3.2. IMPLEMENTACIÓN DEL SISTEMA DE UTILIDAD

1. **Factor**

Esta clase abstracta es el padre de todos los factores, contiene la función virtual *getValue(),* que devuelve la utilidad asociada al factor.

```
public virtual float getValue()
{
    return 0;
}
```

2. **Curve**

Esta clase abstracta hereda de **Factor**. Contiene un **Factor** accesible únicamente por los hijos. Es la clase padre de todas las curvas.

```
protected Factor factor;
public Curve(Factor f)
{
      this.factor = f;
}
```

3. **Fusion**

Esta clase abstracta hereda de **Factor**. Contiene una lista de **Factor** accesible únicamente por los hijos. Es la clase padre de todos los factores de fusión.

```
protected List<Factor> factors;
public Fusion(List<Factor> factors)
{
      this.factors = factors;
}
```

4. **LeafVariable**

Esta clase que hereda directamente de **Factor** es aquella que contiene la referencia a la variable del entorno, usando para ello una referencia a función que devuelve un valor flotante "*Func<float>*" de C#. Esta clase recibe en su constructor la referencia a la función y su valor máximo y mínimo para posteriormente normalizar el valor en la función *getValue().* Se ha intentado sustituir la referencia a función por un puntero o referencia a valor flotante pero las particularidades de C# no me han permitido declarar en la clase una referencia a un valor, por lo que he optado por usar la referencia a función.

```
private float maxValue, minValue;
private Func<float> getVariable;

public LeafVariable(Func<float> getVariable, float maxValue, float
minValue)
{
      this.getVariable = getVariable;
      this.maxValue = maxValue;
      this.minValue = minValue;
}
public override float getValue()
{
      float factor = (getVariable() - minValue) / (maxValue -
minValue);
      if (factor < 0) factor = 0; if (factor > 1) factor = 1;

      return factor;
}
```

5. LinearCurve

Esta clase hereda de la clase abstracta **Curve**. Esta clase genera una función lineal que devuelve un valor de utilidad en función de la pendiente y la ordenada en el origen introducida por el usuario. El constructor tiene valores por defecto para la pendiente y la ordenada en el origen que pueden ser sustituidos por el valor que requiera el usuario.

```
private float m, c;
public LinearCurve(Factor f, float pend = 1, float ind = 0) : base(f)
 {
      this.m = pend;
      this.c = ind;
 }

 public override float getValue()
 {
      return m*factor.getValue() + c;
 }
```

6. ExpCurve

Esta clase hereda de la clase abstracta **Curve**. Esta clase genera una función exponencial que devuelve un valor de utilidad en función del exponente introducido por el usuario. El constructor tiene valores por defecto para el desplazamiento horizontal y vertical que pueden ser sustituidos por el valor que requiera el usuario.

```
private float k, c, b;
public ExpCurve(Factor f, float exp = 1, float despX = 0, float despY
= 0) : base(f)
{
      this.k = exp;
      this.c = despX;
      this.b = despY;
      this.factor = f;
}
public override float getValue()
{
      return (float) (Math.Pow(factor.getValue() - c, k) + b);
}
```

7. LinearPartsCurve & Point2D

Point2D es una clase sencilla que simplemente contiene dos variables flotantes, representan la coordenada ‘x’ e ‘y’ de un punto en el espacio 2D de las curvas. Se usan para definir las curvas lineales a trozos.

```
public float x, y;
public Point2D(float x, float y)
{
      this.x = x;
      this.y = y;
}
```

LinearPartsCurve es una clase que hereda de **Curve**. Contiene una lista de **Point2D** que se usan para definir el punto donde comienza o acaba una función lineal. Al recibir la lista de puntos en el constructor, éstos se ordenan de menor a mayor en función de su coordenada ‘x’ para que los puntos estén ordenados de izquierda a derecha.

```
private List<Point2D> points;
public LinearPartsCurve(Factor f, List<Point2D> points) : base(f)
{
      this.setPoints(points);
}

public void setPoints(List<Point2D> points)
{
      this.points = points;

      this.points.Sort((p1, p2) =>
      {
```

```
            return p1.x.CompareTo(p2.x);
      });
}
```

La función *getValue()* lo que hace es comprobar dónde se encuentra el **Factor** de entrada en el eje x, encuentra los puntos entre los que está contenido y realiza una interpolación lineal para obtener el valor de utilidad. Aunque parezca que *getValue()* es una función complicada, simplemente se verifica si el **Factor** se encuentra entre dos puntos o si se encuentra al principio o al final de la 'gráfica', sin estar contenido entre dos puntos.

```
public override float getValue()
{
      float returnValue = 0.0f;
      float x = factor.getValue();


      for(int i = 0; i < points.Count; i++)
      {
            float xPoint = points[i].x;
            if (i == 0 && x < xPoint) { returnValue = points[i].y;
break; };
            if ((i == points.Count - 1) && x > xPoint) { returnValue =
points[i].y; break; };
            if (x == xPoint) { returnValue = points[i].y; break; }

            if (x > xPoint && x < points[i + 1].x)
            {
                  returnValue = ((x-xPoint)/(points[i+1].x-
xPoint))*(points[i + 1].y - points[i].y) + points[i].y;
                  break;
            }
      }

      return returnValue;
      }
```

8. MaxFusion

Esta clase hereda de la clase abstracta **Fusion**. Recibe una lista de **Factor** de los cuales seleccionará el que mayor utilidad otorgue, y lo devolverá en *getValue()*.

```
public MaxFusion(List<Factor> factors) : base(factors) { }
public override float getValue()
{
```

```
	return factors.Max(f => f.getValue());
}
```

9. MinFusion

Esta clase hereda de la clase abstracta **Fusion**. Recibe una lista de **Factor** de los cuales seleccionará el que menor utilidad otorgue, y lo devolverá en *getValue()*.

```
public MinFusion(List<Factor> factors) : base(factors) { }
public override float getValue()
{
	return factors.Min(f => f.getValue());
}
```

10. WeightedSumFusion

Esta clase hereda de la clase abstracta **Fusion**. Recibe una lista de **Factor** y, opcionalmente, otra de pesos (deben ser el mismo número que de **Factor**). En caso de no recibir la lista de pesos, en *getValue()* devolverá la suma ponderada de todos los factores. Si lo recibe, aplicará el peso correspondiente a cada **Factor** de la lista.

```
private List<float> weights;
public WeightedSumFusion(List<Factor> factors, List<float> weights) :
base(factors)
{
	this.weights = weights;
}
public override float getValue()
{
	if(weights == null) return factors.Sum(f => f.getValue()) /
factors.Count;

	float sum = 0.0f;
	for(int i = 0; i < factors.Count; i++)
	{
		float factor = factors[i].getValue();
		sum += factor * weights[i];
	}

	return sum;
}
```

11. UtilityAction

Esta clase contiene el factor de utilidad que evaluará el **UtilitySystemEngine** para saber qué **UtilityAction** elegir. También contiene por debajo el estado (y, por tanto, la acción que se va a realizar). Esta clase está basada en **LeafNode**, que también ha sido construida a partir de los estados de una FSM. Para poder contener una máquina es necesario el booleano que le indicará si contiene una submáquina o no junto con la máquina asociada, en caso de tenerla.

En la sección de variables hay dos cosas que quizás resaltan más y crean confusión. La primera, "*Func<ReturnValues> valueReturned*", se trata de una referencia a función necesaria para un caso específico en el que una **UtilityAction** de una **UtilitySystemEngine** submáquina sale a una **LeafNode** de un Behaviour Tree, esperando a que el valor devuelto sea diferente de "Running". La segunda, el **BehaviourTreeEngine**. Ésta variable es necesaria para poder realizar las transiciones de una **UtilityAction** al nodo hoja asociado, del **BehaviourTreeEngine** correspondiente.

```
public State utilityState;
public bool HasSubmachine;
public BehaviourEngine subMachine;
private Factor factor;
private UtilitySystemEngine uCurvesEngine;

private Func<ReturnValues> valueReturned;
private BehaviourTreeEngine bt;
```

Esta clase tiene en total cuatro constructores: el primero, para crear acciones normales de un sistema de utilidad. El segundo, para crear acciones con una submáquina dentro. El tercero, para crear una acción normal pero con la peculiaridad de ser usada específicamente para salir de un **UtilityAction** a un **LeafNode** de un Behaviour Tree, siendo la supermáquina. Éste constructor sale directamente al nodo hoja, es usado únicamente para ello. El cuarto es parecido al anterior, sólo que éste sí tiene una acción a ejecutar y, además, espera a que se devuelva un ReturnValues distinto de "Running" para lanzar la transición al nodo hoja. Esto último es gestionado por el método *Update()* de la propia clase. Para no poner a continuación un código demasiado empalagoso con todos los constructores, únicamente mostraré aquí el constructor número tres (acción de salida directa a **LeafNode**)

```
public UtilityAction(string name, Factor factor, ReturnValues
valueReturned, UtilitySystemEngine utilityCurvesEngine,
BehaviourTreeEngine behaviourTreeEngine)
      {
      this.HasSubmachine = false;

      Action action = () =>
      {
            new Transition("Exit_Action_Transition",
this.utilityState, new PushPerception(this.uCurvesEngine),
this.uCurvesEngine.NodeToReturn,
                              valueReturned, behaviourTreeEngine,
this.uCurvesEngine)
                              .FireTransition();
      };
      this.utilityState = new State(name, action,
utilityCurvesEngine);
      this.factor = factor;
      this.uCurvesEngine = utilityCurvesEngine;
}
```

Los métodos de esta clase son bastante sencillos: el primero, *getUtility()*, devuelve el valor de utilidad del factor asociado a la clase (limitando el valor por debajo y por encima en el propio método). El segundo es el método *Update()*, que se encarga de gestionar la espera en caso de que la acción sea del tipo del cuarto constructor, es decir, que tenga que esperar a que el valor devuelto por la función asociada sea distinto de “Running”.

```
public float getUtility()
{
      float utilityValue = factor.getValue();
      if (utilityValue > 1.0f) return 1.0f;
      if (utilityValue < 0.0f) return 0.0f;
      return utilityValue;
}
public void Update()
{
      if(this.valueReturned != null)
      {
            ReturnValues returnValue = this.valueReturned();
            if (returnValue != ReturnValues.Running)
            {
                  new Transition("Exit_Action_Transition",
this.utilityState, new PushPerception(this.uCurvesEngine),
this.uCurvesEngine.NodeToReturn,returnValue, this.bt,
```

```
this.uCurvesEngine).FireTransition();
            }}}
```

12. UtilitySystemEngine

Esta clase es la principal del sistema. Hereda directamente de **BehaviourEngine**. Contiene la acción activa, la lista de acciones que tiene el sistema y el valor de inercia asignado a la acción activa.

```
public UtilityAction ActiveAction;
public List<UtilityAction> actions;
private float inertia;
```

Esta clase tiene dos constructores: el primero, para iniciar un sistema de utilidad sin la posibilidad de ser una submáquina y el segundo tiene la posibilidad de iniciar un sistema de utilidad siendo submáquina. El código proporcionado pertenece a esta última.

```
public UtilitySystemEngine(bool isSubmachine, float inertia = 1.3f)
{
      base.transitions = new Dictionary<string, Transition>();
      base.states = new Dictionary<string, State>();
      this.actions = new List<UtilityAction>();
      base.IsSubMachine = isSubmachine;

      entryState = new State("Entry_Machine", this);
      this.actualState = entryState;
      states.Add(entryState.Name, entryState);

      this.inertia = inertia;

      Active = (isSubmachine) ? false : true;
}
```

Esta clase tiene tres métodos: *Update()*, *getMaxUtilityIndex()* y *Reset()*. El segundo nombrado se trata de una porción de código que separé de *Update()* para hacer el código más entendible. Como indica su nombre, *getMaxUtilityIndex()* devuelve el índice del elemento que mayor utilidad tenga de la lista de acciones, aplicando el peso proporcionado para evadir el problema de la inercia. El método selecciona la primera coincidencia que mayor valor tenga, si dos tienen el mismo, se queda con el primero.
Update() es el método que tienen todas las máquinas. Éste en particular se encarga de evitar errores (sólo se accede a la funcionalidad del método si la lista de acciones contiene

al menos una acción), si la acción activa no existe elige la de mayor utilidad si la máquina está activa, si la máquina no está activa y la acción activa no tiene submáquina, no se ejecuta el método. En caso de que el estado de la acción activa no se corresponda con el estado actual de la máquina, se realiza un reseteo para evitar errores. Posteriormente, se escoge la acción que mayor utilidad tenga y se realiza la transición en caso de ser una acción diferente a la actual. Además, se ejecuta el método *Update()* de la acción activa. *Reset()* se encarga únicamente de poner **ActiveAction** a null, para que *Update()* seleccione de nuevo otra acción.

```
public void Update()
{
      if(actions.Count != 0)
      {
            if (ActiveAction != null)
            {
                  if (!Active && !ActiveAction.HasSubmachine) return;

                  if (ActiveAction.utilityState != this.actualState)
                  {
                      this.Reset();
                  }

                  int maxIndex = getMaxUtilityIndex();
                  int activeActionIndex =
this.actions.IndexOf(ActiveAction);

                  if (maxIndex != activeActionIndex)
                  {
                      ExitTransition(this.actions[maxIndex]);
                  }
                  ActiveAction.Update();
            } else if(Active && ActiveAction == null) {
                  int maxIndex = getMaxUtilityIndex();
                  ExitTransition(this.actions[maxIndex]);
                  ActiveAction.Update();
            }
      }
}

private int getMaxUtilityIndex()
{
      int actionsSize = this.actions.Count;
      List<float> utilities = new List<float>(actionsSize);
```

```
      for (int i = 0; i < actionsSize; i++)
      {
            if(this.actions[i] == ActiveAction)
            {
                  if (this.actions[i].getUtility() * inertia > 1.0f) {
utilities.Add(1.0f); }
                  else { utilities.Add(this.actions[i].getUtility() *
inertia); }

            } else {
                  utilities.Add(this.actions[i].getUtility());
            }

      }
      return utilities.IndexOf(utilities.Max());
}

public override void Reset()
{
      this.ActiveAction = null;
}
```

Según el tipo de acción, hay que realizar una transición dinámica diferente. En caso de contener una submáquina, se deben resetear las percepciones del estado activo para evitar errores (sobre todo de una FSM), y realizar la supertransición correspondiente. Si no contiene una submáquina, la transición de una acción a otra es simple.

```
public void ExitTransition(UtilityAction action)
{
      if (ActiveAction != null)
      {
            if (ActiveAction.HasSubmachine)
            {

      this.ActiveAction.subMachine.ResetPerceptionsActiveState();
                  new Transition("Max_Utility_Transition",
this.ActiveAction.subMachine.actualState, new PushPerception(this),
                  this.GetEntryState(), this,
this.ActiveAction.subMachine)
                  .FireTransition();
            } else
            {
                  new Transition("Max_Utility_Transition",
this.actualState, new PushPerception(this), action.utilityState, this)
```

```
                  .FireTransition();
            }
      } else
      {
            new Transition("Max_Utility_Transition", this.actualState,
new PushPerception(this), action.utilityState, this)
                  .FireTransition();
      }

      this.ActiveAction = action;
}
```

Los tipos de acciones que se pueden crear son los nombrados en la sección dedicada a **UtilityAction**, sin contar el de la submáquina. El primero crea una acción normal y la almacena en la máquina, el segundo crea una acción preparada para salir de la máquina al nodo hoja contenedor del **UtilitySystemEngine** y el tercero crea una acción que sirve para salir de la máquina al nodo hoja contenedor del **UtilitySystemEngine**, con un método a ejecutar asociado y esperando a que devuelva un valor distinto de "Running". Para evitar aglomeración de código, en la memoria se muestra únicamente el código para crear una acción simple.

```
public UtilityAction CreateUtilityAction(string name, Action action,
Factor factor)
{
      if (!states.ContainsKey(name))
      {
            UtilityAction uAction = new UtilityAction(name, action,
factor, this);
            actions.Add(uAction);
            states.Add(name, uAction.utilityState);

            return uAction;
      }
      else
      {
            throw new DuplicateWaitObjectException(name, "The utility
action already exists in the utility engine");
      }
}
```

El otro tipo de **UtilityAction** que se puede crear es el que contiene submáquinas. El **UtilitySystemEngine** permite crear acciones con submáquina que van al estado de entrada

de la submáquina o a un estado especificado como argumento. El siguiente código muestra aquel que va al estado de entrada de la submáquina.

```
public UtilityAction CreateSubBehaviour(string actionName, Factor
factor, BehaviourEngine subBehaviourEngine)
{
      if (!states.ContainsKey(actionName))
      {
            State stateTo = subBehaviourEngine.GetEntryState();
            State state = new State(actionName,
subBehaviourEngine.GetState("Entry_Machine"), stateTo,
subBehaviourEngine, this);
            UtilityAction utilAction = new UtilityAction(state, factor,
this, subBehaviourEngine);
            states.Add(utilAction.utilityState.Name,
utilAction.utilityState);
            actions.Add(utilAction);

            return utilAction;
      } else
      {
            throw new DuplicateWaitObjectException(actionName, "The
utility action already exists in the utility engine");
      }
}
```