AI Project Report

Made by Jordan Munk and Sidney de Geus

ALGORITHMS A.I. FOR GAMES

Auteur :
Jordan Munk s1113349
Sidney de Geus s1113288
Class: ICTGPa

Teacher :
H. Keuning
hw.keuning@windesheim.nl

Version 1.0
June 25, 2017

Preface

This report contains an explanation of our implementation for the AI project. In approximately 3-5 (without diagrams) pages we discuss a Steering, Path Planning, Behaviour and Fuzzy Logic. We will describe all the content using class diagrams. At the end of the report we give a summary how the project went.

Contents

List of Figures

Chapter 1

Steering

1.1 Steering behaviours

In lecture 3 we discussed 10 steering behaviours. Seek and Flee, Hide, Pursue and Evade, Path Following, Arrive, Wall Avoidance, Wander, interpose, Avoid obstacles and Offset pursuit. Our weekly assignment was to implement two simple steering behaviours and one advanced steering behaviour. We implemented the simple behaviour Wander and Seek and for the advanced behaviour we implemented Obstacle Avoidance. At last we needed to implement the behaviour Explore.

1.1.1 Wander

For the wander steering behaviour we tried to produce a realistic "casual" movement. We used for the wander the Unity's random generator. It will generate a random integer between the given wander radius.
The following class diagram is used for the steering behaviour, wander (fig. 1.1).

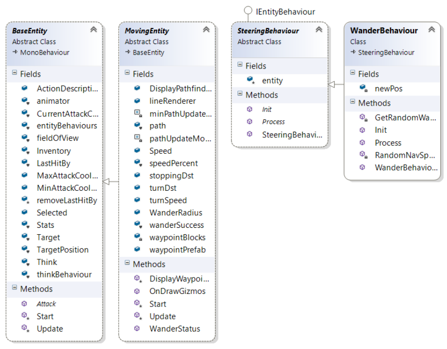


Figure 1.1: Class diagram of Wander

The Steering behaviours can also be combined. You can combine multiple behaviours using a force. Each steering behaviour will output a force and will be calculated. You can use Weighted truncated sum, Prioritization and Prioritized dithering to calculate the force.

1.1.2 Seek

The seek behaviour is implemented in a few classes. At first it starts in the Think behaviour, using Fuzzy Logic it determines if the player wants to chase another player (fig. 1.2).

```
if (FuzzyGetDesirabilitySeek(human.Stats.Hunger, human.Stats.Money) > 85) {
    if (human.Think.CurrentAction().GetType() != typeof(SeekTarget)) {
        SeekTarget seekTarget = new SeekTarget(human);
        return seekTarget;
    }
}
```

Figure 1.2: Seek with Fuzzy

If it wants to seek, it will use an Atomic action, Seektarget to seek a target. In the seektarget class the character will chase a random character. If it is in range it will chase it. We use the following class diagram for Seek (fig. 1.3).

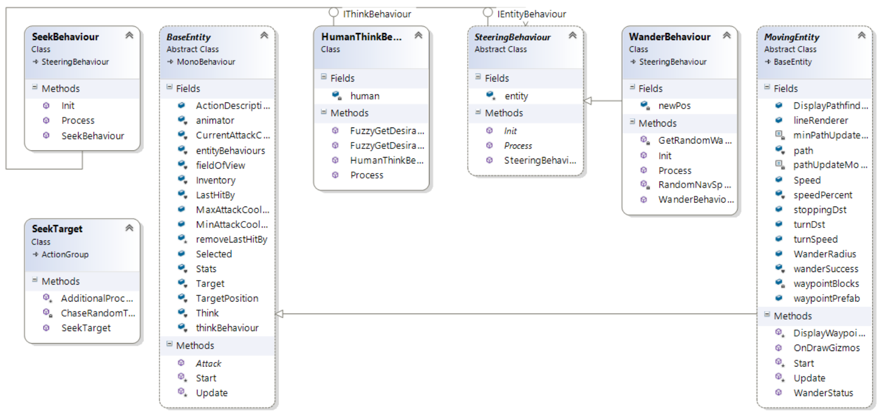


Figure 1.3: Class diagram of Seek

1.1.3 Obstacle avoidance

For the advanced steering behaviour we implemented Obstacle avoidance. We didn't make a obstacle avoidance class but we used it in the path finding. In unity you can give objects a mask. We gave the obstacles the mask Unwalkable. In the path finding we added a few if statements to check if a node is walk able or not. If it is unwalkable we avoid the obstacle and change the direction where the character goes. In (fig.1.4) you can see the unwalkable mask on the obstacles (in red).

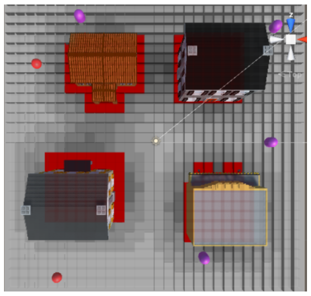


Figure 1.4: Unwalkable mask on obstacles

In the class Node we added a bool walkable. True if it is walkable and false if it is unwalkable. In the grid class we are creating the grid. In the grid we check if there is a obstacle on it. If there is a obstacle on it the grid node gets a unwalkable mask on it. (fig. 1.5) for the class diagram for Obstacle Avoidance.

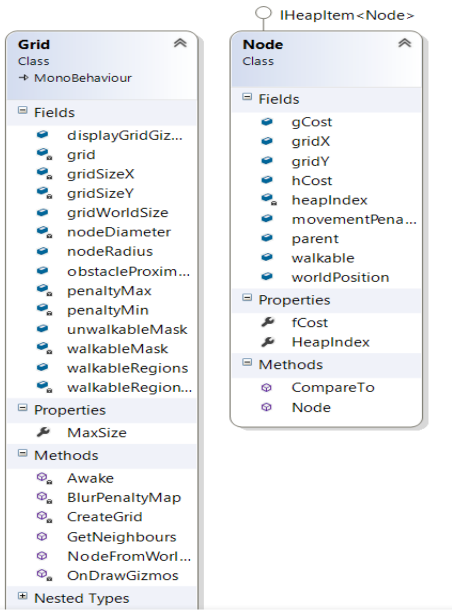


Figure 1.5: Class diagram of Obstacle Avoidance

1.1.4 Follow Path

For the Follow Path steering behaviour we used Path Finding. For an in depth documentation go to the Path Finding section.

1.2 Issues

We had one problem with the Wander behaviour. Because we used randoms, sometimes if the character was in the corner, he tried to go outside the map. We fixed this using while loops (see Path finding for more information).

1.3 Result evaluation

Everything went smooth and we didn't have any complaints. What we want to do if we had more time is to implement the Explore more. We didn't have enough time to implement the feature fully. We are happy about the wander end result. It looks like it actually is a player that casually walks.

Chapter 2

Path Planning

2.1 Path Planning

Path Planning is an important section of our game. Our game is a simulation that will do certain things. But without Path Planning they can't do those certain things. Path Planning is to get a player from point A to point B in the fastest and human-like way. To get the player as fast as possible to point B we used the A* search algorithm. A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

2.1.1 A*

We used the search Algorithm A*. The character from our game will go from point A to point B using way points. In the following figure Bridget (our character) is at her begin point (point A) her target is the black way point on the screen. (fig. 2.1). It will use Smooth path finding (more explanation later) to get to the target.

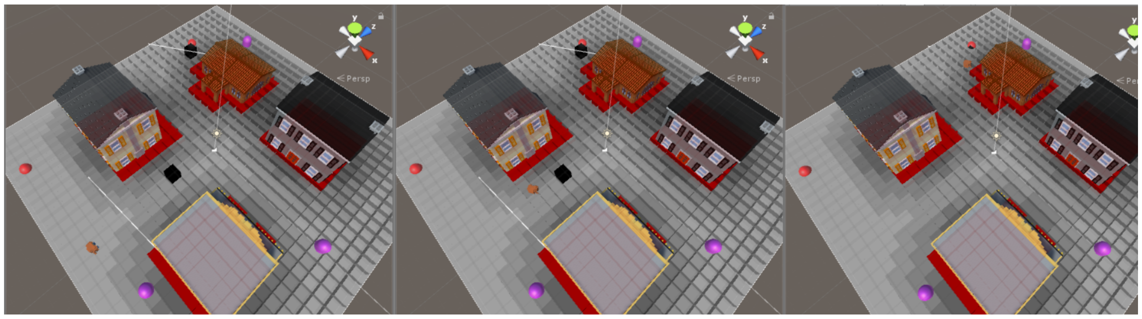


Figure 2.1: Mike goes from point A to point B using A*

2.1.2 Code explanation

For the A* Search Algorithm we started making the Grid. The Grid contains a 30 by 30 world size. Every 0.5 radius there is a Node places on the grid. Every Node can have mask, the grass for instance haves a Grass layer. If you want an unwalkable mask, where the character can't go, you need to add the layer Unwalkable to the Node.

What A* Search Algorithm does is that at each step it picks the node according to the value fCost, which is a parameter equal to the sum of two other parameters, gCost and hCost. So we added to every Node two Integers, the gCost and the hCost. To make the code faster we used Heap sort to sort the nodes in the Grid. The Grid looks like the following (fig. 2.2).

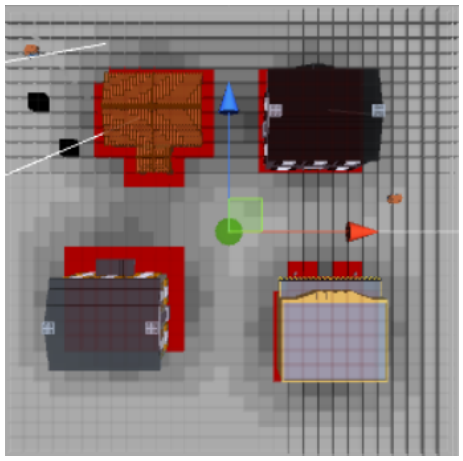


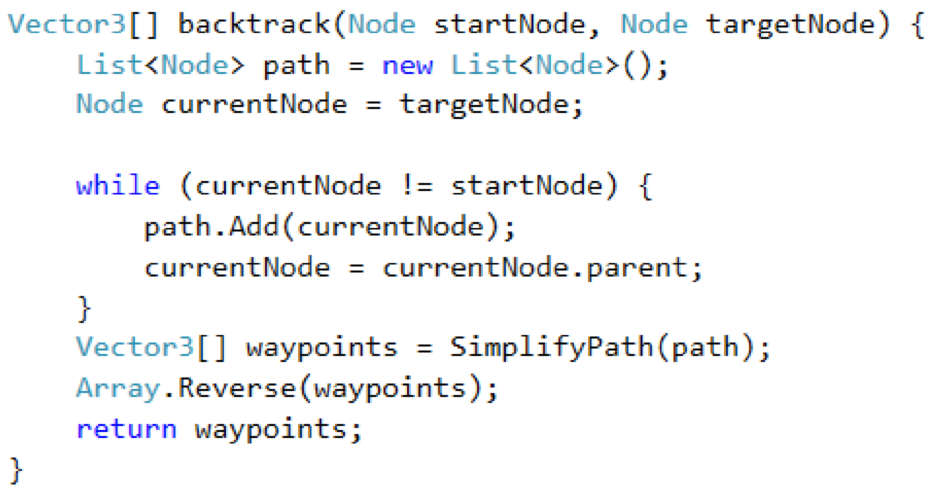
Figure 2.2: Grid with nodes

After we added the Grid, we started working on the Path finding itself. The character haves a few variables to determine how the character will move, like speed, turning distance, turning speed, stopping distance etc. To make the player move on the nodes we calculated where the character will pick the node to move on. At each step it will picks the node having the lowest fCost, and process that node. fCost is the sum of gCost and hCost. These mean the following:

gCost = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
hCost = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.).

After we added everything to the nodes and gave the character the variables, we started working on implementing the Path finding code. In Pathfinding.cs we added a FindPath function. In the FindPath function we started initializing the starting node and the target node. After that we added the starting node in a heap sort list. Now we added a while loop to check if the list is not empty. In this while loop we checked if the mask is walk-able or not and added some values to hCost, gCost and fCost using some math.

One of the methods we use for path finding is backtracking. We used backtracking to check if the character successfully walked to the target position. We used the following backtrack code (fig. 2.3).



```
Vector3[] backtrack(Node startNode, Node targetNode) {
    List<Node> path = new List<Node>();
    Node currentNode = targetNode;

    while (currentNode != startNode) {
        path.Add(currentNode);
        currentNode = currentNode.parent;
    }
    Vector3[] waypoints = SimplifyPath(path);
    Array.Reverse(waypoints);
    return waypoints;
}
```

Figure 2.3: Code of our backtracking

Because we have more characters, we used Threading to make it possible to have more path finding algorithms working on the same time. We added all the Path results in a Queue and Dequeued every frame when there are items in the queue.

The following class diagram is being used. (fig. 2.4).

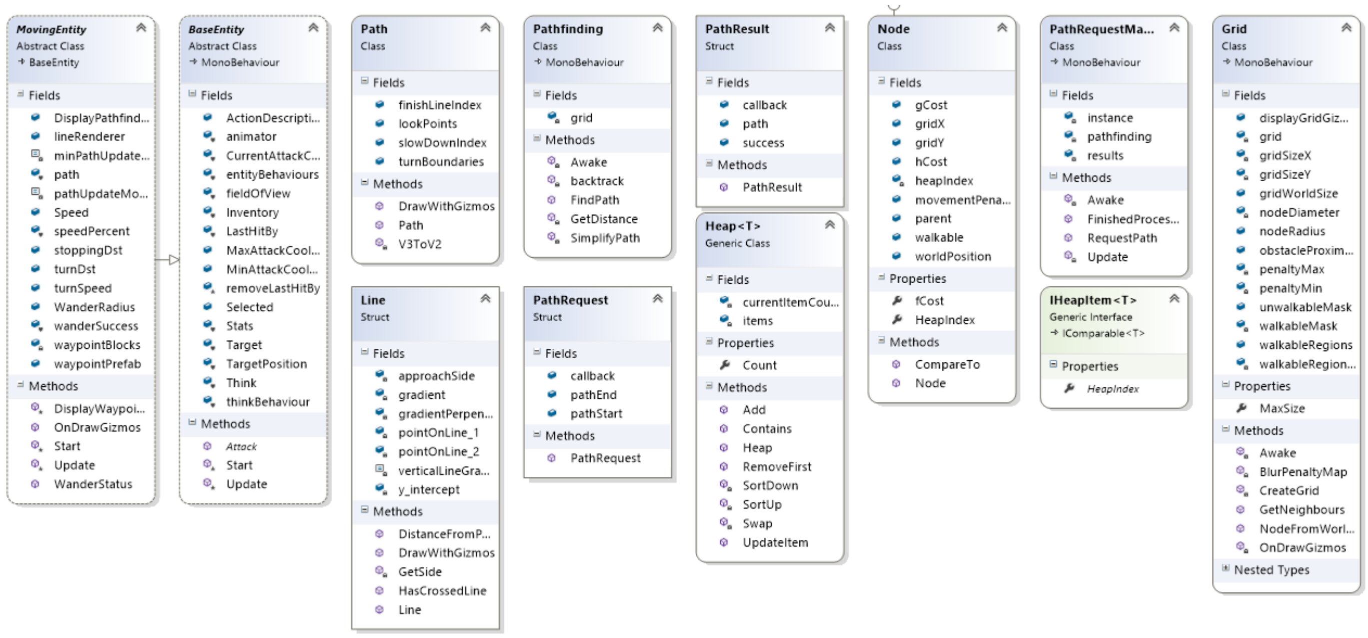


Figure 2.4: Class diagram of Path planning

2.1.3 Path Smoothing

Before we did path smoothing we used a simple path finding system. Instead of moving in a direct line we are translating and rotating the character to move to a certain point. Every way point haves a line on the screen. The player will now move more to the line so it won't go directly to the way point. It will move now a lot smoother. The new translation/rotation looks like the following (fig. 2.5).

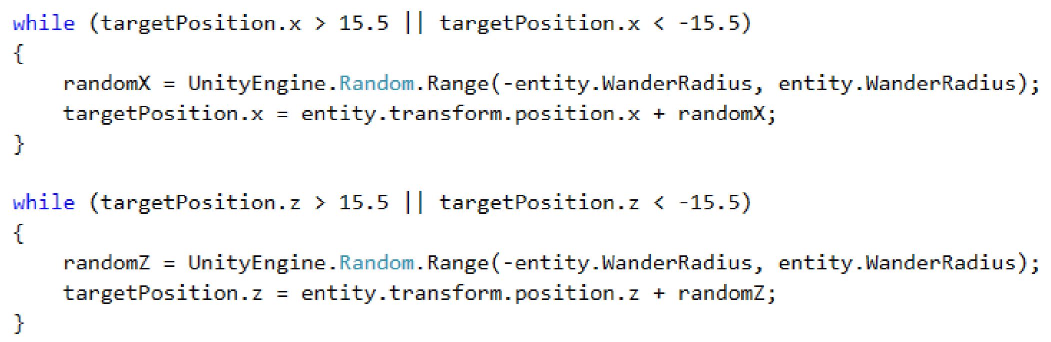
```
Quaternion targetRotation = Quaternion.LookRotation(entity.path.lookPoints[pathIndex] - entity.transform.position);
entity.transform.rotation = Quaternion.Lerp(entity.transform.rotation, targetRotation, Time.deltaTime * entity.turnSpeed);
entity.transform.Translate(Vector3.forward * Time.deltaTime * entity.Speed * entity.speedPercent, Space.Self);
```

Figure 2.5: Translation and Rotation code

2.2 Issues

Path Finding was the hardest part to program. It uses a lot of mechanics. After the path smoothing update it was a little bit buggy. Sometimes the character will move a bit slower than normal. This was hard to fix because we didn't know where the problem was, because we didn't know how A* worked before the project so it was hard to understand.

In the resit we did some more research to fix the problem. After we added path visualization. We saw that the character wants to get outside the plane. Because of that problem the player didn't move so we did some "ugly" fixes to the Wander behaviour. We used 2 while loops to control if the NewPostion will go outside the plane. If it did, we just made a NewPosition till the NewPosition is inside the plane (fig. 2.6).



```
while (targetPosition.x > 15.5 || targetPosition.x < -15.5)
{
    randomX = UnityEngine.Random.Range(-entity.WanderRadius, entity.WanderRadius);
    targetPosition.x = entity.transform.position.x + randomX;
}

while (targetPosition.z > 15.5 || targetPosition.z < -15.5)
{
    randomZ = UnityEngine.Random.Range(-entity.WanderRadius, entity.WanderRadius);
    targetPosition.z = entity.transform.position.z + randomZ;
}
```

Figure 2.6: The While Loops

2.3 Result evaluation

If you play games, u know that the path finding is never perfect. Most of the time the AI is a bit weird. There is still a huge difference between AI and a human. The same with our system. Sometimes is does weird things that we don't understand. In the re

Chapter 3

Behaviours

3.1 Composite Behaviour

For our application we used the composite behaviour. A composite pattern is used were we need to treat a group of objects in a similar way as an single object. (fig. 3.1) shows a class diagram that illustrate the Composite Pattern's structure.

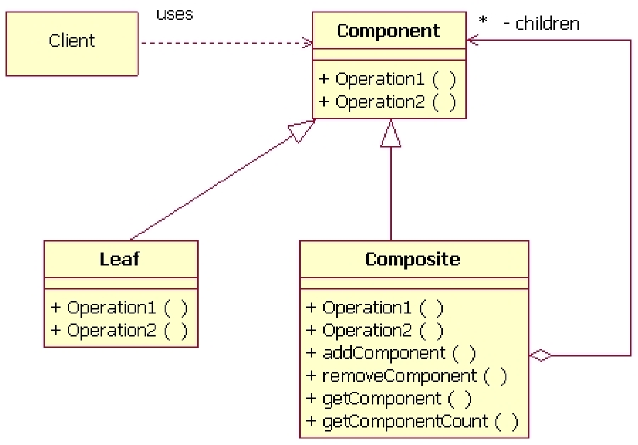


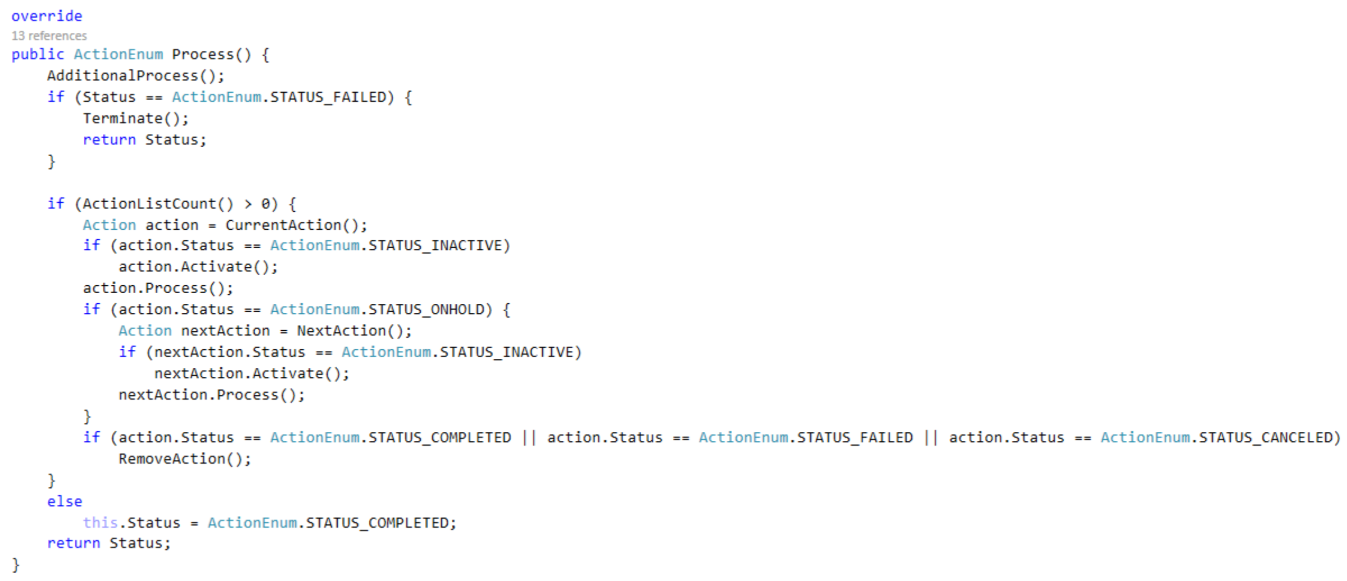
Figure 3.1: Class diagram of Composite pattern

3.2 Our implementation

For this project we have implemented the composite pattern for the AI Logic. There are 2 kinds of actions, ActionGroups (Composite) and single, specific action (Atomic). Every ActionGroup contains one, or multiple actions which can be ActionGroups or single actions as well.

The Action class is an abstract base class that contains things such as a base constructor so that all actions have to follow the principle of calling the base constructor upon creation. This way we prevent from not putting a status upon creation.

ActionGroup is an absract class as well, but is aimed purely at Actions that are composite such as Think. The ActionGroup class implements the Process() from the Action class and defines a basic algorithm that works exactly the same for all composite classes. (fig. 3.2)



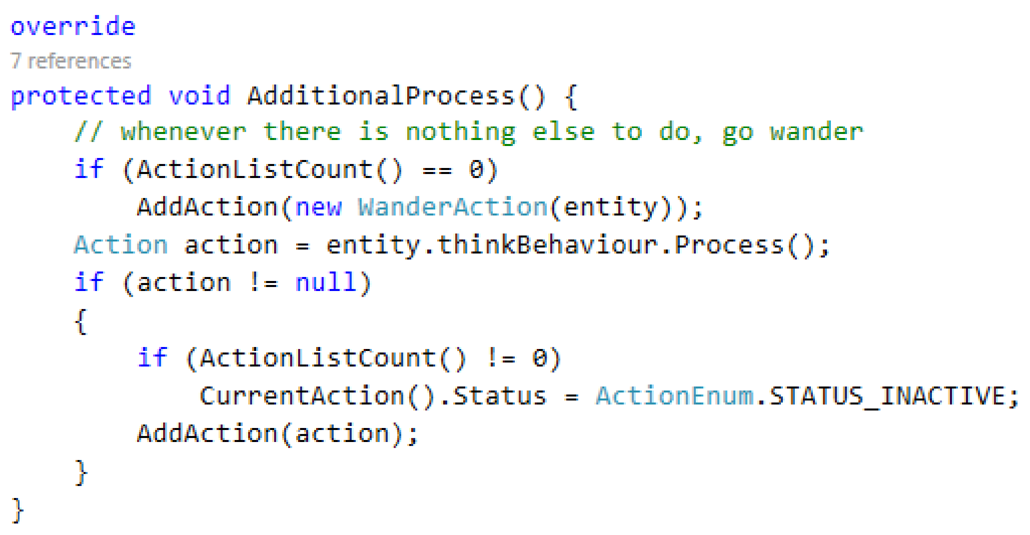
```
override
13 references
public ActionEnum Process() {
    AdditionalProcess();
    if (Status == ActionEnum.STATUS_FAILED) {
        Terminate();
        return Status;
    }

    if (ActionListCount() > 0) {
        Action action = CurrentAction();
        if (action.Status == ActionEnum.STATUS_INACTIVE)
            action.Activate();
        action.Process();
        if (action.Status == ActionEnum.STATUS_ONHOLD) {
            Action nextAction = NextAction();
            if (nextAction.Status == ActionEnum.STATUS_INACTIVE)
                nextAction.Activate();
            nextAction.Process();
        }
        if (action.Status == ActionEnum.STATUS_COMPLETED || action.Status == ActionEnum.STATUS_FAILED || action.Status == ActionEnum.STATUS_CANCELED)
            RemoveAction();
    }
    else
        this.Status = ActionEnum.STATUS_COMPLETED;
    return Status;
}
```

Figure 3.2: Implementation of Action Group

All actions have a Status, and this status is defined by a couple of enums that we created. This basic algorithm keeps processing the first action on the linked list as long as the count is bigger than zero and then it also checks whether the action is on hold (and thus perform the one after it) or whether it's completed, failed or canceled in order to remove it from the stack again. If there is no action in the list, this entire action will be considered completed as well.

At the start of the method we make use of the template pattern in order to add additional functionality to each composite action (such as think) so that there is the option to add custom logic for each action.

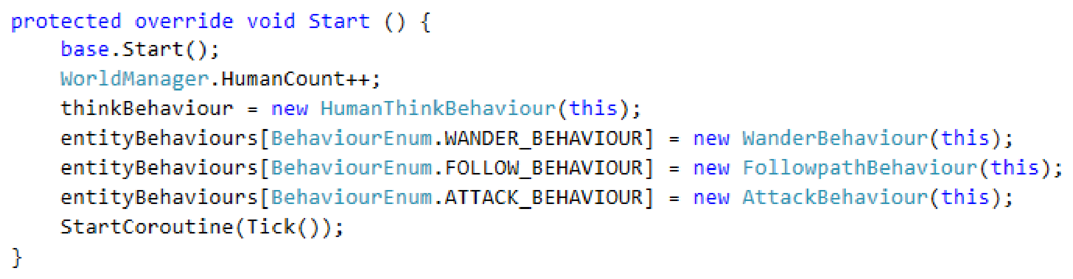
This is how the Think class works, because of the template pattern we can put the additional code to make it behave the same, yet in a unique way specifially based on what type of actionit is. The think class is never ending though, and to solvethis, we made it so that, if the character has no more actions left, he will automatically start wandering right away (fig. 3.3).



```
override
7 references
protected void AdditionalProcess() {
    // whenever there is nothing else to do, go wander
    if (ActionListCount() == 0)
        AddAction(new WanderAction(entity));
    Action action = entity.thinkBehaviour.Process();
    if (action != null)
    {
        if (ActionListCount() != 0)
            CurrentAction().Status = ActionEnum.STATUS_INACTIVE;
        AddAction(action);
    }
}
```

Figure 3.3: Implementation of Think

We also made use of a strategy pattern in order to make every entity have it's unique kind of behaviour by defining the think behaviour upon construction of the object. Because of this almost all core decisions come from the HumanThinkBehaviour class. (Such as all the fuzzy desirability etc.)(fig. 3.4).



```
protected override void Start () {
    base.Start();
    WorldManager.HumanCount++;
    thinkBehaviour = new HumanThinkBehaviour(this);
    entityBehaviours[BehaviourEnum.WANDER_BEHAVIOUR] = new WanderBehaviour(this);
    entityBehaviours[BehaviourEnum.FOLLOW_BEHAVIOUR] = new FollowpathBehaviour(this);
    entityBehaviours[BehaviourEnum.ATTACK_BEHAVIOUR] = new AttackBehaviour(this);
    StartCoroutine(Tick());
}
```

Figure 3.4: Implementation of Strategy pattern entities

.

3.3 Class Diagram

fig. 3.5 shows the class diagram of our composite pattern.

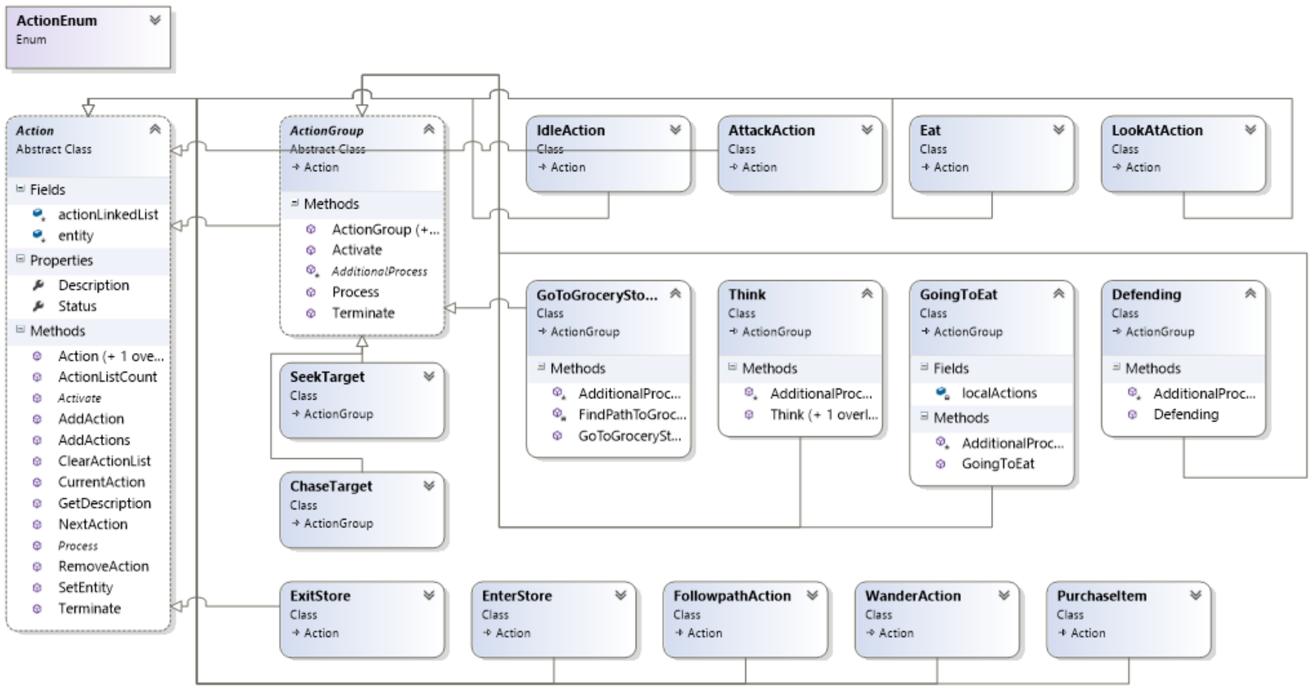


Figure 3.5: Class diagram of our Composite pattern

3.4 Issues

In the game there sometimes is a bug or a problem, however, and that is that it sometimes stacks a few actions endlessly while not removing an older one. This is why the action list looks very big, but it just doesn't remove super old redundant or no longer valid actions.

Chapter 4

Fuzzy Logic

4.0.1 Fuzzy Logic

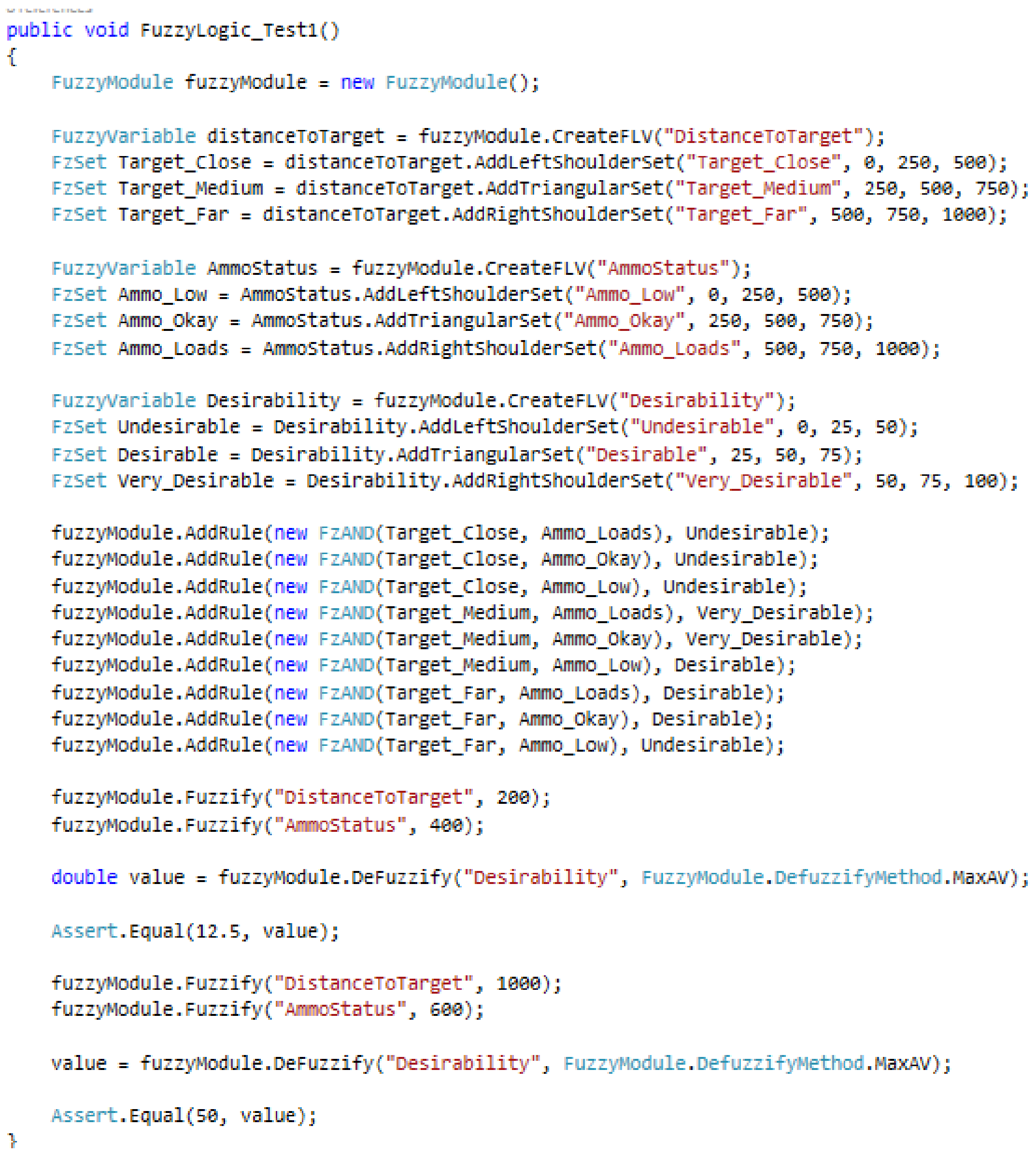
We used Fuzzy Logic for our project to form a value (desirability) from 0 to 1. You can form the value using multiple values. For instance you can use Hunger, Money and Health to make a desirability for going to the grocery store. If it is very high, he is hunger and got enough food and if it is very low, he is poor or not hungry.

4.0.2 Our implementation

We used the given class diagram to implement our Fuzzy Logic system. If you want to make a Fuzzy, you need to start making a Fuzzy Module. On the Fuzzy Module you can add multiple FZL, a FZL is a Fuzzy Variable like Hunger or DistanceToTarget. On the FuzzyVariable you need to add three Fuzzy Sets, these FzSets contains the Left Shoulder, TriangularSet and the Right Shoulder. After you added the FuzzyVariables, you need to add rules. For example, if you have hunger and you are rich, you can buy something in the Grocery Store. After you added the rules you can Fuzzify the fuzzy variables and defuzzify the Desirability.

4.0.3 Testcase

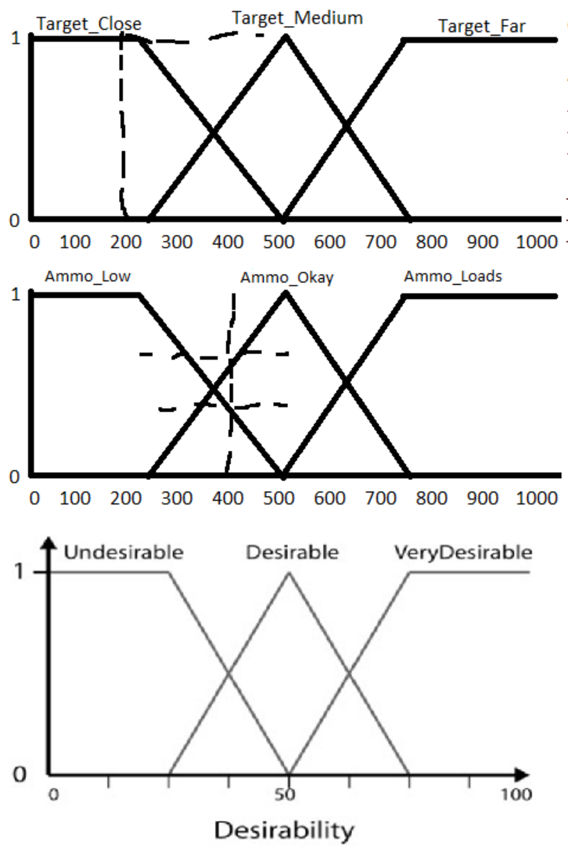
We made some Unit Tests, to test our Fuzzy Logic. We used the folowing Unit test (fig. 4.1).



```
public void FuzzyLogic_Test1()
{
    FuzzyModule fuzzyModule = new FuzzyModule();

    FuzzyVariable distanceToTarget = fuzzyModule.CreateFLV("DistanceToTarget");
    FzSet Target_Close = distanceToTarget.AddLeftShoulderSet("Target_Close", 0, 250, 500);
    FzSet Target_Medium = distanceToTarget.AddTriangularSet("Target_Medium", 250, 500, 750);
    FzSet Target_Far = distanceToTarget.AddRightShoulderSet("Target_Far", 500, 750, 1000);

    FuzzyVariable AmmoStatus = fuzzyModule.CreateFLV("AmmoStatus");
    FzSet Ammo_Low = AmmoStatus.AddLeftShoulderSet("Ammo_Low", 0, 250, 500);
    FzSet Ammo_Okay = AmmoStatus.AddTriangularSet("Ammo_Okay", 250, 500, 750);
    FzSet Ammo_Loads = AmmoStatus.AddRightShoulderSet("Ammo_Loads", 500, 750, 1000);

    FuzzyVariable Desirability = fuzzyModule.CreateFLV("Desirability");
    FzSet Undesirable = Desirability.AddLeftShoulderSet("Undesirable", 0, 25, 50);
    FzSet Desirable = Desirability.AddTriangularSet("Desirable", 25, 50, 75);
    FzSet Very_Desirable = Desirability.AddRightShoulderSet("Very_Desirable", 50, 75, 100);

    fuzzyModule.AddRule(new FzAND(Target_Close, Ammo_Loads), Undesirable);
    fuzzyModule.AddRule(new FzAND(Target_Close, Ammo_Okay), Undesirable);
    fuzzyModule.AddRule(new FzAND(Target_Close, Ammo_Low), Undesirable);
    fuzzyModule.AddRule(new FzAND(Target_Medium, Ammo_Loads), Very_Desirable);
    fuzzyModule.AddRule(new FzAND(Target_Medium, Ammo_Okay), Very_Desirable);
    fuzzyModule.AddRule(new FzAND(Target_Medium, Ammo_Low), Desirable);
    fuzzyModule.AddRule(new FzAND(Target_Far, Ammo_Loads), Desirable);
    fuzzyModule.AddRule(new FzAND(Target_Far, Ammo_Okay), Desirable);
    fuzzyModule.AddRule(new FzAND(Target_Far, Ammo_Low), Undesirable);

    fuzzyModule.Fuzzify("DistanceToTarget", 200);
    fuzzyModule.Fuzzify("AmmoStatus", 400);

    double value = fuzzyModule.DeFuzzify("Desirability", FuzzyModule.DefuzzifyMethod.MaxAV);

    Assert.Equal(12.5, value);

    fuzzyModule.Fuzzify("DistanceToTarget", 1000);
    fuzzyModule.Fuzzify("AmmoStatus", 600);

    value = fuzzyModule.DeFuzzify("Desirability", FuzzyModule.DefuzzifyMethod.MaxAV);

    Assert.Equal(50, value);
}
```

Figure 4.1: Test case of our Fuzzy Logic

In the figure above you can see that I used 3 FuzzyVariables, 2 for the variables to check and 1 for the desirability. After that I created some rules, for example if the Target is close and you have a lot of ammo, you don't want to attack. For the values DistanceToTarget = 200 and AmmoStatus = 400, you get the Desirability of 12.5. I am going to check this manually. In the following figure you see the FLVS and the MaxAv manual calculation (fig. 4.2).



Character A got:
DistanceToTarget = 200
AmmoStatus = 400

Ttarget_close(200) = 1
Ttarget_medium(200) = 0
Ttaret_far(200) = 0

Tammo_low(400) = 0.4
Tammo_okay(4000 = 0.6
Tammo_loads(400) = 0

	Target_Close	Target_Medium	Target_Far
Ammo_Low	Undesirable 0.4	Desirable 0	Undesirable 0
Ammo_Okay	Undesirable 0.6	Very_Desirable 0	Desirable 0
Ammo_Loads	Undesirable 0	Very_Desirable 0	Desirable 0

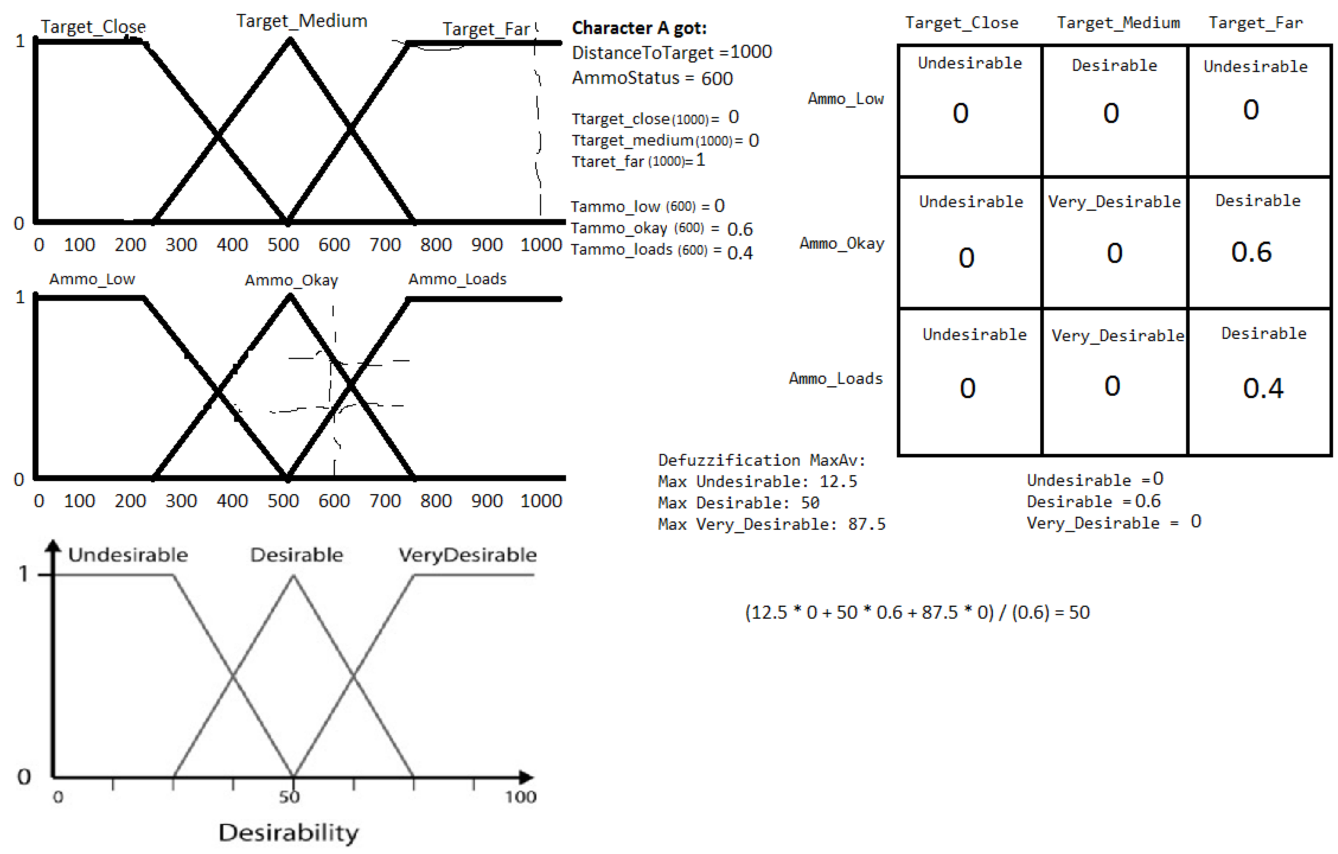
```
Defuzzification MaxAv:
Max Undesirable: 12.5
Max Desirable: 50
Max Very_Desirable: 87.5
```

```
Undesirable = 0.6
Desirable = 0
Very_Desirable = 0
```

```
MaxAv = (12.5 * 0.6 + 50 * 0 + 87.5 * 0)
/ (0.6) = 12.5
```

Figure 4.2: Manual test case 1

To make sure our Fuzzy Logic is working, I did one more manually. With DistanceToTarget = 1000 and AmmoStatus = 600. The end result on our own fuzzy logic system it gives a MaxAf of 50. I am gonna check this manually. The end result of the manually test is the same as our own fuzzy logic system.(fig. 4.3).



Character A got:
DistanceToTarget =1000
AmmoStatus = 600

Ttarget_close(1000)= 0
Ttarget_medium(1000)= 0
Ttaret_far (1000)= 1

Tammo_low (600) = 0
Tammo_okay (600) = 0.6
Tammo_loads (600) = 0.4

	Target_Close	Target_Medium	Target_Far
Ammo_Low	Undesirable 0	Desirable 0	Undesirable 0
Ammo_Okay	Undesirable 0	Very_Desirable 0	Desirable 0.6
Ammo_Loads	Undesirable 0	Very_Desirable 0	Desirable 0.4

```
Defuzzification MaxAv:
Max Undesirable: 12.5
Max Desirable: 50
Max Very_Desirable: 87.5
```

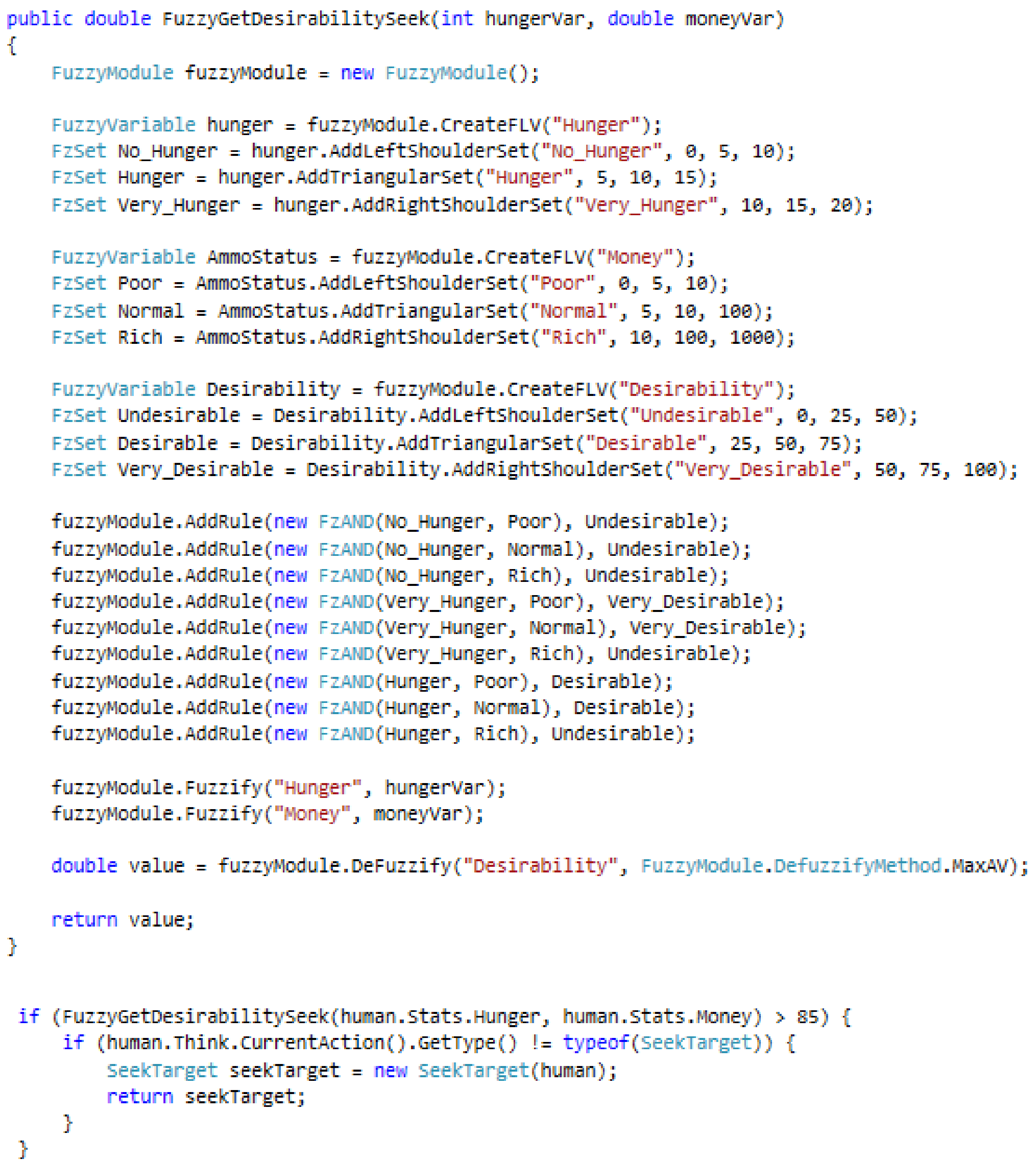
```
Undesirable =0
Desirable =0.6
Very_Desirable = 0
```

(12.5 * 0 + 50 * 0.6 + 87.5 * 0) / (0.6) = 50

Figure 4.3: Manual test case 2

4.0.4 Implementation example

We used Fuzzy Logic for two cases. First we added a Fuzzy Logic to make a character chase another one based on hunger and money. After that we added another Fuzzy Logic to go to the grocery store, this was based one money, hunger and health. In the following example we used Fuzzy on seek.(fig. 4.4).



```
public double FuzzyGetDesirabilitySeek(int hungerVar, double moneyVar)
{
    FuzzyModule fuzzyModule = new FuzzyModule();

    FuzzyVariable hunger = fuzzyModule.CreateFLV("Hunger");
    FzSet No_Hunger = hunger.AddLeftShoulderSet("No_Hunger", 0, 5, 10);
    FzSet Hunger = hunger.AddTriangularSet("Hunger", 5, 10, 15);
    FzSet Very_Hunger = hunger.AddRightShoulderSet("Very_Hunger", 10, 15, 20);

    FuzzyVariable AmmoStatus = fuzzyModule.CreateFLV("Money");
    FzSet Poor = AmmoStatus.AddLeftShoulderSet("Poor", 0, 5, 10);
    FzSet Normal = AmmoStatus.AddTriangularSet("Normal", 5, 10, 100);
    FzSet Rich = AmmoStatus.AddRightShoulderSet("Rich", 10, 100, 1000);

    FuzzyVariable Desirability = fuzzyModule.CreateFLV("Desirability");
    FzSet Undesirable = Desirability.AddLeftShoulderSet("Undesirable", 0, 25, 50);
    FzSet Desirable = Desirability.AddTriangularSet("Desirable", 25, 50, 75);
    FzSet Very_Desirable = Desirability.AddRightShoulderSet("Very_Desirable", 50, 75, 100);

    fuzzyModule.AddRule(new FzAND(No_Hunger, Poor), Undesirable);
    fuzzyModule.AddRule(new FzAND(No_Hunger, Normal), Undesirable);
    fuzzyModule.AddRule(new FzAND(No_Hunger, Rich), Undesirable);
    fuzzyModule.AddRule(new FzAND(Very_Hunger, Poor), Very_Desirable);
    fuzzyModule.AddRule(new FzAND(Very_Hunger, Normal), Very_Desirable);
    fuzzyModule.AddRule(new FzAND(Very_Hunger, Rich), Undesirable);
    fuzzyModule.AddRule(new FzAND(Hunger, Poor), Desirable);
    fuzzyModule.AddRule(new FzAND(Hunger, Normal), Desirable);
    fuzzyModule.AddRule(new FzAND(Hunger, Rich), Undesirable);

    fuzzyModule.Fuzzify("Hunger", hungerVar);
    fuzzyModule.Fuzzify("Money", moneyVar);

    double value = fuzzyModule.DeFuzzify("Desirability", FuzzyModule.DefuzzifyMethod.MaxAV);

    return value;
}


 if (FuzzyGetDesirabilitySeek(human.Stats.Hunger, human.Stats.Money) > 85) {
     if (human.Think.CurrentAction().GetType() != typeof(SeekTarget)) {
         SeekTarget seekTarget = new SeekTarget(human);
         return seekTarget;
     }
 }
```

Figure 4.4: Fuzzy Logic Example

Chapter 5

Conclusion

5.1 Difficulties

For out first try the project didn't go as well as we planned. We had some difficulties on the way. Most of the difficulties we fixed. But some are still there. The big one is Fuzzy Logic. We didn't get that working because we didn't have enough time and it was pretty hard to understand it correctly. Another difficulties was the path finding. This was hard to program because it always did something weird. Most of time it didn't go smoothly and we had a hard time to fix it.

When we did the resit, we changed a lot of our code. This was pretty difficult because some things where really messy.

5.2 Improvements

AI is never perfect. If you play our simulation for some time, the AI will do inhumanly things. If we had more time to program we'd like to improve our AI. Sometimes the path finding is not doing the smartest decisions and we like to make it a bit smoother.
Another big improvement is to fix the Fuzzy Logic. This didn't work in our implementation and we'd like to make it work. Because it is an simulation of humans that live you can make the game as large as you want. We want to keep it simple like we have it now. So we only want to fix our bugs and make our code perfect.

When we did the resit, we fixed most of these improvements. We finally improved Fuzzy and Path finding. The improvement for the resit is to make everything a little bit prettier. Maybe add some roads or trees, but the functional stuff is working good enough.

Bibliography

[1] Craig Reynolds on Steering Behaviors For Autonomous Characters,
`http://www.red3d.com/cwr/steer/`

[2] Envato tuts on Steering Behaviors
`https://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors-gamedev-`

[3] Patrick Lester on Path finding using A*,
`http://www.policyalmanac.org/games/aStarTutorial.htm`

[4] Red Blob Games on Path finding using A*,
`http://theory.stanford.edu/ amitp/GameProgramming/`

[5] Information about Design Patterns,
`https://sourcemaking.com/design_patterns`

[6] Official Unity User Manual,
`https://docs.unity3d.com/Manual/`

[7] Official Unity Forum,
`http://answers.unity3d.com/`