the constructor.

‘FoodPosition’ is the position of the food pile within the drawing panel. This Property will be set to the mouse cursor’s position when the user clicks on the drawing panel, with the ‘foodRadioButton’ checked.

‘FoodSize’ is a double, and will affect the size of the food pile. When the ‘PickUpFood’ method is called, this value will decrease by 0.05. When ‘FoodSize’ reaches a specific value, the item will be removed from the ‘foodList’, and replaced with null. The ants would then forget this food location, and not interact with it in the future. They would be set to wander for some more food piles.

## Evaluation

Overall, I feel that my project has been successful; the program runs efficiently, and functions very similarly to the initial plan. Exceptions do not occur when debugging at any point, and therefore this project can be considered as a success. There has been a considerable use of Methods for specific scenarios, which therefore makes the code easier to understand. The code is also structured respectively, and this is particularly because I spent the time to plan prior to programming the application. However; there are multiple loops within the program, and thus will decrease the framerate when an excessive quantity of ants are added to their individual lists. This may be somewhat of a nuisance when the user attempts to spawn hundreds of ants into the program for testing purposes; as the quantity of ants can affect their efficiency in collecting food.

Despite all this, there is a slight logic error that occurs when the ants are collecting food. When they communicate and call the Method ‘AskForDirections’, they prioritise the food pile within the list that contains the lowest value address. This means that ants will switch from one food pile to another, if the food is located earlier in the list. For instance; an ant may encounter a food pile in the list ‘foodList’ with the address of 0; however, when it encounters other ants that already know the location of a food pile with the address of 1, it will override this. These ants would instead start approaching the food pile with the address of 0, as opposed to the food pile they were already collecting food from. I have reason to believe this is because the program is running through the loop, and assigning the first food pile in the list that is available; as the loop starts at value 0 and increases. This error is not very noticeable, particularly when there is a large amount of ants within the program. Nevertheless; they still behave accordingly.

In terms of improvement, the program contains very little on inheritance and polymorphism within the classes used. This may be an issue for when the user improves the program, and thus increases the number of classes for the ants. Currently, there are two classes for the ants: ‘AntAgent’ and ‘RobberAntAgent’. ‘RobberAntAgent’ behaves extremely similarly to ‘AntAgent’, and contains many of the same variables that ‘AntAgent’ contains. This means that there is a lot of unnecessary code within the ‘RobberAntAgent’ class. If it were to completely inherit its behaviour from the ‘AntAgent’ class, this code would not be necessary and the ‘RobberAntAgent’ class would only contain vital code that makes it unique. The reason for this, is ‘RobberAntAgent’ inherits from ‘AntAgent’, yet still contains its core code for the ‘SOFT152Sterring’. If this code is removed, an exception will occur. However; if it remains, the horizontal green lines will appear on many of the Methods, Properties and Variables, stating that they exist but are not being used. As a result, it inherits from ‘AntAgent’, but does not take complete advantage of the possibilities of inheritance. Potentially, the ‘SOFT152Steering’ in the ‘RobbberAntAgent’ class could be adapted to the aggressive behaviour of the robber ants.