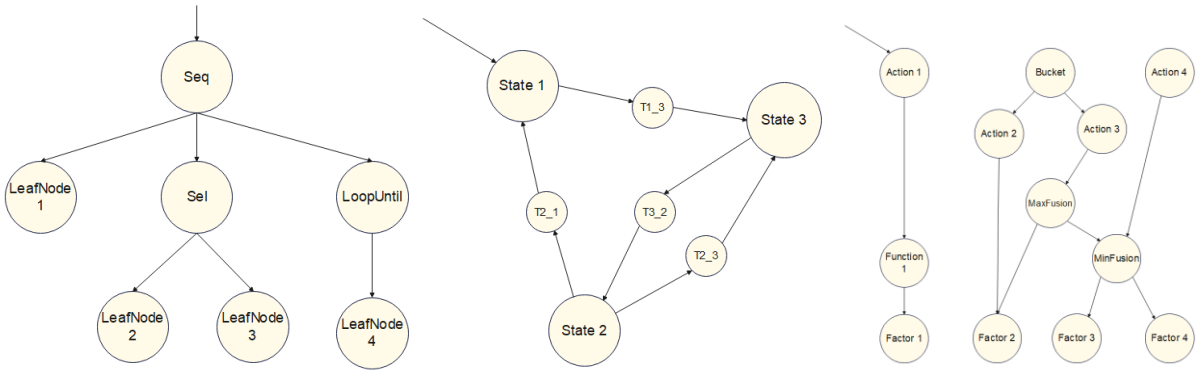
Guía de uso de la API de sistemas de comportamiento

Funcionamiento general

1.1 Como funcionan los sistemas de comportamiento

Los sistemas de comportamiento se crean como **grafos dirigidos** formados por **nodos** conectados entre sí. Cada tipo de grafo tiene métodos específicos para crear y conectar sus nodos. Algunos nodos pueden ejecutar acciones, comprobar percepciones o controlar el flujo de ejecución.

Además, cada tipo de grafo tiene asignado un tipo de nodo concreto y todos los nodos que contengan deben ser de ese tipo, o heredar de ese tipo. A su vez, cada tipo de nodo limita a que nodos pueden conectarse y cuantas conexiones puede tener.



1. Representaciones de los sistemas de comportamiento como grafos (árbol de comportamiento, FSM y sistema de utilidad).

1.2 Ejecución de sistemas de comportamiento

Los grafos y la mayoría de los nodos definen una propiedad *Status* que representa su **estado de ejecución**. Este estado puede tener varios valores:

- *None*: No se está ejecutando.
- *Running*: Se está ejecutando.
- *Success*: La ejecución ha terminado con éxito.
- *Failure*: La ejecución ha terminado con fallo.

También definen métodos de ejecución para modificar su estado interno (Start, Update y Stop). Cada elemento implementa estos métodos de forma distinta, pero existe un comportamiento común:

- *Start*: Inicia la ejecución del grafo/nodo y pone su estado interno en *Running*.
- *Update:* Actualiza el grafo/nodo. Durante la actualización puede cambiar su estado a *Success* o *Failure*.
- *Stop:* Termina la ejecución del grafo/nodo y pone su estado interno en *None*.

1.3 Creación de sistemas de comportamiento

Cada tipo de grafo de comportamiento usa métodos distintos para crearse, pero todos siguen una estructura común.

- Crear una instancia del grafo de comportamiento deseado.
- Crear cada uno de los nodos con su método específico.
- Comenzar la ejecución con el método Start.
- Lanzar el método Update en un bucle de ejecución.
- Ejecutar el método Stop al terminar la ejecución.

1.4 Crear y buscar nodos

Todos los métodos para crear nodos tienen una sobrecarga que permite pasar el nombre del método para guardarlo en un diccionario interno y poder buscarlo después usando ese nombre.

```
var tree = new BehaviourTree();
var leaf = tree.CreateLeafNode("nodo hoja");
…
Node node = tree.FindNode("nodo hoja");

// No lanza excepción si no lo encuentra
Node node = tree.FindNodeOrDefault("nodo hoja");

// Busca el nodo de un tipo específico.
LeafNode node = tree.FindNode<LeafNode>("nodo hoja");
LeafNode node = tree.FindNodeOrDefault<LeafNode>("nodo hoja");
```

Si se intenta crear varios nodos con el mismo nombre se lanzará una excepción.

1.5 Contexto de ejecución

La API incluye la posibilidad de pasar un contexto de ejecución al grafo que se va a ejecutar. El contexto de ejecución es un objeto compartido entre todos los nodos, acciones y percepciones del grafo. Para pasar un contexto al grafo se usa el método `SetExecutionContext`, que propagará la referencia a todos los elementos, incluido subgrafos.

Para crear un contexto de ejecución se crea una clase que herede de *ExecutionContext* y se añaden las variables y métodos necesarios. Para que un elemento use el contexto se sobreescribe el método `SetExecutionContext`.

```
public class CustomContext : ExecutionContext
{
    int count = 0;
}

public class ContextAction : Action
{
    CustomContext context;
    public override void SetExecutionContext(ExecutionContext context)
    {
          this.context = (CustomContext)context;
    }
}

var graph = …
// El método debe usarse después de haber creado todos los nodos del grafo y antes de lanzar Start.
graph.SetContext(new CustomContext());
```

2. Acciones

Algunos tipos de nodos pueden tener asignada una acción. Las acciones contienen tres eventos distintos que se lanzan en los métodos `Start`, `Update` y `Stop` del nodo que la contiene. El método `Update` devuelve un valor `Status` para actualizar el estado interno del nodo que la ejecuta.

2.1 Acciones personalizadas

Para crear una acción personalizada se crea un objeto de la clase `FunctionalAction`, pasando como parámetros un método para cada uno de los eventos.

```
void Start(){...}
Status update(){...}
void Stop(){...}

FunctionalAction action = new FunctionalAction(Start, Update, Stop);
```

El constructor de la clase `FunctionalAction` tiene sobrecargas en las que no hace falta especificar los tres métodos. Si no se especifica el método `Update` devolverá siempre `Running`.

2.2 Acciones con subgrafo

Para crear una acción que ejecute un subgrafo se crea un objeto de la clase `SubsystemAction` y se pasa como parámetro el grafo.

```
BehaviourGraph graph = new BehaviourTree();

SubsystemGraph action = new SubsystemGraph(graph);
```

2.3 Crear tipos de acciones

Es posible crear nuevos tipos de acciones extendiendo la clase `Action` e implementando los métodos `Start` `Update` y `Stop`, aunque solo es necesario el método `Update`.

```
public class CustomAction : Action
{
    public override void Start() { . . . }
    public override Status Update() { . . . }
    public override void Stop() { . . . }
}
```

3. Percepciones

Las percepciones se usan para que los nodos comprueben condiciones. Las percepciones también tienen un evento para cada método del nodo (`Init`, `Check` y `Reset` para los métodos `Start`, `Update` y `Stop` respectivamente). En este caso el método `Check` devuelve un booleano.

3.1 Percepciones personalizadas

Para crear una percepción personalizada se crea un objeto de la clase `ConditionPerception`, pasando como parámetros un método para cada uno de los eventos.

```
void Init(){...}
bool Check(){...}
void Reset(){...}

ConditionPerception perception = new ConditionPerception(Init, Check, Reset)
```

3.2 Percepciones compuestas

Las percepciones compuestas comprueban varias subpercepciones y devuelven el resultado de realizar una operación lógica (AND u OR) sobre los valores booleanos recibidos.

Para crear una percepción compuesta se crea un objeto de la clase deseada y se pasa como parámetro la lista de subpercepciones.

```
List<Perception> listOfPerceptions = new List<Perception>(){perception1, perception2, …};
AndPerception perception = new AndPerception(listOfPerceptions);

// También pueden pasarse las percepciones directamente
AndPerception perception = new AndPerception(perception1, perception2, …);
```

3.3 Percepciones de "Status"

Las percepciones de tipo `ExecutionStatusPerception` comprueban el valor `Status` de un nodo o grafo y devuelven `true` si coincide con el o los valores `Status` especificados.

Para crear una percepción de este tipo se pasa como argumento el elemento que se va a comprobar y el valor o valores Status que activan la percepción.

```
// Para comprobar el estado de un elemento debe implementar la interfaz IStatusHandler.
IStatusHandler handler = new BehaviourTree();

// Comprueba si el árbol de comportamiento aún está en ejecución
StatusPerception p = new StatusPerception(handler, StatusFlags.Running);

// Comprueba si aún está en ejecución o ha terminado con éxito (y aun no se ha parado la ejecución).
StatusPerception p = new StatusPerception(handler, StatusFlags.Running | StatusFlags.Success);

// Puede usarse también para comprobar si una máquina de estados está ejecutando un estado concreto, ya
// que solo un estado puede estár en ejecución al mismo tiempo.
StatusPerception p = new StatusPerception(state, StatusFlags.Actived);
// Actived equivale a Running | Success | Failure.
```

3.4 Percepciones "Timer"

Las percepciones de tipo Timer devuelve false hasta que pase una cantidad de tiempo concreta especificada en el constructor. Puede usarse para lanzar una transición cuando pase el tiempo.

```
  float time = 5f;
  TimerPerception p = new TimerPerception(time);
```

3.5 Crear tipos de percepciones

Es posible crear más tipos de percepciones creando una clase que herede de `Perception` y sobreescribiendo los métodos `Init`, `Check` y `Reset`, aunque solo es imprescindible el método `Check`.

```
public class CustomPerception : Perception
{
    public override void Init() { … }
    public override bool Check() { … }
    public override void Reset() { … }
}
```

4. Percepciones push

Las percepciones push sirven para lanzar transiciones de forma externa al grafo que las contiene, sin tener que comprobar una condición. Puede usarse para notificar a un sistema de comportamiento cuando ocurra un evento como pulsar una tecla.

Para crearla se pasa como argumento uno o varios elementos que deben implementar la interfaz `IPushActivable`. Las transiciones implementan esta interfaz activándose cuando se lanza el método `Fire`.

```
State state1 = fsm.CreateState(…);
State state2 = fsm.CreateState(…);

StateTransition transition = fsm.CreateTransition(state1, state2);

// Crear la percepción push
PushPerception p1 = new PushPerception(transition);

// Activar la percepción push lanzará la transición t1.
p1.Fire();
```

5. Árboles de comportamiento

5.1 Funcionamiento de un árbol de comportamiento

Los árboles de comportamientos se ejecutan desde el nodo raíz hasta uno de los nodos hoja en cada iteración hasta que el estado del nodo raíz sea *Success* o *Failure*.

Los nodos de un árbol de comportamiento son de tres tipos:

- **Nodos hoja**: Ejecutan una acción y su estado se actualiza con el valor que devuelva la acción en su `Update`.
- **Decoradores**: Ejecutan su nodo hijo y su estado se actualiza con el estado del nodo hijo modificado, dependiendo del tipo de decorador.
- **Composite**: Cada iteración ejecuta uno de sus hijos. Cuando la ejecución de un hijo acaba (devuelve *Success* o *Failure*) el nodo compuesto pasa a ejecutar el siguiente hijo, hasta que el valor devuelto sea *Failure* si es un `SequencerNode` o *Success* si es un `SelectorNode`, o no queden más hijos.

5.2 Crear un árbol de comportamiento

El primer paso es crear una instancia de árbol de comportamiento:

```
BehaviourTree bt = new BehaviourTree();
```

Para crear un nodo hoja se usa el método `CreateLeafNode` y se pasa como parámetro la acción que realizará el nodo.

```
  Action action = …
  LeafNode leafNode = bt.CreateLeafNode(action);
```

Para crear un nodo decorador se usa el método genérico `CreateDecorator`, pasando como parámetro genérico el tipo de decorador y como argumento el nodo hijo. Las variables propias de cada tipo de decorador (iteraciones en `IteratorNode`, percepción en `ConditionNode`, etc.) se inicializan después usando métodos Setter o las variables directamente.

```
LoopUntilNode loop = bt.CreateDecorator<LoopUntilNode>(childNode)
    .SetTargetStatus(Status.Success);
loop.TargetStatus = Status.Success;
```

Para crear un nodo compuesto se usa el método genérico `CreateComposite`, pasando como parámetro genérico el tipo de nodo compuesto y como argumentos la lista de nodos hijo y un booleano que indica si el orden de ejecución de los hijos es aleatorio o no. También es posible pasar los nodos hijos directamente como parámetros.

```
List<BTNode> listOfChilds = new List<BTNode>(btNode1, btNode2, …);
SequencerNode sequencer = bt.CreateComposite<SequencerNode>(listOfChilds, false);

SequencerNode sequencer = bt.CreateComposite<SequencerNode>(false, btNode1,btNode2, …);
```

Por último, se debe especificar cual es el nodo raíz del árbol. Si no se especifica será el primero en ser creado.

```
bt.setStartNode(sequencer);
```

5.3 Crear nuevos tipos de nodos decoradores

Se pueden crear nuevos tipos de decoradores extendiendo la clase `DirectDecoratorNode` e implementando el método `ModifyStatus`. El método recibe el valor Status del nodo hijo y lo devuelve modificado.

Ejemplo: Un decorador que cambia el valor Success por Failure.

```
public class Failer : DirectDecorator
{
    protected override Status ModifyStatus(Status childStatus)
    {
        if(childStatus == Status.Success) return Status.Failure;
        else return childStatus;
    }
}
```

5.4 Crear nuevos tipos de nodos compuestos

Se puede crear un nuevo tipo de nodo compuesto heredando de `SerialCompositeNode` e implementando los métodos `KeepExecutingStatus` y `GetFinalStatus`.

El primero recibe el valor Status final del nodo hijo que se está ejecutando y devuelve un booleano que indica si la ejecución tiene que continuar al siguiente hijo o no. El segundo permite modificar el valor Status final.

Ejemplo: Nodo compuesto que ejecuta sus hijos hasta que uno devuelve Failure y devuelve el estado del último hijo ejecutado invertido.

```
public class InverterSequence : SerialCompositeNode
{
    protected override bool KeepExecuting(Status status)
    {
        return Status == Status.Success;
    }

    protected override Status GetFinalStatus(Status status)
    {
        if(childStatus == Status.Success) return Status.Failure;
        else if(childStatus == Status.Failure) return Status.Success;
        else return childStatus;
    }
}
```

6. Máquinas de estados

6.1 Funcionamiento de una máquina de estados

Las máquinas de estados o FSM están formadas por estados y transiciones. Cada interacción de la FSM se ejecuta el estado actual y se comprueban sus transiciones. Las transiciones pueden ser entre dos estados o transiciones de salida.

6.2 Crear una máquina de estados

Para crear una nueva máquina de estados:

```
FSM fsm = new FSM();
```

Para crear estados se usa el método `CreateState` y se pasa como argumento la acción que ejecuta el estado. Si no se especifica ninguna acción, el estado permanecerá en "*Running*" hasta que se active alguna transición.

```
Action action = new FunctionalAction(…);
State state = fsm.CreateState(action);
```

Las transiciones entre estados se crean usando el método `CreateTransition` y se pasa como argumentos el estado origen y el estado destino.

```
StateTransition transition = fsm.CreateTransition(state1, state2);
```

Para crear transiciones de salida se usa el método `CreateExitTransition` y se pasa como argumento el estado inicial y valor Status de salida.

```
ExitTransition exit = fsm.CreateExitTransition(state1, Status.Success);
```

Todos los métodos que crean transiciones tienen varios parámetros opcionales:

- *Perception*: Se puede especificar una percepción a la transición para que solo se active si la percepción devuelva true. Si no se especifica, por defecto la transición se activará siempre que se compruebe (transición lambda).
- *Action*: La transición puede ejecutar una acción cuando se active. Como la activación de las percepciones solo dura un ciclo de ejecución (se ejecutan los métodos Start, Update y Stop de la acción en el mismo ciclo) es recomendable que la acción de las transiciones sea de tipo `FunctionalAction`.
- *StatusFlags*: Es posible especificar cuando una transición es comprobada por su estado origen en base si su valor *Status* coincide con el valor de las "flags" de la transición.
    - Por defecto el valor es *Actived* y la transición se comprobará siempre.
    - Si queremos que la transición no se compruebe nunca, el valor debe ser *None*. Es útil si queremos que la transición solo sea activable a través de *PushTransitions*.
    - Si queremos que la transición se compruebe cuando la acción del estado haya terminado, el valor puede ser Success, Failure o Finished, según si se debe comprobar sólo si ha terminado con éxito, con fallo o con ambas.

```
// Transición que comprueba una percepción y ejecuta una acción
Action action = new FunctionalAction(…);
Perception perception = …;
StateTransition t1 = fsm.CreateTransition(st1, st2, perception, action);

// Transición que se activa cuando el estado inicial termina su ejecución
StateTransition t1 = fsm.CreateTransition(st1, st2, statusFlags: StatusFlags.Finished);

// Transición que no se comprueba nunca
StateTransition t1 = fsm.CreateTransition(st1, st2, statusFlags: StatusFlags None);
```

También es posible crear un estado con probabilidades y especificar la probabilidad de cada transición.

Las transiciones que no tengan probabilidad asignada se comprobarán de forma normal.

```
ProbabilisticState probState = fsm.CreateProbabilisticState(action);
StateTransition st = fsm.CreateTransition(probState, otherState);
probState.SetProbability(st, 0.5f);
```

Para especificar el estado inicial de la máquina de estados (por defecto el primero en crearse), se usa el siguiente método:

```
  fsm.SetEntryState(state1);
```

6.3 Máquina de estados de pila

Las máquinas de estados de pila pueden guardar en una pila el ultimo estado activo y volver a el desde cualquier otro estado.

Para crear una nueva máquina de estados de pila:

```
StackFSM stackfsm = new StackFSM();
```

Los métodos para crear nodos son los mismos, pero añade métodos para crear transiciones específicas:

Para crear una transición push, se usa el método *CreatePushTransition* y se pasan como argumentos el estado origen y destino, y los parámetros opcionales.

```
PushTransition push = StackFSM.CreatePushTransition(st1, st2, …);
```

Para crear una transición pop, se usa el método "CreatePopTransition" y se pasan como argumentos el estado origen y los parámetros opcionales.

```
PopTransition pop = StackFSM.CreatePopTransition(st1,  …);
```

7. Sistemas de utilidad

7.1 Funcionamiento de un sistema de utilidad

Los sistemas de utilidad eligen un elemento de un conjunto en cada iteración en base a un valor de utilidad. Estos elementos obtienen su valor de utilidad a partir de nodos de tipo Factor.

Los sistemas de utilidad tienen dos tipos de nodos:

- *UtilitySelectableNode*: Son nodos que puede seleccionar el sistema de utilidad para ejecutarlo. Pueden ser *UtilityActions*, que ejecutan una acción y tienen asociadas un *Factor* como nodo hijo, *UtilityBuckets*, que funcionan como grupos de acciones con prioridad o UtilityExitNode que terminan la ejecución de la máquina de estados.
- *Factor*: Son nodos que sirven para definir la utilidad de los elementos del sistema de utilidad. Pueden ser factores variable, factores función o factores fusión.

7.2 Crear un sistema de Utilidad

Para crear un factor variable se usa el método `CreateVariableFactor` y se pasa como argumentos una función que devuelva un valor float, y dos float con el valor mínimo y máximo que puede devolver para normalizarlo entre 0 y 1.

```
float v = 0f;
VariableFactor v = us.CreateVariableFactor("variable", () => v, 0f, 1f);
```

Para crear factores función se usa el método genérico `CreateFunctionFactor` pasando como parámetro genérico el tipo de factor función y como argumento el factor hijo.

```
// Factor con una funcion linear de pendiente -½ y ordenada en el origen 1.0.
LinearFunction lf = us.CreateFunctionFactor<LinearFunction>(factor).SetSlope(-.5f).SetYIntercept(1f);
// Factor con función que eleva al cuadrado la utilidad del factor hijo
CustomFunction cf = us.CreateFunctionFactor<CustomFunction>(factor).SetFunction(x => x * x);
// Factor que construye la función con puntos
PointedFunction pf = us.CreateFunctionFactor<PointedFunction>(factor)
   .SetPoints(new List<Vector2>(new Vector2(0,0.4), new Vector2(0.5f,0.8f), new Vector2(1, 0.6f))
```

Para crear factores fusión se usa el método `CreateFusionFactor` pasando como parámetro genérico el tipo de factor fusión y como argumento la lista de factores hijos.

```
List<Factor> childFactors = new List<Factor>(f1, f2, f3);

// La utilidad del factor será el mínimo de f1, f2 y f3
MinFusionFactor = us.CreateFusionFactor< MinFusionFactor>(listOfFactors)
// La utilidad del factor será el máximo de f1, f2 y f3
MaxFusionFactor = us.CreateFusionFactor< MaxFusionFactor>(listOfFactors)

// La utilidad del factor será 0.1 * f1 + 0.3 * f2 + 0.6 * f3
WeightedFusionFactor = us.CreateFusionFactor<WeightedFusionFactor>(listOfFactors)
    .SetWeights(new float[]{0.1f, 0.3f, 0.6f});
```

Para crear acciones de utilidad se usa el método *CreateAction* pasando como argumentos el factor para calcular la utilidad y la acción que realiza. De forma opcional se puede pasar el valor booleano al parámetro *finishOnComplete* para que el sistema de utilidad termine su ejecución después de terminar de ejecutar la acción (por defecto es *false*).

```
  Action a = new FunctionalAction(…);
  UtilityAction action = us.CreateAction(factor, a, true);
```

También es posible crear nodos que salgan directamente del sistema sin ejecutar ninguna acción, usando el método *CreateExitNode* y pasando como parámetros el factor para calcular la utilidad y el valor Status de salida.

```
  UtilityExitNode action = us.CreateExitNode(factor, Status.Success);
```

Para crear grupos de acciones de utilidad se usa el método *CreateBucket* pasando como argumentos el valor de inercia y el umbral del grupo.

```
  UtilityBucket bucket = us.CreateBucket(1.3f, 0.3f);
```

Para incluir una acción o un elemento de salida en un bucket se añade en su constructor. También es posible crear grupos dentro de otros

```
UtilityAction groupAction = us.CreateAction(factor, a, true, bucket);
UtilityExitNode action = us.CreateExitNode(factor, Status.Success, bucket);
UtilityBucket subBucket = us.CreateBucket(1.3f, 0.3f, bucket);
```

7.3 Crear nuevos tipos de factor función

Para crear un nuevo tipo de factor función se crea una clase que herede de FunctionFactor y se implementa el método Evaluate. El método recibe la utilidad del nodo hijo y la devuelve modificada.

Ejemplo: Función que devuelve la utilidad del nodo hijo elevada al cuadrado.

```
public class SquareFactor : FunctionFactor
{
    protected override float Evaluate(float childUtility)
    {
        return childUtility * childUtility;
    }
}
```

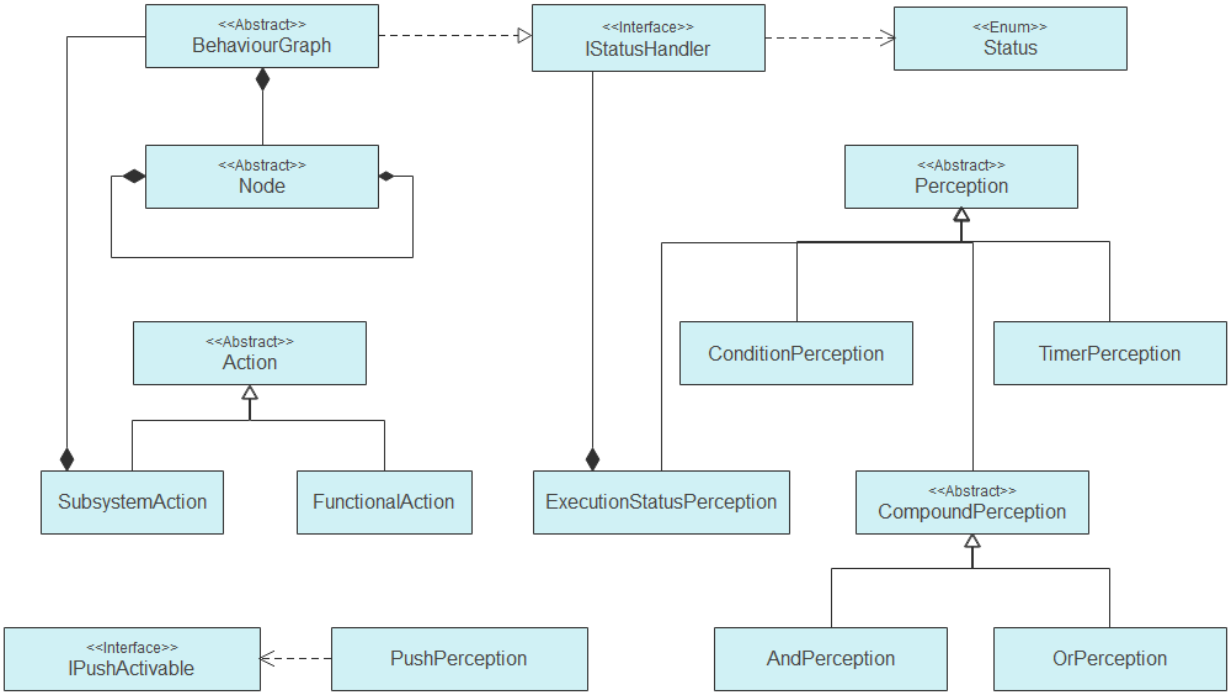
7.4 Crear nuevos tipos de factor fusión

Para crear un nuevo tipo de factor función se crea una clase que herede de FusionFactor y se implementa el método Evaluate. El método recibe una lista con las utilidades de los nodos hijos y devuelve su valor de utilidad calculado.

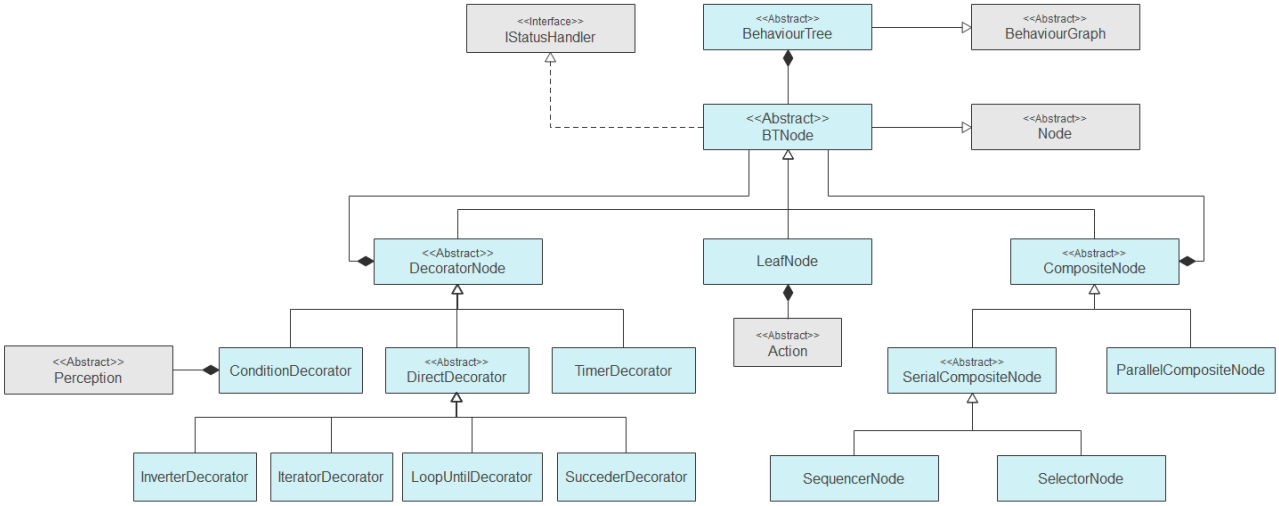
Ejemplo: Factor fusión que devuelve la media de la utilidad de los nodos hijos.

```
Using System.Linq;
public class SquareFactor : FunctionFactor
{
    protected override float Evaluate(List<float> childUtilities)
    {
        return childUtilities.Average();
    }
}
```

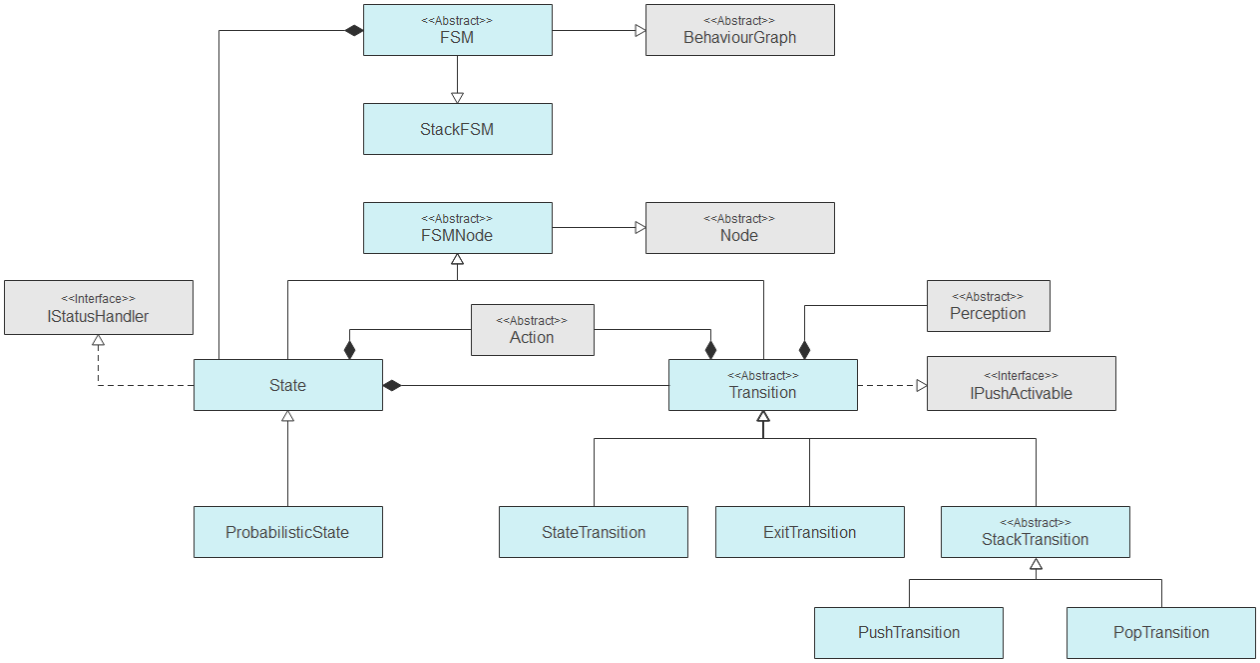
8. Diagramas de clases



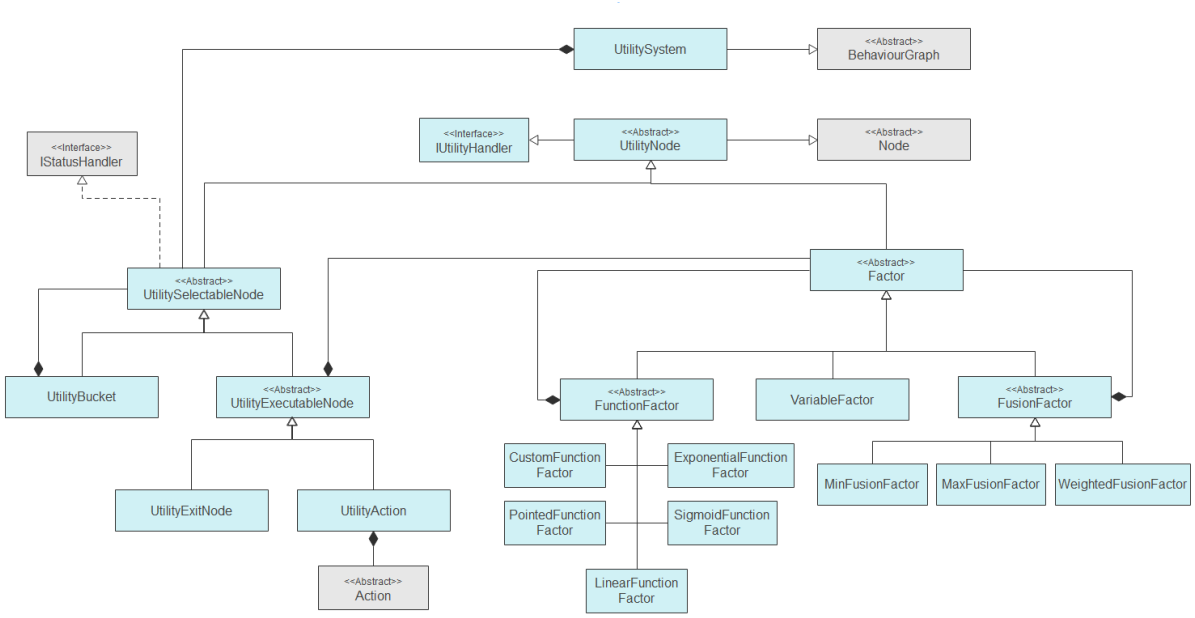
2 Diagrama de clases principal.



3 Diagrama de clases de árboles de comportamiento.



4 Diagrama de clases de máquinas de estados.



5 Diagrama de clases de Sistemas de utilidad.