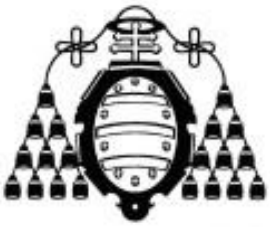
UNIVERSIDAD DE OVIEDO



MÁSTER EN INGENIERÍA WEB

TRABAJO FIN DE MÁSTER

"DISEÑO DE UN LENGUAJE DE DOMINIO ESPECÍFICO PARA EL MODELADO DE LA INTELGENCIA ARTIFICIAL EN VIDEOJUEGOS"

DIRECTOR: VICENTE GARCÍA DÍAZ

CO-DIRECTOR: JORDÁN PASCUAL ESPADA

AUTOR: ISMAEL POSADA TROBO

Vº Bº del Director del Proyecto

Agradecimientos

Agradezco el apoyo que mi familia y novia me brindaron por su comprensión y apoyo en los momentos difíciles, a los directores de proyecto por la ayuda que me brindaron tratando siempre de hacerme las cosas más llevaderas y a todos aquellos que tuvisteis un minuto para dedicarme un ánimo y aportasteis vuestro granito de arena de alguna manera en la realización de este proyecto.

Pero sin duda quiero dedicártelo a ti, Welito. Allá donde estés, estoy seguro que te hará mucha ilusión. Va por ti.

Resumen

Siguiendo con la tónica dominante durante estos últimos años en lo que respecta a generación de aplicaciones a partir de lenguajes específicos de dominio, en este caso, siendo el dominio los videojuegos, el principal objetivo que afrontamos con la realización de esta propuesta va encaminado a los comportamientos que pueden experimentar dentro de los videojuegos los personajes no jugables, NPC por sus siglas en inglés (Non Playable Character).

En su día, se creó el proyecto TGame [1], editor que permite a los usuarios la creación de videojuegos de plataformas simples, haciendo uso de un DSL o lenguaje específico de dominio permitiendo a los usuarios con pocos o nulos conocimientos de programación realizar sus propias creaciones.

TGame permitía la definición conjunta del videojuego, sin ahondar en las diferentes partes que pueden conformar un videojuego, como pueden ser IA de los enemigos, entre otros.

Bebiendo de la misma fuente con la que se realizó en su día el proyecto TGame, en este caso vamos un paso más allá y vamos a proveer al usuario de las herramientas necesarias para que, sin tener unos conocimientos extensos o incluso bajas nociones informáticas, sea capaz de, además de poder crear un videojuego, proveer de un comportamiento personalizado a los NPC´s.

Esto se consigue mediante la mezcla de un lenguaje específico de dominio que, siguiendo una serie de reglas ya predefinidas, permita a los usuarios crear este tipo de comportamientos de una forma fácil, intuitiva y sencilla.

T2Game (llamaremos así a nuestra propuesta a partir de ahora) está destinado a aquellas personas que si bien sienten devoción por el mundo de los videojuegos, quieren ir un paso más allá en la generación de videojuegos sin poseer ese grado de conocimiento que inconscientemente siempre es requerido por otras plataformas y/o herramientas.

En este documento se realiza una propuesta sobre una posible solución a la construcción de patrones de comportamiento e interacción complejos. Los objetivos principales son los siguientes:

- Proponer un diseño que permita a los usuarios sin conocimientos de programación definir de forma simple patrones de comportamiento e interacción complejos en enemigos de videojuegos de plataformas. La solución ha de ser lo suficientemente potente para permitir variaciones en cuanto a comportamientos, pudiendo elegir en cada momento qué hacer según qué acción.
- Favorecer la definición siguiendo el patrón "*Keep it Simple*" comparándolo con otros editores o frameworks presentes en el mercado, así como otras soluciones investigadas recientemente.

En términos de producción, pretendemos que una de los puntos fuertes de nuestra solución sea la rapidez y la sencillez con la que vamos a poder crear los comportamientos de los NPC´s,

pudiendo llegar en un futuro a ser una solución adoptada por las herramientas más comunes de desarrollo de videojuegos.

Palabras Clave

DSL, MDE, VIDEOJUEGOS, PATRONES DE COMPORTAMIENTO, MOTORES DE TRANSFORMACIÓN, PLANTILLAS JET

Abstract

Application generators from domain specific languages, in this case, computer games domain, has prevailed during the last few years. On this regard, the main target we follow making this Thesis is to focus our effort over the behavior of a NPC (Non playable character) could take within computer games.

A time ago, TGame Project, the editor, was developed in order to allow users to create platform computer games, using a DSL or specific domain language, allowing them to create their ideas with a programming misknowledge.

TGame allowed us to define the whole process of a videogame, without going into complex parts of the game, as for instance, IA of enemies.

With our Thesis we are going through a step further and we will provide users the necessary tools so that, with a non-extensive or even low notions of programming knowledge, users will be able to provide a customized behavior to the NPCs, in addition of creating a computer game from the beginning.

This will be obtained through a mix of the domain specific language which follow predefined rules, it will allow users to create this kind of customized behaviors in a easy, intuitive and straightforward way of doing it.

T2Game is designed to those people who feel devotion about computer games, going a step further over the generation of games with that lack of knowledge that is always required by external third-party tools.

In this Thesis, we will make a proposal about a possible solution of how to make and deploy complex behavior patterns. The main goals are the following ones:

- Propose a good design allowing users with a lack of programming knowledge to define behavior patterns and interaction complex patterns of an enemy in a simple way. Solution must be the most powerful to allow different behavior variations, choosing at every moment what to do.
- Stimulate the pattern definition always keeping in mind “Keep it Simple” pattern, comparing to any other alternatives out in the market, as well as any other researchers in this field.

In terms of production, our goal is that speed and simplicity of our solution will be one of the strongest point, allowing the creation of the NPC´s behaviors with the necessary efforts in order to convert our solution in a future solution, adopting it as a part of the behavior development.

Keywords

DSL, MDE, COMPUTER GAMES, BEHAVIOR PATTERNS, GENERATORS, JET TEMPLATES

Índice General

Índice de Figuras

Capítulo 1. Introducción

1.1 Motivación

Prácticamente desde el inicio de la programación de videojuegos, las herramientas provistas para la generación de éstos contenían un componente altamente ligado a la programación que escindía en cierta medida el acceso o la elaboración de este tipo de software por parte de la sociedad.

De esta manera, sólo un grupo seleccionado de personas eran capaces de desarrollar videojuegos, pues tenían un alto grado de conocimiento acerca de la materia, en especial de programación, que aumentaba de forma exponencial la complejidad de estos sistemas. A su vez también, estaban muy limitados pues los costes de desarrollo eran prácticamente inalcanzables a muchos.

Con el paso de los años los videojuegos se convirtieron en negocio de masas, pasando a formar parte de empresas desarrolladoras, personas altamente cualificadas y de diversos campos (físicos, matemáticos, programadores, diseñadores, y un largo etc.).

Es pues a lo largo de estos últimos años cuando se produjo la floración de diversas plataformas de videojuegos que no hicieron más que explotar de una manera exponencial el desarrollo de videojuegos, incrementando en tanto el número de personas que se dedican a su desarrollo.

Para entonces, y hace concretamente 7 años, se creó la herramienta TGame [1], elaborada por Ismael Posada Trobo. TGame es un entorno de creación de videojuegos plataformas simples 2D a partir de un DSL[1] o lenguaje específico de dominio.

En este primer acercamiento, nuestro objetivo fue proveer a usuarios con escasos o nulos conocimientos de programación de las herramientas necesarias para la creación de videojuegos de plataformas. De esta manera, nuestra herramienta permitía a estas personas el desarrollo de videojuegos de un modo fácil y sencillo a diferencia de otras que existían por aquel entonces, que dificultaban ciertas partes del "*workflow*" o flujo de trabajo en general.

A la par que la aplicación que por entonces habíamos creado, multitud de nuevas y no tan nuevas plataformas siguieron apareciendo para que la creación de videojuegos fuera algo al alcance de cualquiera, claramente siempre dentro de unas limitaciones. ¿En qué sentido hay limitaciones?

Los videojuegos en sí presentan muchos aspectos configurables por no decir todos. Son multitud de partes y/o variantes tanto técnicas como visuales que a la hora de crear un videojuego se han de tener en cuenta.

Por ello, multitud de editores presentes hoy en día contienen herramientas para la creación de videojuegos, pudiendo "jugar" con estas características. Es fácil crear videojuegos rápidamente en la mayoría de ellos, pero en cuanto pretendemos aumentar la complejidad de éstos, en

[1] DSL, del inglés, "*Domain Specific Language*" o lenguaje específico de dominio dedicado a resolver un problema en particular.

términos de funcionalidades o de características, las cosas empiezan a complicarse y el nivel de conocimiento en diversas áreas empieza a ser cuanto menos necesario.

Muchas de estas funcionalidades y características son soportadas por los editores y frameworks en forma de asistentes a la hora de ser editadas. Podemos encontrarnos con creación de mapas, definición de actores (carga de texturas, situación en el mapa,…) etc. Otros simplemente dejan ciertas características a la parte programática. Pero, sin duda, realmente no ofrecen una clara asistencia a ciertos aspectos más complejos que pueden marcar el devenir de un videojuego en términos de interacción y calidad.

Estos asistentes en su gran medida permiten a los usuarios con pocos o nulos conocimientos de programación crear videojuegos, siempre dentro de unas limitaciones. Es difícil crear una abstracción entendible para el usuario con aspectos complejos de un videojuego. De ahí viene la razón por la que muchos editores no contemplan la edición o modificación de éstos aspectos dentro de un videojuego. Muchos de ellos, a pesar de lograr hacer una buena abstracción representativa para el usuario, no consiguen ser entendidas en la mayoría de los casos por usuarios de a pie, sino más bien por usuarios ya un poco más especializados.

Uno de los problemas que se observan mayoritariamente en la creación de estos videojuegos a partir de editores con asistente es el poder definir los comportamientos que experimentan los recursos dentro de un videojuego. Estos comportamientos podemos considerarlos como las acciones que se desencadenan en cuanto se produce un evento.

Nuestra propuesta va encaminada a proveer a los usuarios una solución que de una forma abstracta y entendible permita la creación de patrones de comportamiento e interacción complejos.

Es por ello que nació T2Game, siguiendo la misma estela que en su día se proyectó con TGame. En este caso, T2Game va más destinado a la confección y/o elaboración de esos comportamientos (ligado directamente a los personajes no jugadores, NPC[2]).

El usuario va a poder crear y editar esos patrones de comportamiento haciéndolos más "humano" sin esa base programática que tanto es exigida por las restantes plataformas de desarrollo de videojuegos.

Juntando las características principales del proyecto TGame, con las de su segunda versión, T2Game, vamos a poder liberar al usuario de esas nociones de programación, dando más libertad a la parte creativa.

Es por hoy que, T2Game, abre las puertas a un nuevo mundo de creación de patrones de comportamiento en vistas a la creación fácil de videojuegos, pudiendo aumentar la complejidad e interacción de los mismos en cuanto al número de acciones se refiere, permitiendo usar la herramienta como un generador de inteligencia para enemigos que reduce considerablemente la complejidad de implementación.

[2] Un personaje no jugador, o NPC, es aquel que es controlado generalmente por el programa, y no controlado por un humano. De ahí que requiera unos comportamientos predefinidos, haciéndonos creer que tiene vida propia. En otras palabras, que parezca que está manipulado por alguien.

Diferentes alternativas que permiten la creación de patrones de comportamiento están presentes en el mercado, así como diversos proyectos de investigación que focalizan sus esfuerzos en la creación de estos comportamientos.

En puntos posteriores veremos las posibilidades que nos brindan estas herramientas, tales como *Game Maker* y su forma "pseudo-programática" de codificar comportamientos. *Blueprints* y su sistema innovador "*Event-Graph*", uso de diagramas para la definición de acciones a partir de eventos, así como las posibilidades que nos brinda *Unity*, un framework de edición de videojuegos que nos brinda las herramientas necesarias para poder construir cualquier tipo de comportamiento asociado a un enemigo.

Así mismo, se comentarán los lenguajes de programación más utilizados por estas herramientas, por lo que haremos una mención especial al índice TIOBE [2], índice que indica la popularidad de un lenguaje de programación.

Capítulo 2. Fijación de Objetivos

2.1 Objetivos

Los principales objetivos de la investigación son las siguientes:

- Sentar las bases para un diseño que permita a los usuarios sin conocimientos de programación definir de forma simple patrones de comportamiento e interacción complejos en enemigos de videojuegos de plataformas. La solución ha de ser lo suficientemente potente para permitir variaciones en cuanto a comportamientos, pudiendo elegir en cada momento qué hacer según qué acción.
    - Esta solución incluirá además el diseño de un lenguaje específico para el modelado de los patrones de comportamiento, que ha de ser ampliable a desarrollos futuros, pudiendo ser alimentado con más opciones por diversos desarrolladores siguiendo la misma tónica constructiva.

Construcción de un prototipo real y funcional que genere mejoras sustanciales en el proceso de creación de videojuegos de plataformas.

Capítulo 3. Estado del arte

A continuación se expone una visión global del estado del arte, que abarca desde el origen del desarrollo de videojuegos con componente de Inteligencia Artificial hasta nuestros días.

3.1 Desarrollo de videojuegos y comportamientos de enemigos

El desarrollo de videojuegos tiene su origen cuando tras la II Guerra Mundial, las principales potencias construyeron sus primeras supercomputadoras programables apareciendo por entonces las primeras aplicaciones destinadas al entretenimiento.

Dada la escasa potencia de las computadoras en comparación con las actuales, los desarrollos eran llevados a cabo por un número muy reducido de personas, siendo su principal objetivo usar estas creaciones como demostraciones.

Estos desarrollos eran simples en comparación con los actuales, pero no dejaron de crecer gracias sobre todo a la creatividad de los desarrolladores. Iban ganando en complejidad así como en jugabilidad para suerte o desgracia de los consumidores.

3.1.1 Precursores

Parte de culpa de este incremento de la complejidad de los videojuegos la tiene Alan Turing, considerado el padre de la inteligencia artificial. En 1948, Turing había diseñado un videojuego de ajedrez que no pudo ser probado por la baja potencia de los procesadores de aquella época. Fue cuando en 1952 se puso a prueba su obra simulando los movimientos de una computadora, surgiendo así el concepto de inteligencia artificial [3].

Fueron numerosas obras las que, tras la desaparición de Turing, fueron apareciendo incorporando un componente de inteligencia artificial que se iría mejorando conforme avanzaban los años hasta llegar a nuestros días.

En los últimos años, la inteligencia artificial ha influenciado extensamente en gran medida el desarrollo de videojuegos [4] [5] [6]. Ha ido desarrollándose focalizándose en diferentes categorías de videojuegos, entre las que se incluyen taxonomías tales como plataformas, first-person-shooter (FPS), juegos de rol (RPGs), etc. [7], suponiendo uno de sus mayores retos para la creación de comportamientos en NPCs [8].

Estas técnicas de inteligencia se clasificaron en dos grandes grupos: Determinista y No deterministas [9]. Deterministas son aquellas que definen los comportamientos del NPC y normalmente son predecibles. No Deterministas son aquellas que no son predecibles, facilitando el aprendizaje de nuevos comportamientos emergentes.

Nosotros nos centraremos en la inteligencia artificial determinista, comportamientos que a priori están predefinidos y pueden ser predecibles. Esta predictibilidad se consigue a través de

técnicas tradicionales usadas comúnmente por los desarrolladores de inteligencia artificial de videojuegos, como pueden ser:

- "*Cheating*": damos al NPC más información que la que damos al usuario jugador.
- Autómatas de estados finitos: permitimos al NPC que cambie su comportamiento en base a los eventos que se produzcan en el videojuego.
- Algoritmos "*Path Finding*" o búsqueda de ruta más corta: proveen al NPC de "conocimientos" hacia dónde tiene que ir evitando obstáculos y llegando a los objetivos.
- Comportamientos a base de scripts: los NPC se comportan de una manera específica bajo circunstancias específicas.
- Animaciones expresivas: dan al NPC la ilusión de las emociones.
- Comportamientos en grupo: proveen comportamientos en masa, en manadas, etc.

En la confección de los patrones de comportamiento que aporta nuestra propuesta, haremos uso de las técnicas de inteligencia artificial deterministas a través de "Cheating" y Comportamientos a base de scripts (Scripting-Behavior).

Es pues en la confección de patrones de comportamiento donde queremos eximir al usuario de tener esos conocimientos de programación necesarios para su creación y/o manipulación. Para ello, vamos a hacer uso de la metodología MDE a través de un editor de comportamientos.

3.1.2 MDE

Con el boom de los videojuegos que se viene palpando desde la década de los 90, ha sido masiva la proliferación de herramientas creadoras de videojuegos. Prácticamente todas seguían la misma filosofía de un desarrollo rápido de videojuegos sin profundizar en ninguna característica en particular.

Éstas proveen al usuario de un conjunto de herramientas predefinidas que combinadas pueden dar resultados más que interesantes y dan la posibilidad a cualquier usuario que dispusiera de un ordenador de desarrollar un videojuego.

Si bien en la época de los 90 los desarrollos aún seguían siendo autónomos con algunas compañías afincadas líderes de mercado, no es hasta bien entrado el siglo XXI cuando estas herramientas creadoras de videojuegos entran en auge. Los desarrollos eran íntegros desarrollados por una misma persona la cual debía poseer conocimientos avanzados de programación, de física, de matemáticas, entre otros.

Con el auge de la industria, se quiso llegar a un público más llano al que se le permitiera crear videojuegos evitando poseer tanto conocimiento de las diferentes áreas que puede abarcar un desarrollo de este tipo de aplicaciones. Para entonces, el tiempo consumido para el desarrollo de este tipo de aplicaciones no había experimentado grandes cambios [10], siendo extremadamente alto.

Después de los significantes avances en materia de lenguajes de programación y de entornos de desarrollo integrados (IDEs[3]), desarrollar este tipo de sistemas complejos usando las herramientas actuales requería un esfuerzo hercúleo [11]. Es entonces cuando entra en escena la metodología MDE (Model Driven-Engineering), ingeniería dirigida por modelos, debido principalmente al auge de la complejidad del software.

MDE es una metodología de desarrollo de software centrada en la creación y explotación de modelos de dominio. Sugiere el uso de modelos como el eje principal del ciclo de vida de un software, acortando los tiempos y el esfuerzo en el proceso de desarrollo del software en aras de aumentar la productividad [12]. Estos modelos de dominio son abstracciones, con el fin de representar el vocabulario y conceptos claves del dominio del problema, que están focalizados en resolver un problema específico limitando su alcance. Normalmente, estos modelos vienen expresados a través de DSL´s (Domain Specific Languages) [13] y son desarrollados a partir de un metamodelo.

El metamodelo es un modelo conceptual en el que se representan todos los temas relacionados con un problema específico. Es el encargado de definir el dominio sobre el que se va a actuar, definiendo los elementos de un lenguaje de modelado [14].

3.1.2.1 DSL

DSL o "*Domain Specific Language*", del inglés lenguajes específicos de dominio, son lenguajes que están específicamente diseñados para unas necesidades expresivas dentro de los intereses de una aplicación [13]. Según [15], es un lenguaje de programación dedicado a un problema de dominio en particular, o una técnica de representación o resolución de problemas específica.

En los DSL se crean específicamente para resolver problemas de un determinado dominio, con lo cual no están pensados para resolver problemas fuera de este dominio. Componen el modelo, el cual está desarrollado a partir de un metamodelo. Son lenguajes con objetivos muy específicos, tanto en diseño como en implementación. Pueden ser de diagramación visual como textual, siendo por lo general "pequeños", ofreciendo un conjunto limitado de notaciones y abstracciones. Son por lo general lenguajes declarativos, por lo que pueden verse como lenguajes de especificación, así como lenguaje de programación [16] [1].

3.1.2.2 Modelo

Un modelo es una abstracción teórica del mundo real cuya tarea fundamental es reducir la complejidad, permitiéndonos ver las características importantes que están detrás de un proceso, ignorando detalles de menor importancia [17].

Están pensados principalmente para incrementar la productividad y la compatibilidad entre sistemas, simplificando la etapa de diseño para engrandecer la comunicación entre equipos de trabajo [1].

[3] IDE, "*Integrated Development Environment*", del inglés, entorno de desarrollo integrado, es una aplicación software que proporciona servicios integrales para facilitar las labores del usuario desarrollador.

Hay varias formas de definición de modelos, como son programáticamente, con plugins de EMF basados en el metamodelo y mediante DSLs. Para nuestra propuesta, utilizaremos aquellos definidos a partir de plugins de EMF ("*Eclipse Modeling Framework*") [18].

Los plugins de EMF, entre otras muchas cosas, presentan una característica que permite "traducir" el modelo a un lenguaje de especificación XMI [19].

XMI es un framework de integración basado en XML [20] para el intercambio de modelos, que nació para proveer de una manera estándar la forma en la que las herramientas UML [21] se intercambiaban modelos. Define unas reglas para la generación de Schemas XML[4] a partir de un metamodelo.

Para nuestra propuesta, XMI en su versión 2.0 tal y como recomiendan en [22], será utilizado para la comunicación entre el modelo y la aplicación de generación de código.

Ésta aplicación de generación de código tendrá por objetivo transformar el modelo en código generado. Nuestro objetivo, una vez tenemos definido el modelo, es obtener código Java [23] listo para ser ejecutado.

Este proceso se denomina M2T ("*Model 2 Text*"), que como su propio nombre indica, obtiene texto a partir de un modelo. En otras palabras, a partir de un modelo, unas cajitas o un diagrama, obtiene un lenguaje que puede ser interpretado por una aplicación y sirve como medio de comunicación entre ambas partes.

Hay varias formas de conseguirlo, pero en nuestra solución abogamos por el uso de motores de transformación a través de plantillas JET ("*Java Emitter Templates*") [24].

Las JET o "*Java Emitter Templates*", tecnología basada en EMF, son una serie de plantillas que permiten la generación de código automático (Java, XML, etc.) a partir del motor de transformación JET.

3.1.2.3 Casos de éxito en MDE

Casos de éxito en la utilización de la metodología MDE son "The Palladio Editor" [25], caso de uso sobre la comparación del desarrollo mediante código y el desarrollo de software mediante metodologías MDE con una misma funcionalidad. En él comparan los beneficios de usar la metodología MDE en términos de eficiencia, tiempo de esfuerzo dedicado, y calidad del código.

Otro caso de éxito de uso de metodologías MDE es el editor de videojuegos "*Gade4All*" [12], editor de videojuegos que proponía el uso de ingeniería dirigida por modelos y que permitía crear videojuegos desde cero eximiendo al usuario de cualquier tipo de conocimiento de programación, únicamente centrándose en la parte creativa de la misma.

[4] El Schema XML describe la estructura y las restricciones de un documento XML.

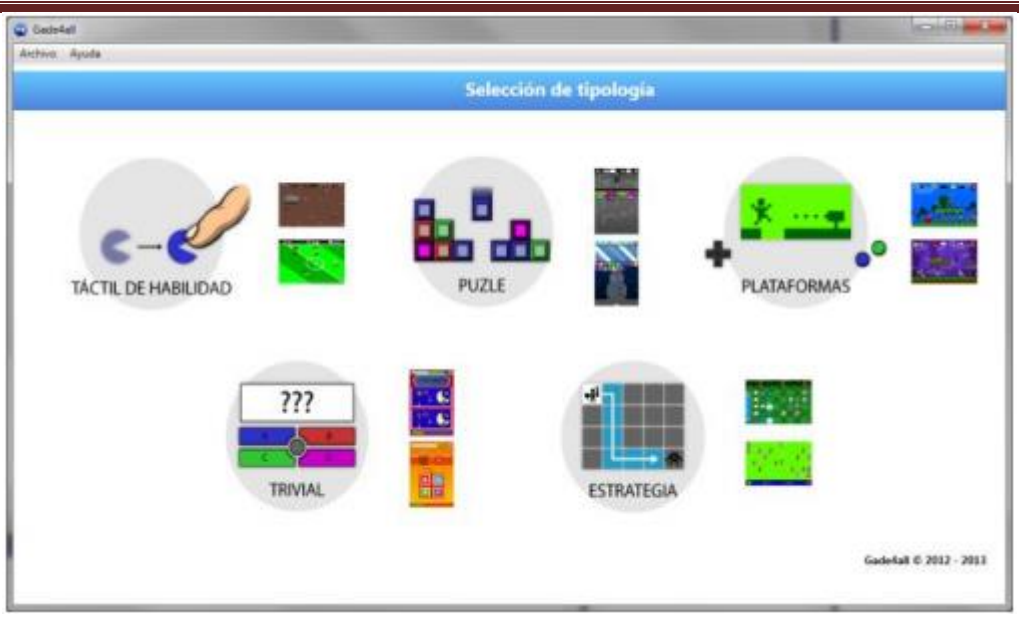


Fig. 1 Interfaz de la herramienta de creación de videojuegos Gade4all

Más casos de éxito relativos a MDE los podemos encontrar en "*Using Domain-Specific Modeling towards Computer Games Development Industrialization*" [26], focalizado en cómo los lenguajes de dominio específico pueden ser usados en conjunto dentro del proceso de desarrollo de software para que tanto los desarrolladores como los diseñadores puedan trabajar de una manera más productiva.

El uso de esta metodología es la clave para nuestra propuesta, pues se ajusta de manera eficiente a partir de la definición de un modelo que represente una abstracción al problema de la creación de patrones de comportamiento.

En [27], abogan por la creación de un lenguaje específico de dominio llamado *Monaco* con el objetivo de crear un *framework* que permita interactuar con una máquina de control basado en eventos, permitiendo la construcción de sistemas con un componente muy bajo en términos de esfuerzo dentro del dominio de la automatización.

En [28] , hacen uso de la tecnología JET para la generación de código automático, presentando un enfoque para gestionar la complejidad basándose en una combinación de técnicas de desarrollo orientadas a aspectos y a la ingeniería dirigida por modelos (MDE).

3.2 Creación de comportamientos en herramientas comerciales

Una de las áreas que más controversia causaba entre los usuarios, era aquella destinada a la definición de los comportamientos de los personajes no jugadores dentro de la aplicación. Los videojuegos modernos están empezando a ser más complicados y modelar personajes "inteligentes" se ha vuelto un tópico interesante [29].

Si bien crear un videojuego con las herramientas actuales es relativamente sencillo como veremos en puntos posteriores, las cosas se complican cuando se desea incrementar el grado

de complejidad de las acciones que ocurren en nuestro videojuego. No hay una parte común en las acciones de los personajes no jugadores en un videojuego que puedan establecerse como un patrón configurable. Cada videojuego tiene su propia "inteligencia".

Diferentes alternativas en la creación de patrones de comportamiento son accesibles por el usuario, que van desde editores con asistentes en la creación de patrones de comportamiento, hasta "*frameworks*" que proveen las herramientas necesarias en términos de codificación para la creación de éstos.

3.2.1 Editores con Asistente

A continuación se analizarán las diferentes alternativas que presentan ciertas similitudes en lo que respecta a nuestra propuesta. En este caso, hablaremos de editores con asistente.

Un editor con asistente es aquel que, para ciertas funcionalidades, provee de herramientas guiadas o tutoriales guiados que permiten la configuración de cierto aspecto de un videojuego con las herramientas que provee la aplicación. Normalmente suelen representarse mediante *Wizards*[5].

3.2.1.1 Game Maker

Game Maker Studio [30] es una herramienta para desarrollar videojuegos basada en un lenguaje de programación interpretado llamado GML. Está diseñado para permitir a los usuarios desarrollar videojuegos en diversas plataformas (iOS, Android, PC), fácilmente sin poseer conocimientos de un lenguaje de programación.

Fig. 2 Logotipo de Game Maker Studio

Con las herramientas simples es posible realizar juegos sencillos, pero si deseamos realizar videojuegos más complejos la necesidad de un lenguaje de programación se hace patente, tratando en todo momento de minimizar la escritura de éste.

En lo que respecta a la parte de edición de enemigos, tenemos muchas posibilidades de creación, basándose todas ellas en el concepto de acciones asociadas a eventos. Es decir, dado un enemigo, quiero que ocurra esto cuando se dé esta situación.

[5] *Wizard,* del inglés, mago, magia, puede definirse como un asistente de configuración que suelen presentarse mediante interfaces de usuario mostrándose al usuario como un conjunto de diálogos. Suelen llevar al usuario a través de ellos mediante una serie de pasos pre-definidos acometiendo tareas que suelen ser complejas, facilitándolas en su mayor medida.

Las acciones indican cosas que ocurren en el juego, y están localizadas en los eventos de los objetos. Cualquiera que sea el evento que tome lugar, esas acciones son ejecutadas, resultando en un comportamiento específico para los objetos. Vienen en forma de sets de acciones y siguen el concepto de *drag & drop* sobre el destino o sobre el objeto que se desee, como por ejemplo para el caso de las colisiones.

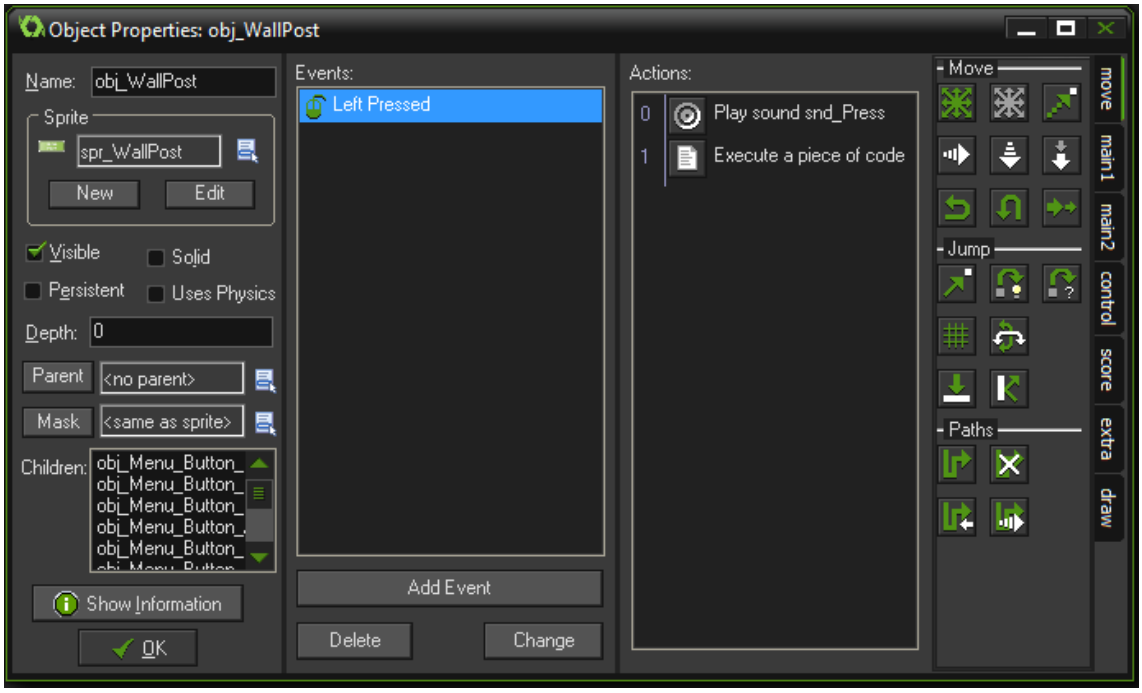


Fig. 3 Opciones de configuración de distintos tipos de eventos y acciones en Game Maker Studio

Las opciones que presenta son muy variadas y se pueden combinar dando resultados muy interesantes.

Como ventaja en comparación a nuestra propuesta, en la mayoría de los casos, parametriza la entrada de estas acciones, pudiendo variar por ejemplo las coordenadas, lo que significa que el usuario ha de poseer conocimientos mínimos sobre vectores de posición en un espacio 2D, entre otros.

Fig. 4 Pasos para completar la asignación de eventos y acciones sobre un actor en Game Maker

Si no nos centramos únicamente en la parte de acciones, otra posibilidad de crear estos comportamientos es mediante el uso del lenguaje de programación GML, lo que significa que para llevar a cabo esta etapa, el usuario ha de poseer buenos conceptos acerca de programación.

```
//argument0 - первое направление
//argument1 - второе направление
var diff;
diff = (argument1 - argument0) mod 360;
if diff < 0
    diff += 360;
if abs(diff) > 180
    return (360 - abs(diff)) * -sign(diff);
else
    return diff;
```

Fig. 5 Fragmento de código del lenguaje GML

Estas acciones no son modificables a posteriori, por lo que una vez establecidas permanece como si de una "blackbox" se tratara, requiriendo de la aplicación Game Maker para su modificación. Estas modificaciones son posibles con T2Game, pues es posible editarlas a posteriori.

Si bien permite muchas opciones de configuración en cuanto a la parte de comportamientos, permitiendo crear comportamientos no ligados expresamente a una taxonomía, se han de realizar mayormente mediante introducción de código, que es precisamente lo que intentamos evitar con nuestra propuesta.

Es una herramienta excepcional, pero en la gran mayoría de los casos es necesaria una buena dosis de conocimientos matemáticos y/o de programación que no están al alcance de cualquier usuario.

3.2.1.2 Stencyl

Stencyl [31], desarrollado por StencylWorks, es una herramienta de creación de videojuegos que proporciona a los diseñadores de un editor gráfico para la realización de videojuegos para diversas plataformas.

Fig. 6 Logotipo de la herramienta Stencyl, de StencylWorks

Al igual que ocurre con Game Maker Studio, crear videojuegos de forma sencilla es relativamente fácil, pero una vez queremos dotar a nuestro videojuego de más funcionalidad las cosas se complican. Está basado en la definición de escenas y actores, en la que cada escena se corresponde con un nivel en el juego.

En lo que respecta a la creación de comportamientos de enemigos, Stencyl usa el concepto de "*Behaviours*[6]". Los "*Behaviours"* son entidades reusables, habilidades configurables que se adjuntan a los tipos de actor o escenas. Cada "*Behaviour*" puede ser configurado para cada Actor, donde además puede ser parametrizado, lo que se llaman Atributos.

[6] Behavior (US) o Behaviour (UK) del inglés, significa comportamiento, conducta.

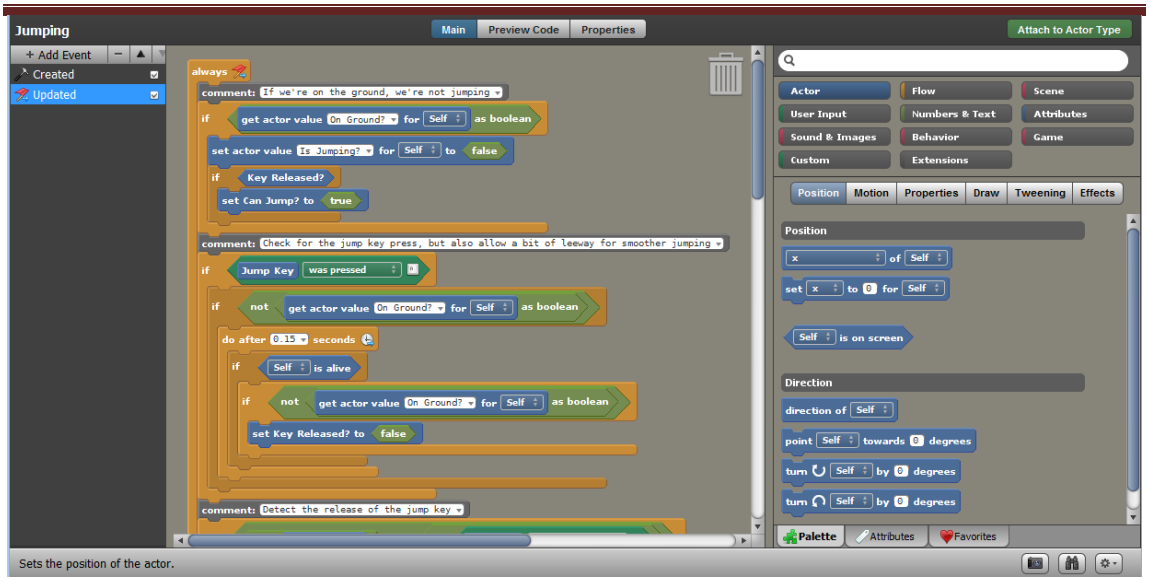


Fig. 7 Pantalla de creación de comportamientos o Behaviours en Stencyl

Estos "*Behaviours*" van ligados a eventos. Estos eventos ocurren recibiendo una respuesta de los llamados "*Behaviours*". Esto suena muy bien de cara a la configurabilidad, pues se basa en el concepto de cajitas que se van interconectando formando una especie de puzle. A simple vista, un programador puede asegurar de que se trata de una especie de "pseudocódigo" del código que a más bajo nivel se va a ejecutar.

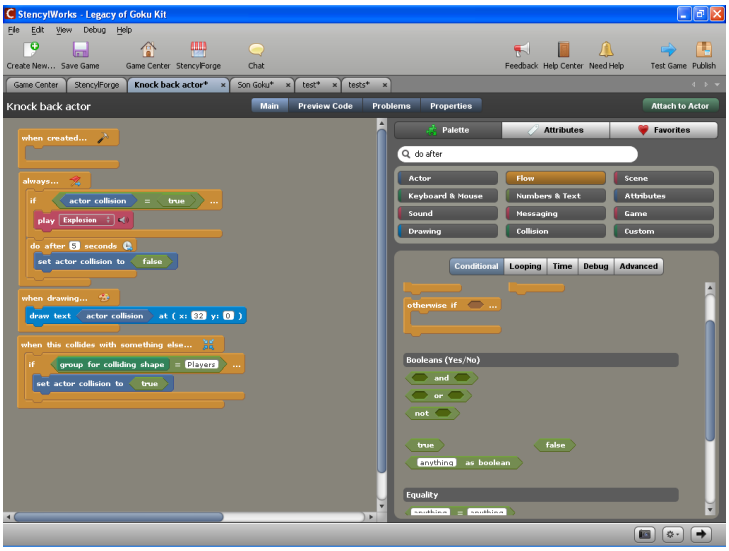


Fig. 8 Posibilidades de creación de Behaviours en Stencyl

No requiere de conocimientos de un lenguaje específico de programación, pero es recomendable tener nociones acerca del funcionamiento de eventos en el ámbito de la informática.

No hay un lenguaje claramente definido para la creación de patrones de comportamiento, pues como comentamos anteriormente, se basa en el concepto de interconexión de nodos o cajitas formando una especie de puzle.

3.2.1.3 Construct-2

Desarrollado por la compañía Scirra [32], Construct-2 es un editor de videojuegos 2D especialmente destinados a plataformas web (HTML5 [33]) destinado a usuarios con pocos conocimientos en la materia, permitiéndoles la rápida creación de videojuegos usando la filosofía "*drag & drop*" y utilizando un editor de comportamientos que tiene como particularidad la posibilidad de añadir más funcionalidad mediante un sistema de lógica.

Fig. 9 Logotipo de la aplicación Construct-2, desarrollada por la compañía Scirra

Estos comportamientos constituyen lo que se llama el "*Event System*", el cual dado una serie de eventos, éstos contienen sentencias condicionales y "*triggers*" (disparadores), que una vez se produzcan en el tiempo, se ejecutan según las condiciones preestablecidas.

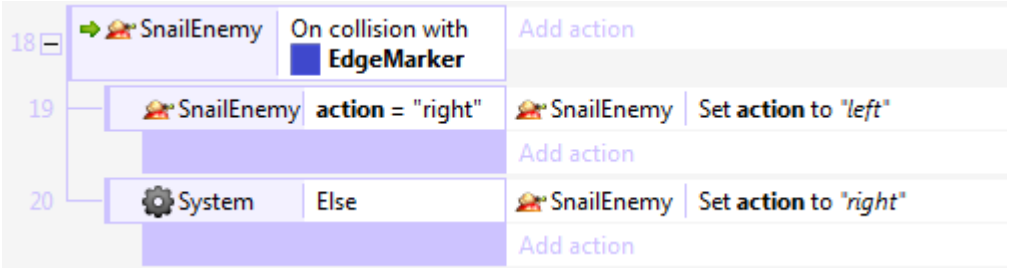


Fig. 10 Sistema de asignación de eventos contenido en la herramienta Construct-2

Pueden aplicarse conceptos de lógica tales como sentencias AND y OR permitiendo sofisticados sistemas, pero con la particularidad de que han de ser programados mediante un lenguaje de programación.

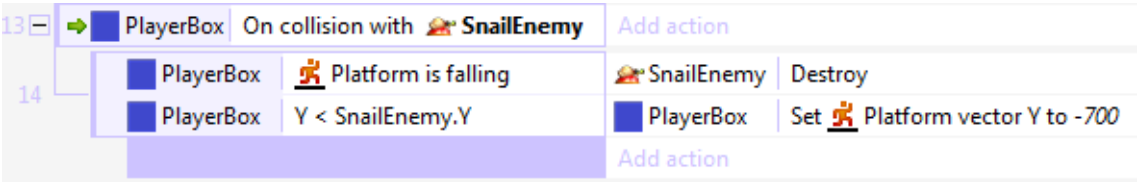


Fig. 11 Inclusión de 'Expressions' en Behaviors en la herramienta Construct-2

En lo que respecta a los "*Behaviors*", son paquetes "pre-cocinados" para añadir funcionalidad a los tipos de objetos, cada uno con un conjunto de propiedades modificables. La finalidad de estos "*Behaviors*" no es establecer las propiedades finales de los comportamientos enemigos, sino que han de acompañarse con el anteriormente mencionado sistema de lógica. Son, esencialmente, atajos para mejorar la productividad en cuanto a la definición de comportamientos, comúnmente llamados en inglés "time-savers".

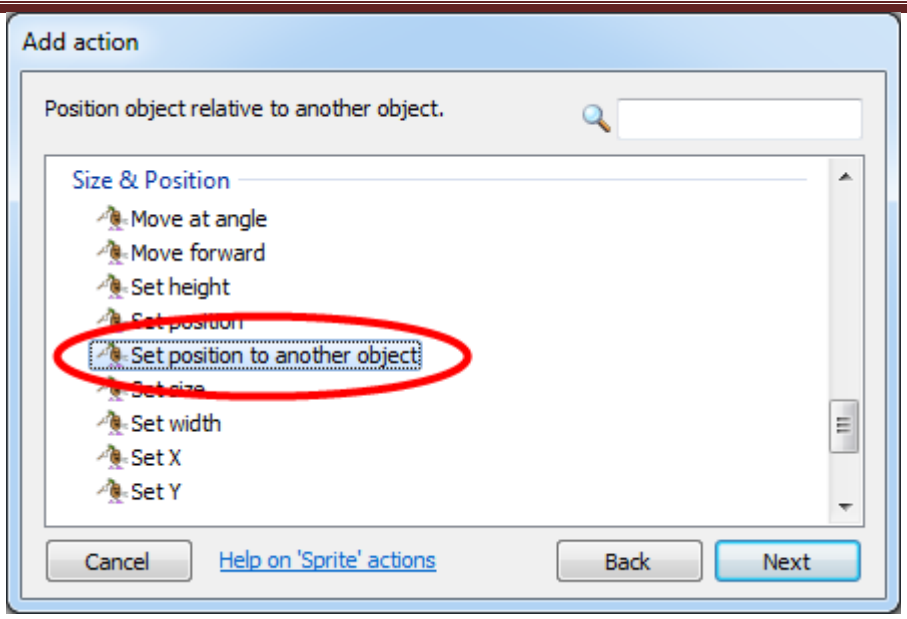


Fig. 12 Asignación de acciones a objetos en Construct-2

Una de las particularidades que mantiene es que los "*Behaviors*" usados en Contruct-2 son aplicados a todos los objetos de la misma instancia, al igual que hacemos en nuestra solución.

Mantiene bastantes similitudes con respecto a nuestra solución, aunque exige de un mínimo de conocimientos algebraicos y de lógica para ciertas acciones de edición.

3.2.2 Frameworks / Engines

3.2.2.1 Blueprints de Unreal

Blueprints Visual Scripting [34] es un sistema de definición del "*gameplay*[7]" de un videojuego creado por el equipo de Unreal Engine basado en el concepto de nodos interconectados. Permite implementar o modificar cualquier elemento del videojuego, como reglas, ítems, etc.

Fig. 13 Logotipo Unreal Engine

[7] *Gameplay*, del inglés, reproducción de juego, se dice del conjunto de acciones que puede realizar un jugador para interactuar con el juego, o la forma en la que éste interactúa con el propio jugador. [56]

Cada nodo representa una entidad, un evento, una función o una variable, que entrelazados forman un conjunto de nodos, o lo que se conoce como el "*Graph Area*[8]". Blueprints es un sistema de eventos y funciones, lo que significa que cuando un evento ocurre en el juego, las funciones actúan sobre los actores del videojuego.

Una de las ventajas de Blueprints es que rápidamente podemos crear un prototipo de varios elementos dentro del videojuego, todo ello sin la necesidad de programar. En muchos casos, es posible crear elementos del "*gameplay*" incluso más rápido que con lo que se tardaría con un lenguaje de programación.

Brinda la posibilidad de integrar características complejas que previamente necesitaban de un buen puñado de líneas de código, permitiendo a los diseñadores usar "virtualmente" todos aquellos conceptos y herramientas que generalmente sólo estaban disponibles para los programadores.

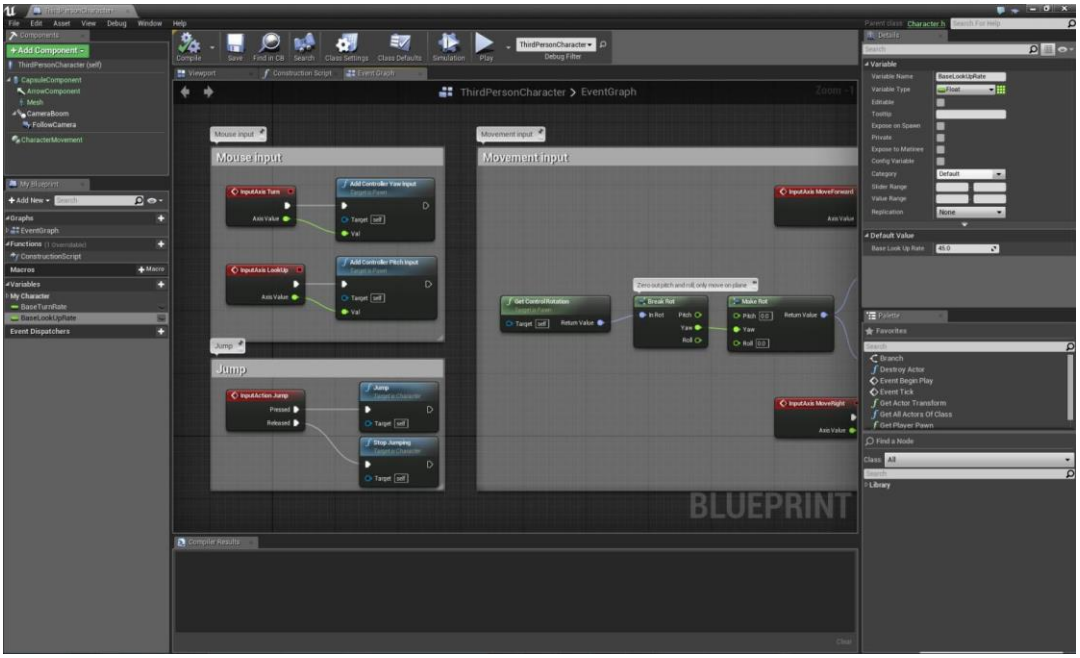


Fig. 14 Editor Blueprint Visual Scripting

No obstante, Blueprints no vino al mundo para sustituir al código, sino para complementarlo. Está pensado principalmente para que un programador pueda crear más nodos de manera que los desarrolladores utilicen dichos nodos, para de esta manera éstos puedan tener un control total sobre el flujo del videojuego.

En cierta medida puede asimilarse a nuestra solución, pues puede verse como un conjunto de macros autoejecutadas por un motor de transformación interno. No obstante, requiere un punto de apoyo "programático" ya que no está pensado inicialmente para que sustituya directamente al código.

[8] *Graph Area* hace alusión al lugar donde se presentan los diferentes nodos.

3.2.2.2 Unity

Unity [35] es un framework multiplataforma creado por Unity Technologies destinado a la creación de videojuegos multiplataforma.

Fig. 15 «Unity® logo» de Unity Technologies [9]

Es una herramienta que puede usarse conjuntamente con otras destinadas sobre todo al ámbito de diseño gráfico como son 3ds Maya [36], ZBrush [37], entre muchas otras, destinada principalmente a usuarios con amplios conocimientos en el ámbito de los videojuegos, concretamente en la parte programática.

Para la definición de enemigos, y básicamente para la definición de cualquier actor dentro de un videojuego, provee un sistema de scripting basado en Mono [38] en el cual se pueden moldear todos aquellos elementos que vayan a componer nuestro videojuego.

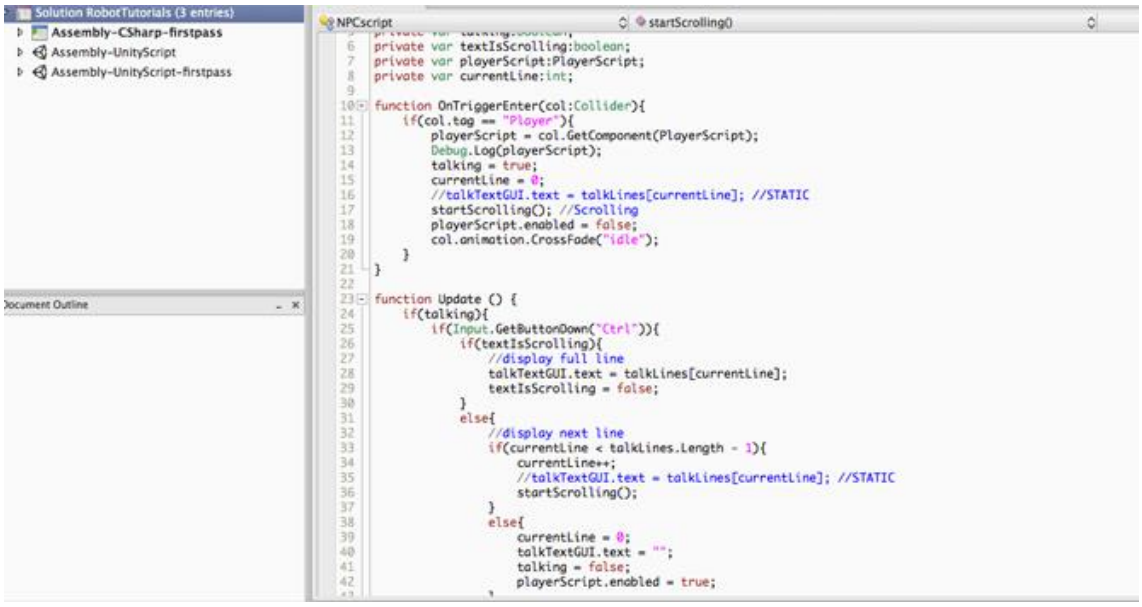


Fig. 16 Unity Script

Esto tiene su parte buena, pues no tiene limitaciones en cuanto a abstracciones realizadas para moldear cualquier tipo de característica dentro de un videojuego. Todo aquello que se pueda realizar programáticamente, puede tener cabida en Unity.

[9] http://unity3d.com/public-relations/brand

Pero por el contrario, todos aquellos usuarios que carezcan del suficiente conocimiento en esta área van a encontrar una barrera considerablemente difícil de superar. La curva de aprendizaje de esta tecnología es bastante elevada.

```
void Chasing ()
{
    // Create a vector from the enemy to the last sighting of the
player.
    Vector3 sightingDeltaPos = enemySight.personalLastSighting -
transform.position;

    // If the the last personal sighting of the player is not close...
    if(sightingDeltaPos.sqrMagnitude > 4f)
        // ... set the destination for the NavMeshAgent to the last
personal sighting of the player.
        nav.destination = enemySight.personalLastSighting;

    // Set the appropriate speed for the NavMeshAgent.
    nav.speed = chaseSpeed;
}
```

Fig. 17 Métodos y variables de apoyo a los programadores

En cuanto a la definición de comportamientos de enemigos, no posee un editor específico para ello, sino que cede la responsabilidad a las habilidades programáticas de los usuarios. Provee una serie de métodos que sí pueden tenerse en cuenta a la hora de plasmar dichos comportamientos (intersección entre rectángulos, usados para las colisiones entre actores, clases Vectores que delimitan un punto en el espacio y facilitan la colocación de los actores en el espacio dimensional o tridimensional, etc.).

Si bien permite una edición cuasi-infinita de comportamientos, prevalece sobre todo el componente programático, siendo algo indispensable para la elaboración de los comportamientos dentro del videojuego en esta tecnología.

3.2.2.3 Cocos2D

Cocos2D es un framework de código libre u "*open source*[10]" escrito en lenguaje de programación Python [39] destinado a la creación de videojuegos 2D. Fue creado por Ricardo Quesada, desarrollador de videojuegos, junto con un grupo de amigos, naciendo como un motor de desarrollo de videojuegos simples. Como curiosidad, debe su nombre a que se desarrolló en la ciudad de Los Cocos, Argentina.

[10] *Open source* se define como el software que permite el acceso a su código de programación, facilitando modificaciones por parte de programadores ajenos.

Fig. 18 Logotipo del framework Cocos2D

Actualmente contiene muchas vertientes, principalmente destinadas a la consecución de un videojuego para una plataforma específica, como puede ser Cocos2D-HTML5 [40] (destinada a videojuegos para la web), Cocos2D-XNA [41] (destinada a videojuegos para plataformas tales como PC o Xbox), entre otros.

Para la definición de los elementos del videojuego, se basa principalmente en la codificación mediante lenguaje de programación *Objective-C* [42], lenguaje propietario de *Apple* [43], y *Python*. Otras vertientes permiten esta codificación en otros lenguajes, como pueden ser *Javascript* [44] o *C#* [45].

Posee además diferentes editores para, por ejemplo, la definición de actores, la definición del sistema de físicas, definición de archivos de audio, etc.

```
// Sample of enemy creation in the HelloWorldLayer class
-(void)addEnemyAtX:(int)x y:(int)y {
  CCSprite *enemy = [CCSprite spriteWithFile:@"enemy1.png"];
  enemy.position = ccp(x, y);
  [self addChild:enemy];
}

// in the init method - after creating the player
// iterate through objects, finding all enemy spawn points
// create an enemy for each one
for (spawnPoint in [objectGroup objects]) {
  if ([[spawnPoint valueForKey:@"Enemy"] intValue] == 1){
    x = [[spawnPoint valueForKey:@"x"] intValue];
    y = [[spawnPoint valueForKey:@"y"] intValue];
    [self addEnemyAtX:x y:y];
  }
}
```

No posee un editor específico de comportamientos de enemigos, pues han de codificarse íntegramente en lenguaje de programación. Para ello, provee una serie de métodos de ayuda a la confección de la solución, como parte del framework que representa. Al igual que ocurre con el framework *Unity*, esta creación de comportamientos es prácticamente infinita, pero exige al usuario de unos conocimientos avanzados en programación.

Capítulo 4. Descripción de la propuesta

Partiendo de las ventajas y desventajas que se pudieron observar anteriormente, creemos que nuestra propuesta podría ser la solución a la hora de facilitar el modelado de los patrones de comportamiento relativo a los enemigos dentro de los videojuegos de taxonomía plataformas.

Para comprobar la idoneidad de la solución propuesta para el modelado de procesos de negocio y desarrollo de software dirigido por modelos, se ha construido el siguiente prototipo: T2Game.

4.1 Arquitectura

A continuación vamos a explicar detalladamente la arquitectura propuesta para nuestra solución. Estará compuesta por los siguientes puntos:

- Metamodelo.
- Modelo.
- Sintáxis Gráfica: editor
- Importación al motor de transformación JET.
- Plantillas JET.
- Proyecto *android_platform_miw*.
- Proyecto *Gade4all*.

4.1.1 Metamodelo

El metamodelo define los elementos de un lenguaje de modelado (metaclases), las relaciones entre ellos (metaasociaciones) y sus restricciones. Es decir, define el dominio sobre el que vamos a actuar, que en este caso es el modelado de patrones de comportamiento de enemigos en videojuegos de taxonomía plataformas. Este metamodelo define la sintaxis abstracta y la semántica estática.

La sintáxis abstracta describe los conceptos en el lenguaje y las relaciones estructurales entre ellos. Se aplican unas reglas o restricciones que determinarán que un modelo formado con este metamodelo es válido. A partir de una sintaxis abstracta podemos definir una o varias sintaxis concretas, definiendo cómo los elementos del metamodelo aparecen en una notación textual o gráfica utilizable por personas [14].

La semántica estática hace referencia a las restricciones de “significado” sobre los modelos.

Para el caso de nuestra propuesta, presentamos un conjunto de entidades todas ellas interrelacionadas entre sí con una serie de restricciones, que son:

Behaviour: es el nodo principal, aquel que engloba a otras entidades formando el conjunto completo del metamodelo. Estará compuesto por cero o varias entidades *When* así como cero o varias entidades *Action*. Representa el conjunto global del patrón de comportamiento.

When: entidad cuyo objetivo es plantear una condición en la elaboración del modelo. Lleva consigo un conjunto de identificadores predefinidos, permitiendo al usuario elegir entre un conjunto de ellas. Está formado además por entidades *Condition*, *Otherwise* y *Action*. Es una entidad que puede ir de manera independiente, sin depender de otras identidades.

Condition: entidad asociada directamente a la entidad *When* que delimita la acción al identificador preasociado, llamado *idCondition*. Siempre que se construya una entidad *When*, la entidad *Condition* debe estar presente.

Los identificadores preasociados válidos (*idCondition*) para la entidad *Condition* son los siguientes:

- *IsThereWall*
- *playerIsNear*
- *playerHasEnoughLife*
- *playerHitsMe*

Otherwise: entidad opcional asociada a la entidad *When*. Plantea la condición opuesta a la condición planteada por la entidad *When*. Todo aquello que la entidad *When* no logre satisfacer, es realizado por la entidad *Otherwise*. Puede aparecer (n-1 < x < n) veces, siendo x la entidad *Otherwise* y n la entidad *When*.

Action: entidad que puede ir de manera independiente, estableciendo una acción dentro del comportamiento. Estas acciones están predefinidas de antemano por un conjunto finito de ellas, mediante la variable *idAction* (o identificador de la acción).

Los identificadores preasociados válidos (*idAction*) para la entidad Action son los siguientes:

- *stop*
- *turnAround*
- *moveHorizontal*
- *startFire*
- *stopFire*
- *killPlayer*

Si la entidad *Action* va asociada a una entidad *When*, esta *Action* se procesará cuando la *Condition* de la entidad *When* sea satisfactoria. Si en cambio va asociada a la entidad *Otherwise*, se procesará cuando la entidad *Condition* de la entidad *When* no se satisfaga.

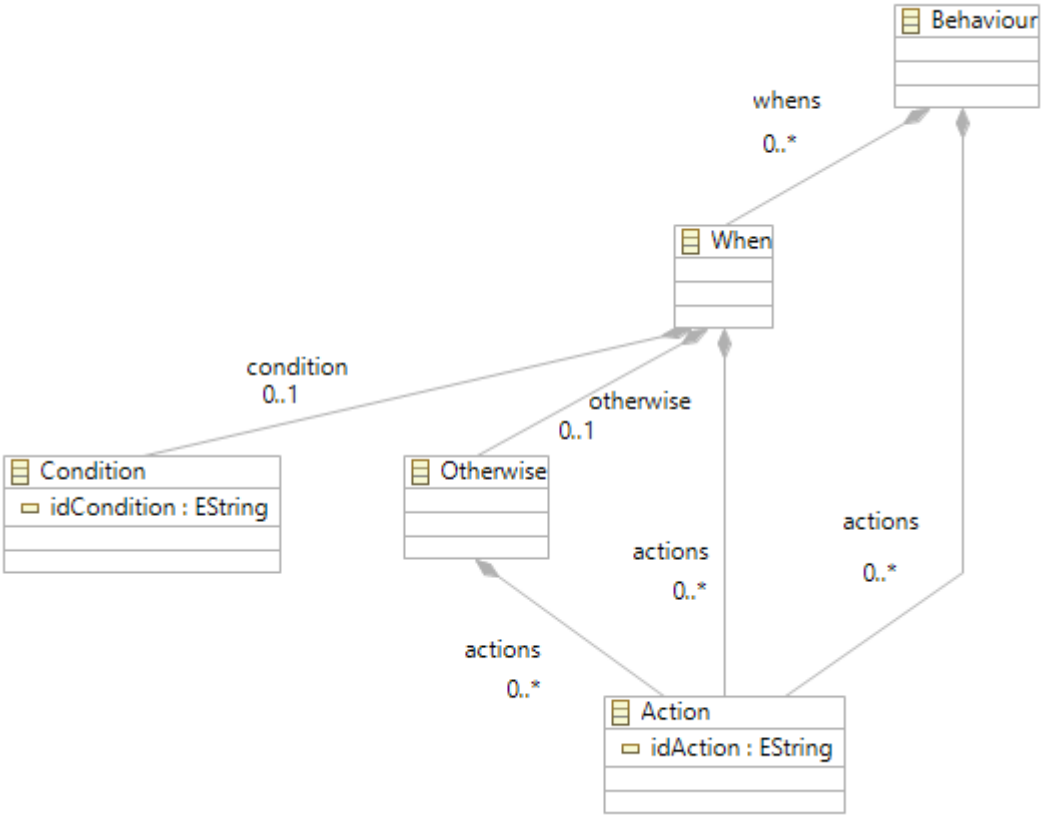


Fig. 19 Sintaxis abstracta del metamodelo de la solución propuesta

El metamodelo es la clave de la ingeniería dirigida por modelos (MDE) y es imprescindible para la construcción de lenguajes específicos de dominio (DSL), validación de modelos, transformación de modelos, generación de artefactos e integración de herramientas [14].

Este metamodelo fue diseñado bajo las herramientas provistas por Eclipse Modeling Framework (EMF) a través de Ecore, modelo EMF que define los conceptos que se van a manipular.

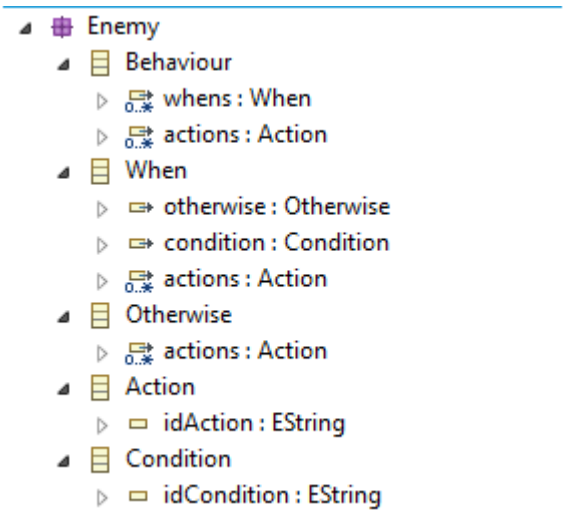


Fig. 20 Definición del metamodelo Ecore

4.1.1.1 Eclipse Modeling Framework (EMF)

Eclipse Modeling Framework, por sus siglas en inglés EMF, es un framework de modelado y generación de código que facilita la construcción de aplicaciones basadas en un modelo de estructura de datos. A partir de una especificación XMI, EMF provee las herramientas y el soporte necesario para producir un conjunto de clases Java para el modelo, a través de una serie de clases "*Adapter*[11]" o adaptadores que permiten la creación y edición del modelo mediante editores visuales y/o línea de comandos.

Fig. 21 Logotipo de Eclipse Modeling Framework (EMF)

EMF es un estándar para los modelos de datos, estando soportado por gran mayoría de tecnologías y frameworks.

En nuestro caso, haremos uso de la tecnología que EMF provee mediante el uso de plugins[12] que nos permitirá la construcción y elaboración tanto del metamodelo, el modelo así como la parte de transformación de código.

Para la parte desarrollo del modelo, haremos uso de la especificación XMI en su versión 2.0, usada por defecto por la tecnología EMF en cuanto a persistencia de datos.

4.1.1.1.1 XMI

XMI o "*XML Metadata Interchange*" es una especificación estándar creada por el Object Management Group (OMG) [46] para el intercambio de metainformación, escrita para una rápida compartición de modelos entre diferentes herramientas de modelado.

Se desarrolló siguiendo el Schema XML y es comúnmente usada como medio de comunicación, intercambiando información entre las herramientas de modelado y software de generación de código como parte del Model-Driven Engineering (MDE).

En nuestra solución, esta especificación será la encargada de comunicar el modelo con el motor de transformación JET.

[11] Adapter, del inglés, adaptador, es un patrón de diseño en el que uno de sus principales propósitos es la reutilización de una clase que coopera con clases no relacionadas entre sí o que presentan interfaces incompatibles.
[12] Plugin, del inglés, enchufable, hacer referencia a los complementos que presenta una aplicación aportándole un nueva funcionalidad generalmente muy específica.

```
▼<Enemy:Behaviour xmlns:xmi="http://www.omg.org/XMI" xmlns:Enemy="Enemy" xmi:version="2.0">
  ▼<whens>
    ▼<otherwise>
       <actions idAction="turnAround"/>
       <actions idAction="moveHorizontal"/>
     </otherwise>
     <condition idCondition="IsPlayerNear"/>
     <actions idAction="stop"/>
   </whens>
  ▼<whens>
     <condition idCondition="playerHitsMe"/>
     <actions idAction="killPlayer"/>
   </whens>
 </Enemy:Behaviour>
```

Fig. 22 Ejemplo de especificación XMI

Para ello, se compone de una serie de elementos que son generados a partir del modelo. Estos elementos son:

4.1.1.2 Elemento <Behaviour>

Es el elemento raíz, el Root Element. De él penden otra serie de elementos que unidos formarán el conjunto de reglas/restricciones que conforman este DSL.

Estos elementos pueden ser <When>, <Condition>, <Otherwise> y <Action>.

```
<Behaviour>
    ...
</Behaviour>
```

4.1.1.3 Elemento <When>

El elemento *When* es usado en aras de obtener una condición con destino positivo. Es decir, todo aquello que contempla son condiciones afirmativas. En este prototipo no está contemplada la condición negativa, pero añadirla no sería tarea compleja, por lo que la enmarcaremos en la fase de trabajo futuro.

Va asociado directamente con el elemento <Condition>, tratado a continuación.

```
<When>
    <condition idCondition="" />
    <action idAction="" />
    <otherwise>
        ...
    </otherwise>
</When>
```

4.1.1.4 Elemento <Condition>

El elemento *Condition* va condicionado al elemento *Whens*. Especifica mediante el valor de un atributo la funcionalidad o condición que se va a contemplar dentro del desarrollo del videojuego.

```
<When>
    <condition idCondition="IsPlayerNear | playerHitsMe" />
    <action idAction="" />
    <otherwise>
```

```
        ...
    </otherwise>
</When>
```

Los valores que puede adquirir el atributo *idCondition* hacen alusión al conjunto de métodos predefinidos en el videojuego destino que son susceptibles de ser llamados para realizar evaluaciones. Anteponen la vocal c_ (de condición) y son los siguientes:

4.1.1.4.1 IsThereWall

Condición que evalúa si hay un muro. Esta evaluación se hace mediante la comparación de los rectángulos de colisión que envuelven al enemigo y al muro en cuestión. Éstos rectángulos de colisión son utilizados para poder hacer operaciones entre rectángulos, tales como intersección de rectángulos, unión, etc.

En el código del videojuego destino, hace alusión al siguiente método:

```
private boolean c_IsThereWall(){
        // Calculate tile position based on the side we are walking towards.
        float posX = position.getX() + this.getBoundingRectangle().width() / 2 *
(int)direction;
        int tileX =(int)Math.floor(posX / Tile.Width) - (int)direction;
        int tileY = (int)Math.floor(position.getY() / Tile.Height);

        if (level.GetCollision(tileX + direction, tileY - 1) ==
TileCollision.Impassable)
        {
            return true;
        }
        else if(level.GetCollision(tileX , tileY) == TileCollision.Impassable &&
                level.GetCollision(tileX + direction, tileY) == TileCollision.Passable)
        {
            return true;
        }
        return false;
    }
```

4.1.1.4.2 playerIsNear

Condición que evalúa si el personaje jugador, es decir, el protagonista, está cerca con respecto a un enemigo dado. Por defecto, establecemos una distancia de 150 pixeles. Esta distancia podría parametrizarse dando más volatilidad, por lo que será contemplada en aspectos futuros de la solución.

```
private boolean c_playerIsNear() {
        float distance = 150;
        return RectangleExtensions.DistanceBetween(level.getPlayer().getPosition(),
this.getPosition()) <= (distance*level.ScreenWidth()/800);
}
```

4.1.1.4.3 playerHasEnoughLife

Condición que evalúa si el personaje principal o protagonista tiene la suficiente vida como para realizar alguna acción. Por ejemplo, podemos tenerla en cuenta a la hora de atacar a un personaje. Si éste presenta poca vida, podemos dejarle pues interpretamos que está lo suficientemente dañado, o incluso podemos rematarlo, dado que tiene poca vida.

El código que hace alusión a esta condición predefinida es el siguiente:

```
private boolean c_playerHasEnoughLife() {
        return level.getPlayer().getHealth() > 25;
    }
```

Al igual que ocurre con la condición anterior, el valor asignado es estático, por lo que en procesos futuros se podría tener en cuenta a la hora de parametrizar.

4.1.1.4.4 playerHitsMe

Condición que evalúa, mediante los rectángulos de colisión antes comentados, si el personaje principal ha colisionado contra un enemigo, siempre y cuando la salud del enemigo sea superior a 0, es decir, no esté muerto.

El código predefinido para alcanzar esta condición es el siguiente:

```
private boolean c_playerHitsMe() {
        return
this.getBoundingRectangle().intersect(level.getPlayer().getBoundingRectangle()) &&
this.getHealth() > 0;
    }
```

4.1.1.5 Elemento <Otherwise>

El elemento *Otherwise* es la negación del elemento *When*. Es decir, siempre que no se cumpla la condición establecida en el elemento *When*, entrará en funcionamiento aquellas clausulas o acciones establecidas dentro del elemento *Otherwise*. Este elemento es opcional e incluye un número ilimitado de acciones predefinidas o elementos *Action*.

```
<When>
    <condition idCondition="" />
    <action idAction="">
    <otherwise>
        <action idAction="">
        ...
    </otherwise>
</When>
```

4.1.1.6 Elemento <Action>

Es sin duda uno de los elementos más importantes, pues especifica qué acciones se van a ejecutar. Es quien con su significado, establece las acciones llevadas a cabo por el enemigo bajo según qué condiciones. Puede aparecer solitariamente, ir englobado con el elemento *When* e incluso englobado con el elemento *Otherwise*.

El conjunto de acciones que se pueden utilizar es infinito. Estas acciones llevan consigo un atributo identificador que será el encargado de establecer qué acción se va a procesar.

```
<action idAction="moveHorizontal">
<When>
    <condition idCondition="" />
    <action idAction="turnAround">
    <otherwise>
        <action idAction="stop">
        ...
    </otherwise>
</When>
```

El conjunto de valores que puede adquirir el identificador *idAction* son los siguientes:

4.1.1.6.1 stop

Esta acción establece una parada en el movimiento del enemigo, independientemente que esté en movimiento o no.

El código predefinido al que hace referencia esta acción es el siguiente, y vendrá acompañado de la vocal a_+nombreAcción, en alusión a una acción.

```
private void a_stop() {
    this.direction = 0;
}
```

4.1.1.6.2 turnAround

Esta acción tiene como objetivo hacer girar la trayectoria del enemigo respecto al eje X. Es decir, dada una dirección en la que el enemigo está caminando, ejecutando esta acción hará que el enemigo haga un giro de 180 grados, caminando en la dirección opuesta.

El código predefinido que ejecuta esta acción es el siguiente:

```
private void a_turnAround() {
    // Then turn around.
    this.direction = (direction == 1)?-1:1;
}
```

El valor 1 representa la dirección del enemigo mirando hacia la parte positiva del eje X, mientras que el valor -1 representa la dirección del enemigo mirando hacia la parte negativa del eje X.

4.1.1.6.3 moveHorizontal

Esta acción va bastante ligada a la anterior en cuanto a significado, pero su objetivo únicamente es hacer avanzar al enemigo. No tiene en cuenta si la dirección es positiva o negativa. Simplemente hace que avance por el eje X. Si es acompañada por la acción turnAround, establecerán el movimiento más la dirección hacia la que enemigo deberá caminar.

El código predefinido es el que sigue:

```
private void a_moveHorizontal() {
    // Move in the current direction.
    this.direction = (direction == 1)?1:-1;
}
```

4.1.1.6.4 startFire

Esta acción hace que el enemigo dispare. Suele usarse en consonancia con alguna condición, para que dada esa condición, el enemigo dispare.

El código asociado a esa acción es el siguiente:

```
private void a_startFire(long gameTime) {
        BulletEnemy bullet = null;
        firespan = firespan - gameTime;
        if (firespan <= 0)
        {
            //waitTime = MaxWaitTime; //enemy stops.
            this.StopWalking();

            bullet = new BulletEnemy(level, level.getPlayer(),
                    this,
```

```
                    this.getBulletWidth(),//Final width
                    this.getBulletHeight(),//Final Height
                    this.getNameBullet(),//Num of Frames
                    this.getImageBulletNumFrames(),//Name of the bullet image
                    this.getImageBulletWidth(), //Width
                    this.getImageBulletHeight(), //Height
                    Player.lastFaceDirection, //Direction
                    this.getDamageBullet(),//damage
                    this.ImageBulletHasLoop()); //Has loop

            //Image rotation
            bullet.setRotation((float)Math.atan2(level.getPlayer().getPosition().getX()
- this.getPosition().getX(), level.getPlayer().getPosition().getY() -
this.getPosition().getY()));

            {//If we are in the same X-axis of the enemy
                float rotationAux = bullet.getRotation();
                if (Math.abs(rotationAux) > 1.57 && Math.abs(rotationAux) < 1.58)//PI/2
                    bullet.setRotation(0.0f);
            }

            enemyBullets.add(bullet);
            firespan = firedelay;
        }
        UpdateEnemyBullets(gameTime);
        CheckEnemyBulletsAgainstPlayer(gameTime);
    }
```

Para este caso, el disparo del enemigo tiene en cuenta la posición del personaje principal, de tal manera que parezca que estamos acometiendo un disparo "inteligente". Así mismo, experimenta una especie de "*delay*" o retraso de tiempo entre un disparo y otro, para no crear un efecto de superposición de disparo.

Todos estos añadidos pueden tenerse en cuenta para futuras soluciones, pues hay infinidad de formas de parametrización del disparo. Volviendo a nuestro caso, la acción disparar del enemigo provoca un disparo por parte del enemigo hacia el personaje principal.

4.1.1.6.5 stopFire

Esta acción va ligada con la acción *startFire*, y como su propio nombre indica, lanza la acción para que el enemigo deje de disparar al personaje principal.

El código al que hace referencia es el siguiente:

```
private void a_stopFire(){
    firespan = 1;
}
```

Como curiosidad, si nos fijamos en la acción *startFire*, la condición para que un enemigo dispare a nivel de código es que la variable *firespan*, sea menor o igual que 0. En este caso para que el enemigo deje de disparar, sólo nos bastaría con establecer el *firespan* a 1.

4.1.1.6.6 killPlayer

Esta acción compromete dos actores en su ejecución. Uno es el enemigo y otro el personaje principal. Su objetivo es matar al personaje principal, o lo que es lo mismo, establecer la salud del personaje principal a 0.

El código asociado a esta acción es el siguiente:

```
private void a_killPlayer() {
    level.getPlayer().setHealth(0);
    if (level.getPlayer().getHealth() <= 0)
    {
```

```
            OnPlayerKilled(this);
        }
        else
        {
            OnPlayerKilled(null);
        }
    }
```

Esta acción suele usarse muy comúnmente cuando queremos que, tras una colisión del personaje principal con un enemigo, sea el personaje principal quien muera, haciendo creer que ha sido el enemigo quien nos mató.

4.1.2 Modelo

El modelo está desarrollado a partir del metamodelo. Este modelo es una abstracción que está focalizada en desarrollo de aplicaciones específicas de un problema, en nuestro caso, patrones de comportamiento en videojuegos de plataformas. El modelo usa los elementos definidos en el metamodelo para especificar el patrón de comportamiento de un enemigo en videojuegos de plataformas.

Estos modelos están pensados inicialmente para el incremento de la productividad y la compatibilidad entre sistemas, simplificando la etapa de diseño para engrandecer la comunicación entre equipos de trabajo [1].

Un modelo se considera efectivo del paradigma MDE si este modelo tiene sentido desde el punto de vista del usuario y sirve como base para una implementación de una solución, permitiendo el desarrollo de software y sistemas [47].

Estos modelos pueden definirse de varias formas, que son:

- Programáticamente.
- Con plugins de EMF basados en el metamodelo.
- DSLs o lenguajes específicos de dominio.

En nuestro caso, vamos a desarrollar nuestro modelo a partir de uno de los plugins de EMF basándonos en el metamodelo.

Para la definición de nuestro modelo, hemos de especificar para cada entidad un conjunto de propiedades, si es que son necesarias, pues algunas entidades carecen de ellas.

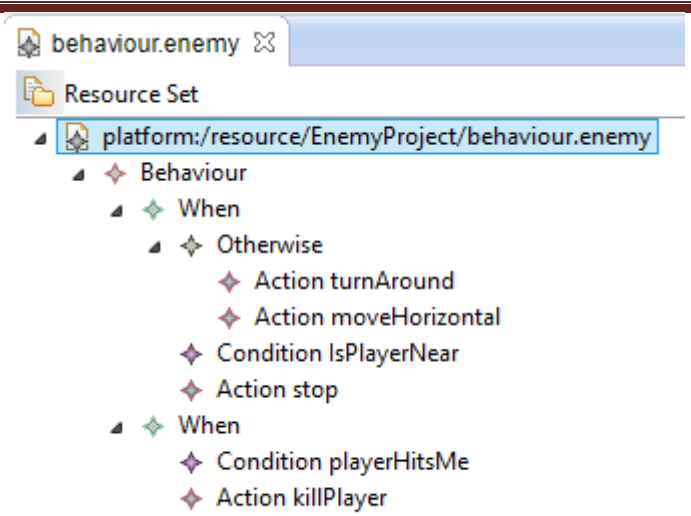


Fig. 23 Desarrollo de un ejemplo de modelo

Este modelo está generado sobre un proyecto Java vacío o *Empty Java Project*, añadiendo un archivo de tipo *Enemy Model*.

Si deseamos agregar más entidades de las ya existentes, simplemente situándonos sobre la entidad padre y añadiendo más hijos o "*child elements*" podremos completar aún más si cabe el patrón de comportamiento de nuestro enemigo.

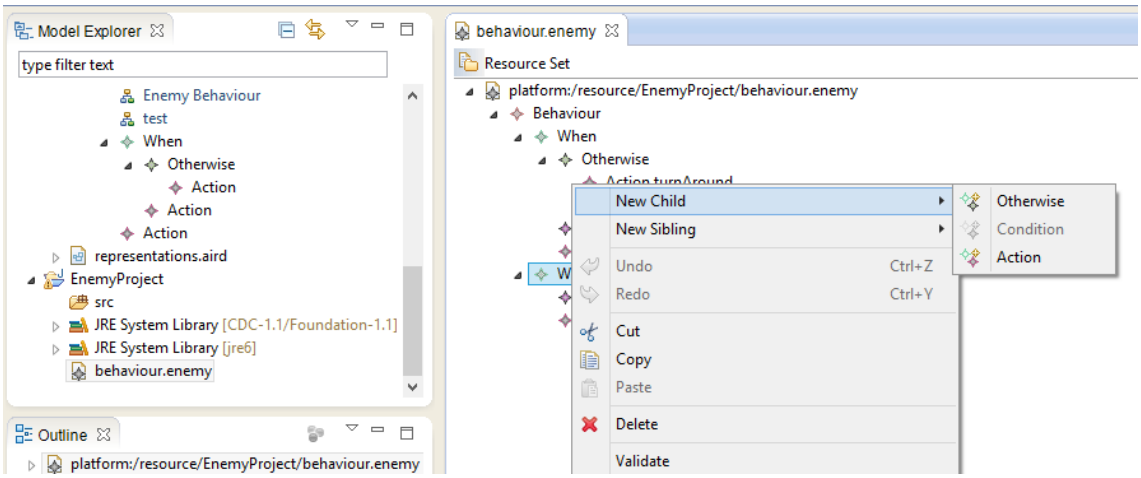


Fig. 24 Añadiendo entidades con EMF

Una vez tengamos configurado correctamente el patrón de comportamiento, podemos visualizar la especificación XMI generada para dicho modelo, con las relaciones entre las entidades especificadas en el metamodelo.

Esta especificación XMI será utilizada como medio de comunicación con el motor de transformación JET.

4.1.3 Sintáxis Gráficas: editor Sirius

Para la parte gráfica que nos permita definir diferentes modelos, se utilizará la herramienta Sirius [48].

Sirius es un proyecto Eclipse que permite crear fácilmente herramientas de modelado gráfico. Se aprovecha de las tecnologías de modelado de Eclipse, incluyendo el framework EMF para la gestión de modelos.

Basado en un enfoque de "*viewpoints*[13]", Sirius hace posible tratar con arquitecturas complejas de dominios específicos, permitiendo un alto nivel de personalización, pudiendo definir estilos, comportamientos, scripts, etc.

Para definir nuestro modelo, en la herramienta de modelado debemos de definir una serie de reglas para que el proceso de modelado se corresponda con el modelo al que queremos llegar. Todo ello siempre dentro de las restricciones que previamente establecimos en el metamodelo.

Para ello, tras definir un *viewpoint* sobre el proyecto *basicEnemy,* definimos las diferentes propiedades que especificarán cómo y de qué manera interactuarán los diferentes nodos en la creación de comportamientos. El proyecto basicEnemy será aquel que contenga todas las especificaciones para poder desarrollar el editor. Está creado a partir de las herramientas de eclipse, teniendo en cuenta el metamodelo que hemos definido previamente.

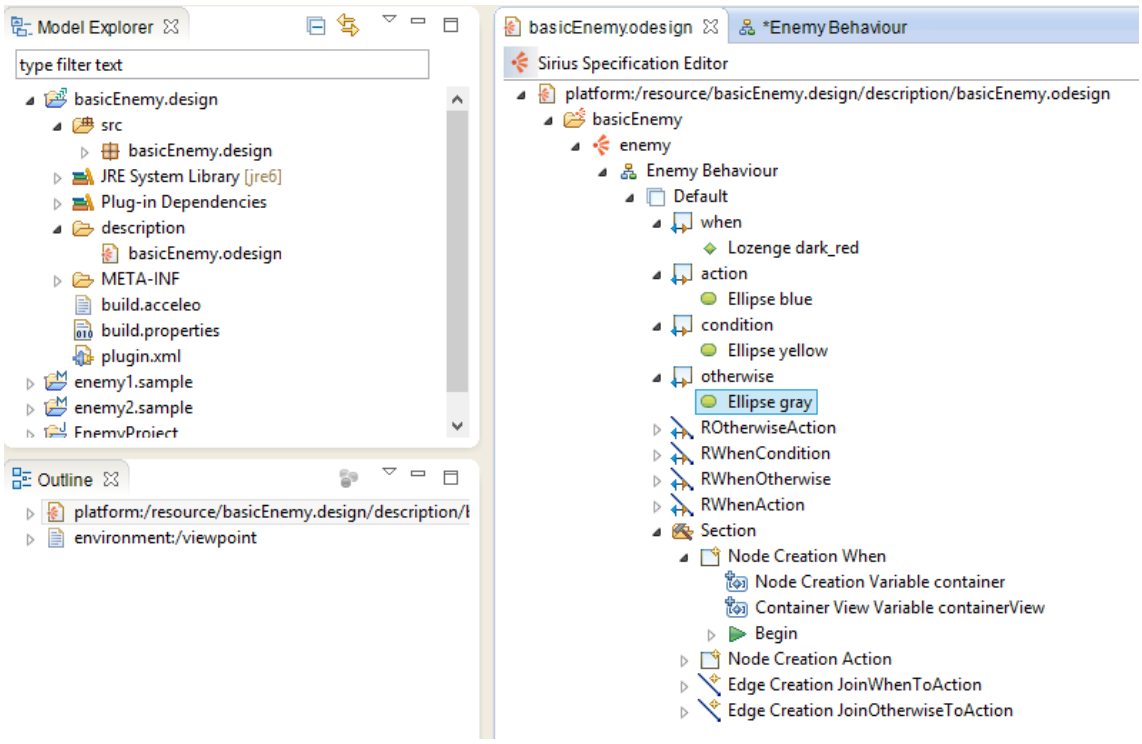


Fig. 25 Propiedades del Viewpoint en Sirius

[13] *Viewpoint*, en Sirius, define un conjunto de representaciones, que pueden ser diagramas, tablas, árboles, etc.

Una vez especificadas las diferentes propiedades, es hora de definir nuestro modelo a través de la herramienta.

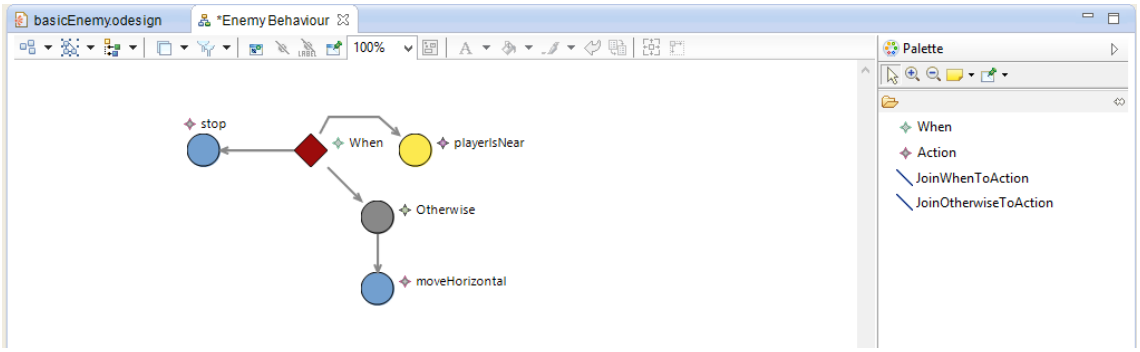


Fig. 26 Definiendo modelos en Sirius

Este editor generará una especificación XMI, que será luego el motor de transformación JET el encargado de gestionarla.

basicEnemy.odesign | Enemy Behaviour | enemy2.enemy

```
<?xml version="1.0" encoding="UTF-8"?>
<Enemy:Behaviour xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:Enemy="Enemy">
  <whens>
    <otherwise>
      <actions idAction="moveHorizontal"/>
    </otherwise>
    <condition idCondition="playerIsNear"/>
    <actions idAction="stop"/>
  </whens>
</Enemy:Behaviour>
```

Fig. 27 Especificación XMI generada a partir del editor Sirius

4.1.4 Importación al motor de transformación JET

Una vez hemos definido el modelo, es hora de cargarlo en el editor a través de la clase Java preparada para tal efecto. Esta clase se encargará de cargar el modelo en el proyecto, listo para ser tratado a posteriori por el motor de transformación.

La estructura que presenta el proyecto transformador de modelos es el siguiente:

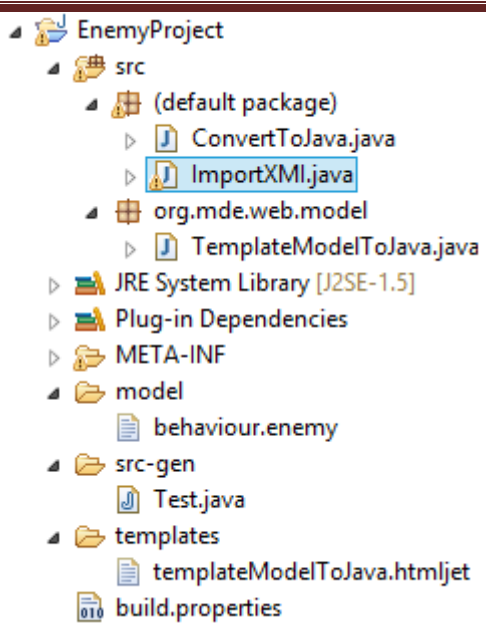


Fig. 28 Esqueleto del proyecto de transformación del modelo

Para importar el modelo que ya hemos desarrollado al motor de transformación, hacemos uso de la clase *ImportXMI.java*, cuyo código puede encontrarse a continuación.

```
public class ImportXMI {

    public static void main(String[] args) {
        //Initialize
        EnemyPackageImpl.init();

        //Cargar modelo XMI
        //Permitir que se deserialice el XMI con formato .enemy
        //Get Singleton Factory
        Factory.Registry.INSTANCE.getExtensionToFactoryMap().put("enemy",new XMIResourceFactoryImpl());

        //Un conjunto de recursos
        ResourceSet resourceSet = new ResourceSetImpl();

        //Obtener un recurso
        Resource resource = resourceSet.getResource(URI.createURI("model/behaviour.enemy"),true);

        Behaviour behaviour = (Behaviour)resource.getContents().get(0);
    }
}
```

Fig. 29 Código de la clase ImportXMI.java

4.1.5 Plantillas JET (Java Emitter Templates)

El motor de transformación JET es un motor M2T (Model to Text) basado en un modelo EMF. En JET, se definen una serie de plantillas, llamadas plantillas JET que a través del motor de transformación JET, permite la generación de código automático (Java, XML, etc.).

El generador de código es una parte fundamental en el desarrollo dirigido por modelos, ya que a partir de estas plantillas, se definen las implementaciones que se han de aplicar en las transformaciones.

En nuestra propuesta, la plantilla JET, llamada *templateModelToJava.htmljet*, está formada por el siguiente código:

```
<%@ jet package="org.mde.web.model" class="TemplateModelToJava" imports =
"com.enemy.Enemy.*" %>

<% Behaviour behaviour = (Behaviour)argument; %>

public void Update(long gameTime){
    if (state == EnemyState.Dying) return;
<%for(Action actionsBehaviour :
behaviour.getActions()){%>a_<%=actionsBehaviour.getIdAction().toString()%>();<%}%>
<%for(When when : behaviour.getWhens()){%>
    if(c_<%=when.getCondition().getIdCondition().toString()%>()){
        <%for(Action action :
when.getActions()){%>a_<%=action.getIdAction().toString()%>();<%}%>
    }<%if(when.getOtherwise() != null){%>
    else{<%for(Action actionOtherwise : when.getOtherwise().getActions()){%>
        a_<%=actionOtherwise.getIdAction().toString()%>();<%}%>
    }<%}%>
<%}%>

    //Always last execution. No configurable.
    execute(gameTime);
}
```

En esta plantilla, se recibe el modelo precargado en el motor de transformación JET que hayamos especificado en etapas anteriores. Este motor de transformación se encargará de traducirlo a código Java mediante una serie de pautas que se han establecido en esta plantilla.

Para lograr esta transformación, se ha creado la clase *ConvertToJava.java*, que es la encargada de recoger el modelo previamente cargado, y aplicarle la transformación de la plantilla JET, dando como resultado un archivo *Test.java*.

```
public class ConvertToJava {

    public static void main(String[] args) {
        EnemyPackageImpl.init();

        Factory.Registry.INSTANCE.getExtensionToFactoryMap().put("enemy",new XMIResourceFactoryImpl());

        ResourceSet resourceSet = new ResourceSetImpl();

        Resource resource = resourceSet.getResource(URI.createURI("model/behaviour.enemy"),true);
        Behaviour behaviour = (Behaviour)resource.getContents().get(0);

        TemplateModelToJava java = new TemplateModelToJava();

        try {
            FileWriter output = new FileWriter("src-gen/Test.java");

            BufferedWriter writer = new BufferedWriter(output);

            writer.write(java.generate(behaviour));

            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Fig. 30 Código de la clase ConvertToJava

Este archivo *Test.java* contiene el código generado por el motor de transformación que será el encargado de la parte de actualización del "*gameloop*[14]" dentro de la clase *EnemyShooter.java*

[14] El *gameloop* puede considerarse como el código central de nuestro videojuego, como si de una sala de control se tratara. Las partes más importantes de las que suele constar el gameloop son *inicializaciones*, *actualizaciones* y *dibujado*. Permite que un videojuego funcione sin problemas, independientemente si hay entrada o no por parte del usuario.

(proyecto android_platform_miw), encargada de establecer qué comportamientos y de qué manera se van a ejecutar durante el "*gameplay*".

Un ejemplo de generación de código a partir de la plantilla JET lo encontramos en el siguiente código:

```
public void Update(long gameTime){
    if (state == EnemyState.Dying) return;

    if(c_IsThereWall()){
        if(c_playerIsNear()){
            a_stop();

        }else{
            a_turnAround();
            a_moveHorizontal();
        }
    }

    //Always last execution. No configurable.
    execute(gameTime);
}
```

Si nos fijamos en el método anterior, observamos una serie de llamadas a métodos que son los que confrontan el conjunto del patrón de comportamiento que se va a aplicar. Esto se consigue además con la preparación del código destino a modo de "macros" intercambiables.

Estos códigos asociados a acciones y a condiciones pueden ser fácilmente añadidas por expertos en programación en aras de proveer más dinamismo y aumentar el número de acciones y condiciones finitas provistas para la generación de patrones de comportamiento.

Una vez establecido el patrón de comportamiento apropiado, ya sólo nos quedaría ejecutar el videojuego a través del proyecto *android_platform_miw* y ver el resultado obtenido, donde podemos observar que los comportamientos definidos se corresponden a los elegidos.

4.1.6 Proyecto *android_platform_miw*

El proyecto *android_platform_miw* es un proyecto de videojuego de plataformas 2D desarrollado por Ismael Posada Trobo. Es un proyecto que ha sido elegido para el despliegue de los patrones de comportamiento comentados en puntos anteriores.

Este proyecto está generado íntegramente en Android, desarrollado en el lenguaje de programación Java.

Este videojuego estará previamente creado a partir de una de las herramientas mencionadas en puntos anteriores como caso de éxito, como es "Gade4all" [12], de la que Ismael Posada Trobo formó parte de su desarrollo.

4.1.6.1 Android para despliegue del videojuego

Android es un sistema operativo móvil basado en Linux [49], destinado a una utilización en dispositivos móviles, como teléfonos inteligentes, tablets, Google TV, etc. Este sistema maneja, generalmente, aplicaciones como *Market* o su actualización, *Google Play* [50]. La estructura de Android está formada por aplicaciones que se ejecutan en un framework Java de aplicaciones

orientadas a objetos sobre el núcleo de las bibliotecas de Java en una máquina virtual *Dalvik* con compilación en tiempo de ejecución [51].

Fig. 31 Logotipo de Android

Las aplicaciones se desarrollan habitualmente en Java a través del Android Software Development Kit [52], aunque en los últimos años han florecido diferentes formas de creación de aplicaciones mediante otros lenguajes de programación, como son Dart [53], desarrollado por Google, o incluso C [54], mediante el NDK (Native Development Kit) en aras de conseguir mayor rendimiento de aplicaciones.

En nuestro caso, el videojuego destino está implementado en Java, y es parte del desarrollo de una herramienta desarrollada por el grupo de Investigación MDE-Research Group [55] de la universidad de Oviedo, al cual he pertenecido durante dos años. Esta herramienta es Gade4All.

Fig. 32 Logotipo de MDE-Research Group

4.1.7 Proyecto Gade4all

Gade4all [12] es un proyecto innovador destinado a la construcción de una plataforma para la creación de software interactivo. Consiste en una nueva plataforma que tiene como objetivo facilitar el desarrollo de videojuegos y software de entretenimiento a través del uso de lenguajes específicos de dominio (MDE).

Esta herramienta hace posible que los usuarios sin previos conocimientos en el campo de desarrollo de videojuegos 2D puedan crear videojuegos multiplataforma para dispositivos móviles de una forma simple e innovadora.

Fig. 33 Logotipo del proyecto Gade4all

Para nuestra solución, este proyecto ha sido el encargado de generar el videojuego de plataformas 2D que será usado para aplicarle los patrones de comportamiento anteriormente estudiados.

Un diagrama conceptual de nuestro proyecto puede verse en la siguiente figura, en el que se explica de forma detallada las diferentes etapas que adopta nuestra arquitectura.

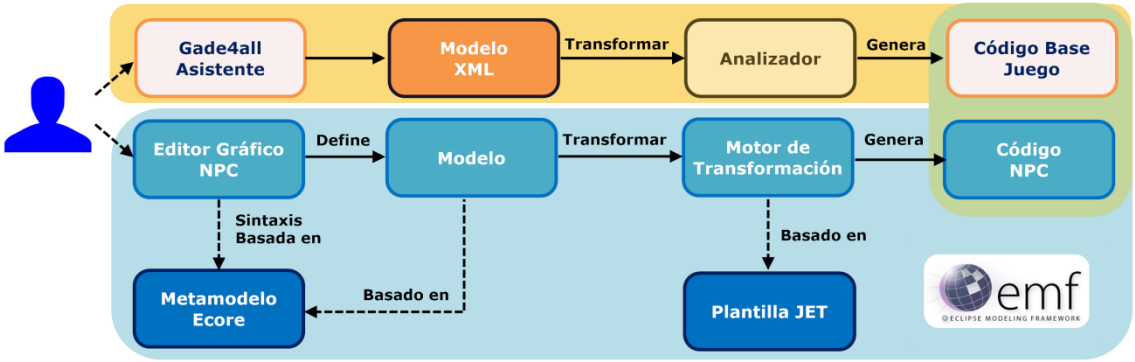


Fig. 34 Diseño conceptual de la arquitectura propuesta

Para finalizar, un esquema general de la solución puede verse en la siguiente imagen. En ella se muestran todas las etapas de las que consta nuestra solución:

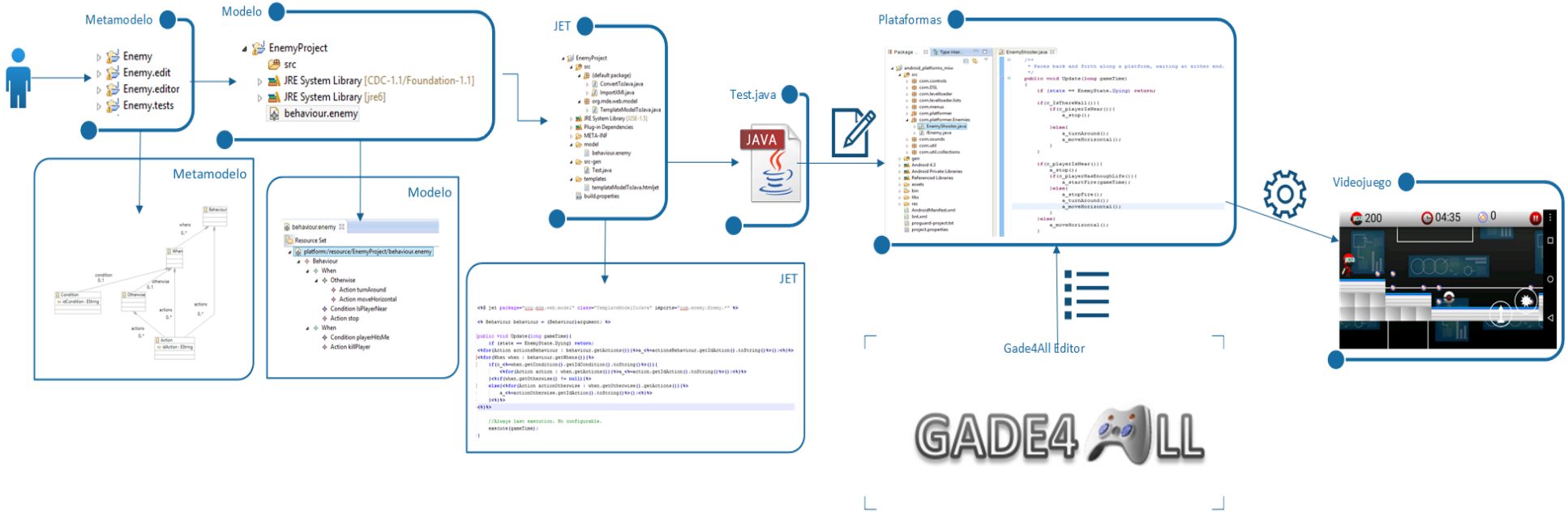


Fig. 35 Esquema general de nuestra solución

Capítulo 5. Uso de patrones de comportamiento: T2Game

A continuación se explicarán diferentes casos de uso a realizar con nuestra solución.

5.1 Introducción

El objetivo de este capítulo es servir de introducción y motivación sobre las ventajas del uso que nuestra solución aporta sin extenderse en detalles técnicos.

Para ello, vamos a desarrollar 4 enemigos con diferentes comportamientos contenidos en el videojuego de plataformas Mario Bros., de la compañía Nintendo, y en el videojuego Metal Slug de la compañía SNK, y uno al azar, respectivamente.

Para los dos primeros casos de uso, intentaremos reproducir dos comportamientos encontrados en el enemigo Goomba de videojuego Mario Bros. Éstos son "*Move & Turn Around*", y "*Player Hits Me*", que serán explicados a continuación.

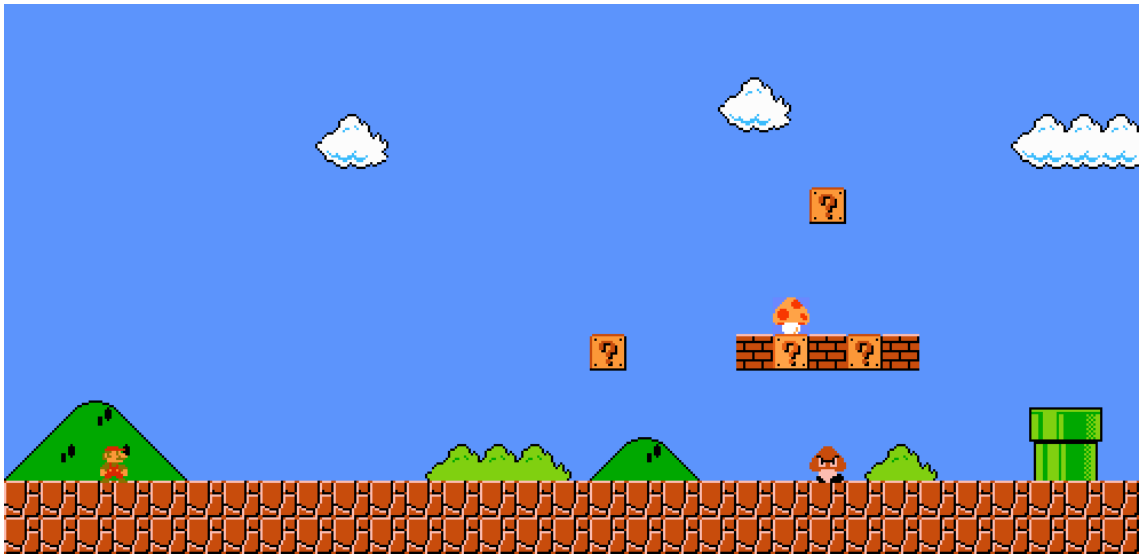


Fig. 36 Captura del videojuego Mario Bros, donde aparece Goomba

Para el caso 3, intentaremos reproducir uno de los comportamientos del soldado enemigo encontrado en el videojuego Metal Slug, mediante el comportamiento "*Shoot Player*".

Fig. 37 Captura del videojuego Metal Slug, con un soldado enemigo a la derecha

Para el caso de uso 4, que llamaremos "*Random Behavior*", vamos a experimentar un popurrí de comportamientos elegidos al azar dentro de las capacidades de nuestra solución.

Para la creación de comportamientos, damos por supuesto que el usuario ya parte de un proyecto creado (simplemente vacío), para poder focalizarnos expresamente en la definición de patrones de comportamiento. Nos centraremos exclusivamente en alimentar al modelo.

5.2 Caso de uso 1: Move & Turn Around

Para este caso, vamos a intentar replicar uno de los comportamientos que realiza el enemigo "*Goomba*" dentro del videojuego Mario Bros. A continuación se explica mediante un diagrama de flujo cual es el comportamiento que perseguimos.

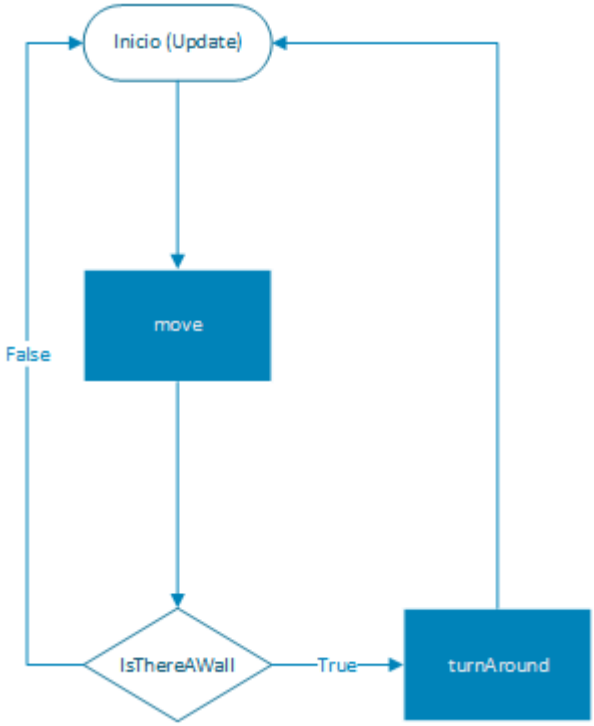


Fig. 38 Diagrama de flujo del comportamiento para el caso de uso 1

Para ello, añadimos un nodo Action, que corresponderá a la cajita “move”.

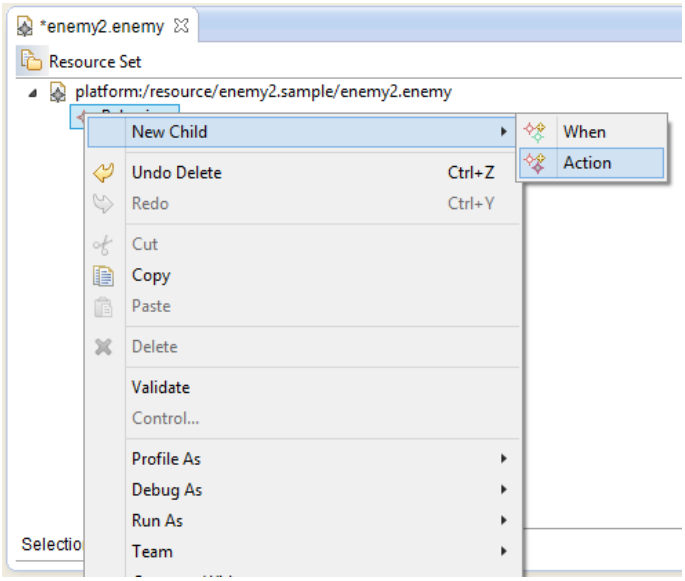


Fig. 39 Añadiendo nuevo nodo Action para el caso de uso 1

A continuación, situándonos sobre el nodo que acabamos de crear, establecemos en la ventana de propiedades el identificador oportuno, esto es, “moveHorizontal”.

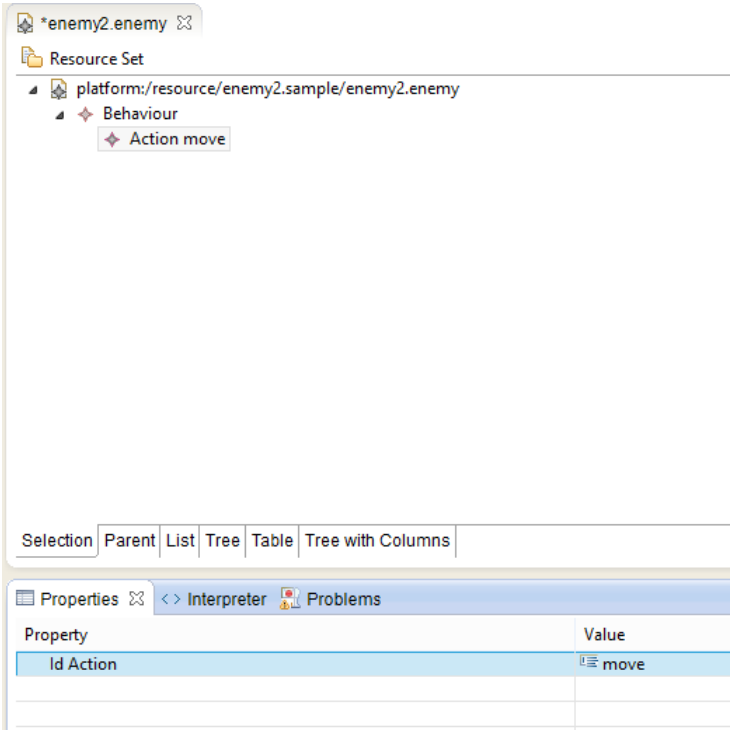


Fig. 40 Especificando propiedades de un nodo Action para el caso de uso 1

Ahora nos situamos sobre el nodo *Behaviour*, y hacemos la misma operación pero con un nodo *When*. Situándonos sobre el nodo *When*, añadimos un nuevo hijo llamado *Condition*, que es quien establecerá la condición:

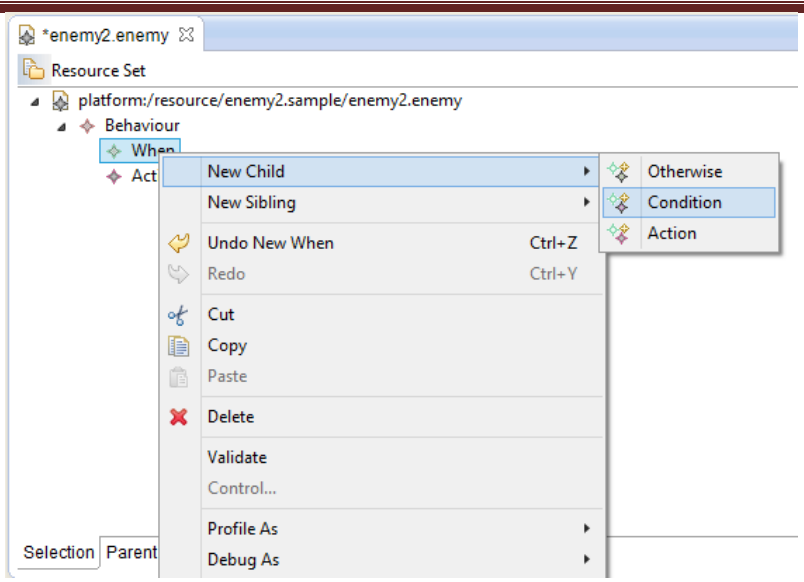


Fig. 41 Creando nuevo nodo Condition

Situándonos sobre el nodo *Condition*, establecemos en propiedades qué condición queremos evaluar. Para este caso, escribiremos "*IsThereWall*".

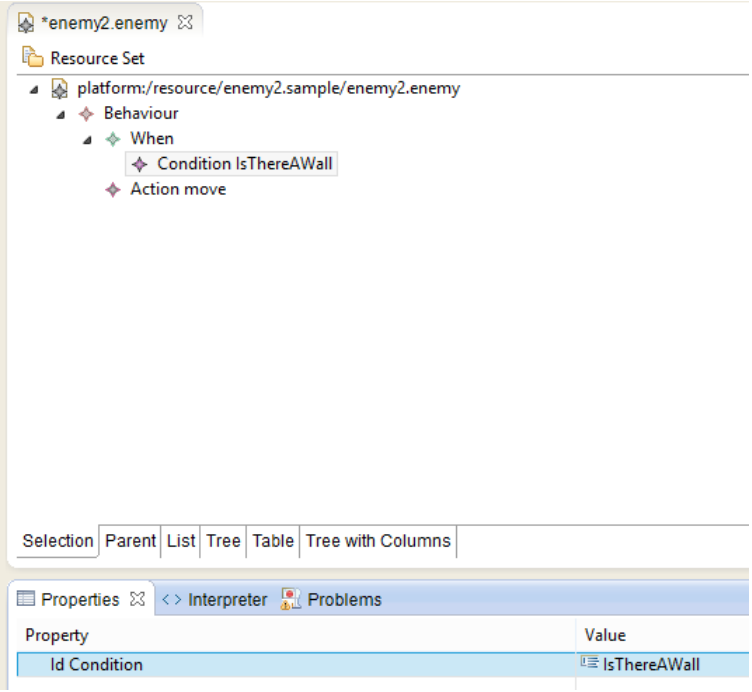


Fig. 42 Estableciendo propiedades en un nodo Condition

Ya tenemos la condición, por lo que ahora nos hace falta especificarle qué acción se ejecutará cuando se dé esa condición. Añadimos un nuevo nodo *Action* situándonos sobre el nodo *When*.

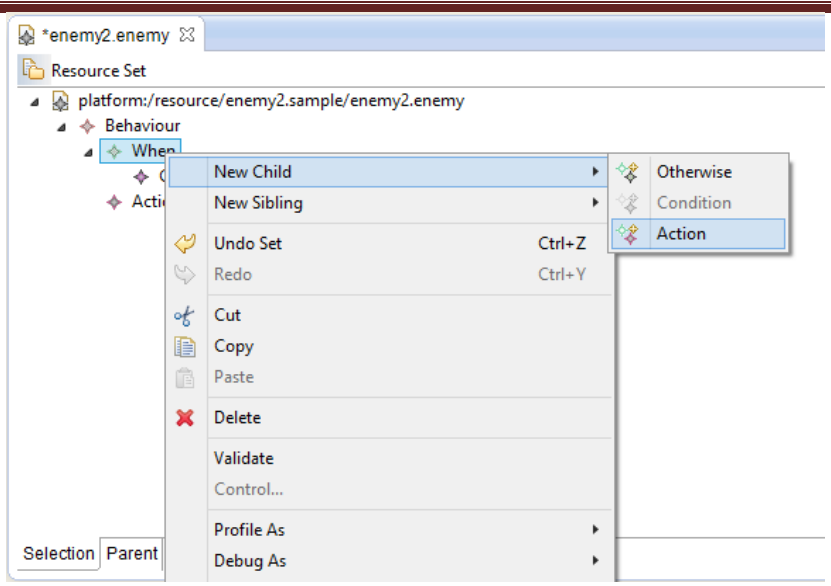


Fig. 43 Añadiendo nuevo nodo Action para el caso de uso 1

Al igual que hicimos anteriormente, estableceremos ahora el identificador de la *Action*. Para este caso, "*turnAround*".

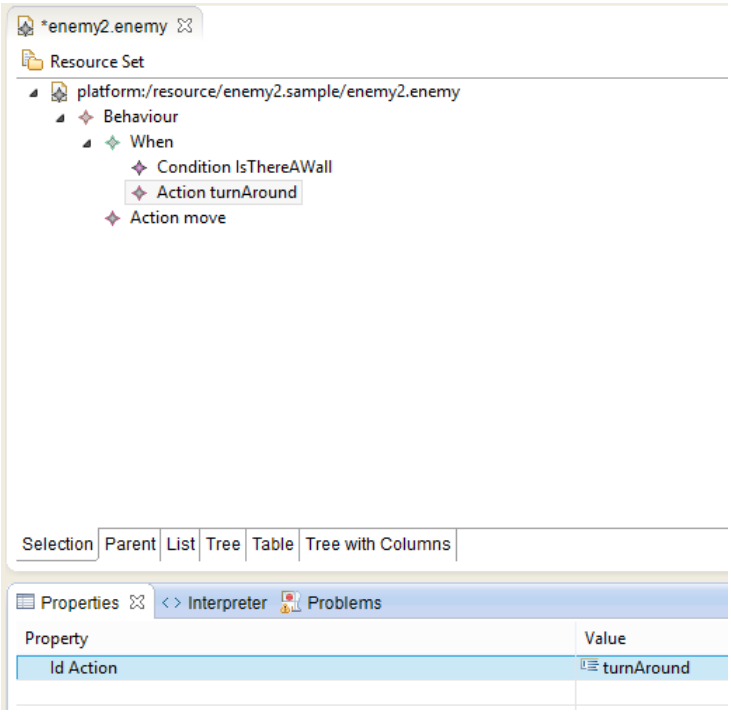


Fig. 44 Aspecto general del modelo para el caso de uso 1

Como curiosidad, podemos ver como se genera la especificación XMI, accediendo al modelo en modo texto.

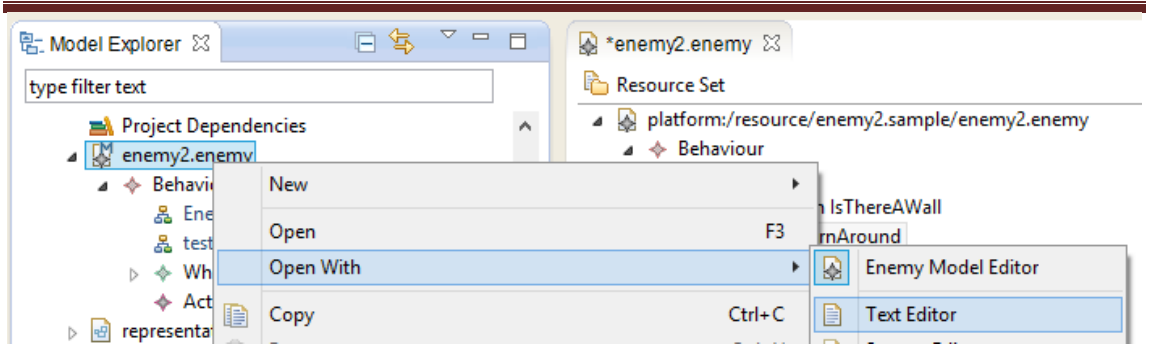


Fig. 45 Acceso al modelo en modo texto

Por otro lado, podemos hacerlo vía editor. Para ello agregaríamos un nodo *when*, que estará compuesto por un nodo *when*, un nodo *condition* y un nodo *otherwise*. A posteriori, agregaríamos las acciones y las relacionaríamos con el nodo correcto. Obtendríamos el siguiente resultado:

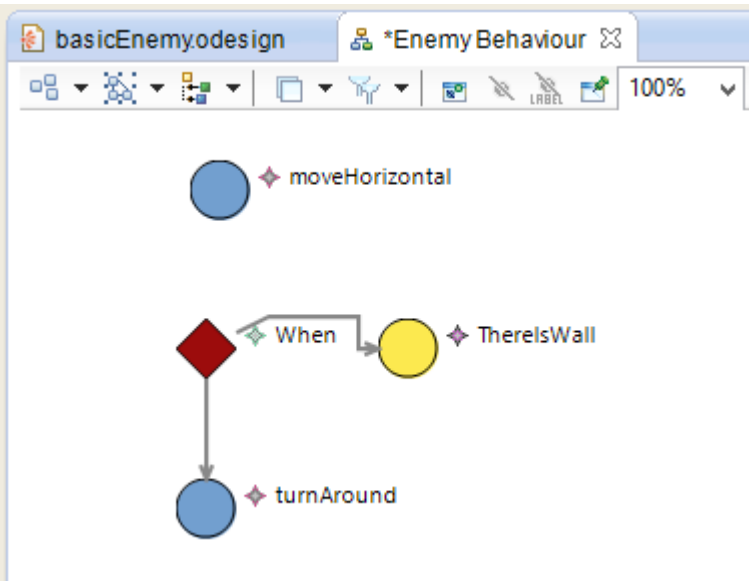


Fig. 46 Usando el editor para el caso de uso 1

Y nos aparecerá lo siguiente:

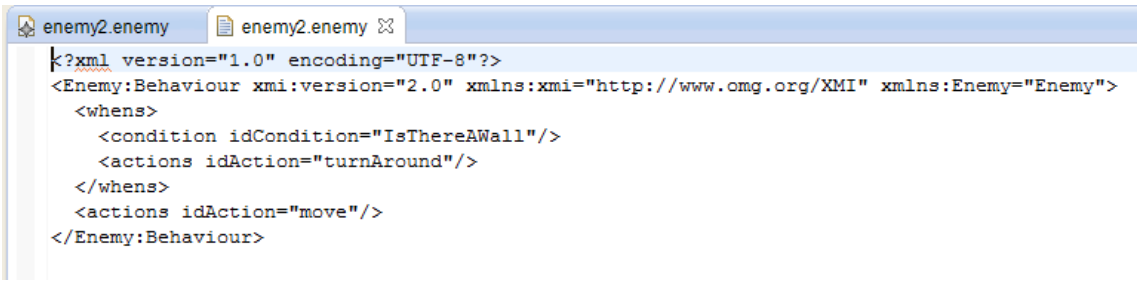


Fig. 47 Especificación XMI para el caso de uso 1

Llegados a este punto, la creación de un patrón de comportamiento estaría finalizada, pues las operaciones que vienen a continuación son de visualización sobre el proyecto *android_platform_miw*. Para llegar a ello y al tratarse de un prototipo, hay varias operaciones

que han de hacerse manualmente, que explicaremos a continuación en aras de conseguir un funcionamiento completo.

5.2.1 Desplegando el patrón de comportamiento

Con el modelo ya desarrollado, pasamos pues a la parte de transformación. Copiamos el modelo (en este caso, *enemy2.enemy*), y lo pegamos en la carpeta *model* del proyecto *EnemyProject*

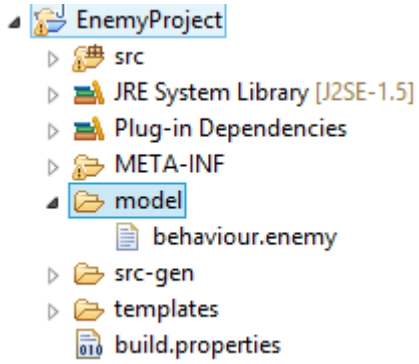


Fig. 48 Aspecto general del proyecto EnemyProject, conteniendo el modelo

A continuación, vamos a la carpeta *src*, y ejecutamos la clase *ImportXMI.java*. Esto cargará el modelo. A posteriori, ejecutamos la clase *ConvertToJava.java*, que nos generará un archivo *Test.java* en la carpeta *src-gen*.

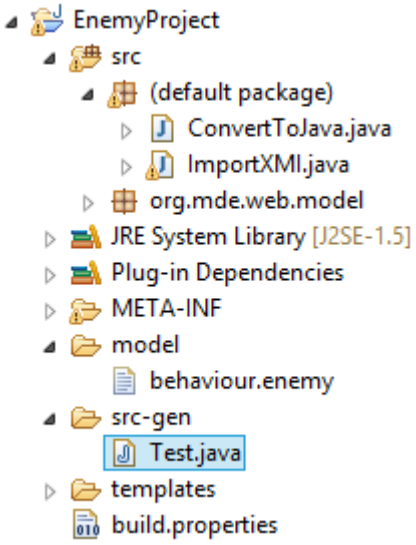


Fig. 49 Contenido del proyecto "motor de transformación JET"

Abrimos el archivo *Test.java*, y vemos el código Java que se ha generado a partir del modelo que hemos desarrollado.

*Test.java | ImportXMI.java | templateModelToJava.htmljet

```
public void Update(long gameTime){
    if (state == EnemyState.Dying) return;

    a_moveHorizontal();
    if(c_IsThereAWall()){
        a_turnAround();
    }

    //Always last execution. No configurable.
    execute(gameTime);
}
```

Fig. 50 Ejemplo de generación en la el fichero Test.java

Ya con el código generado, es tiempo de establecérselo al proyecto *android_platform_miw* en la clase *Enemy.java*. Copiamos el método *Update* de la clase *Test.java* generado y lo pegamos en la clase *EnemyShooter.java*.

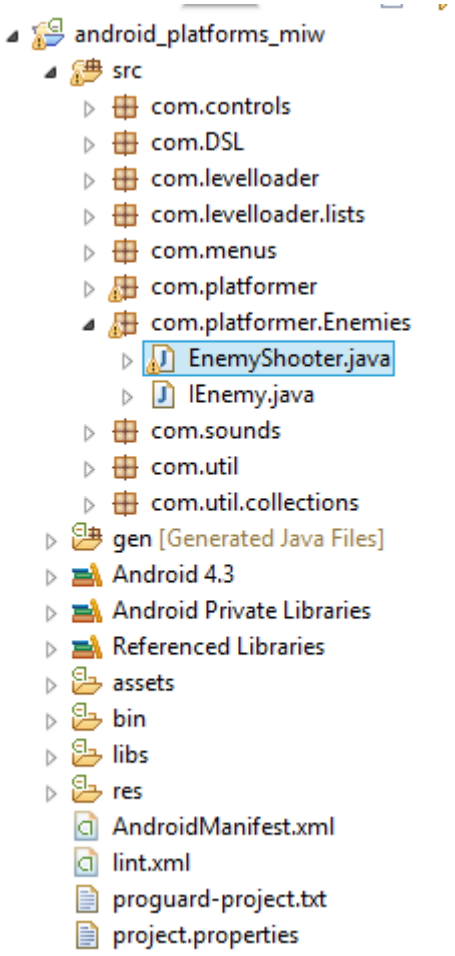


Fig. 51 Árbol de ficheros conteniendo la clase EnemyShooter.java

Ejecutamos y comprobamos que efectivamente, los enemigos contenidos en el videojuego han adquirido el patrón de comportamiento que hemos establecido anteriormente.

5.3 Caso de Uso 2: Player Hits Me

Para el caso 2, vamos a replicar el comportamiento del enemigo *Goomba* del videojuego Mario Bros, en el cual este enemigo si colisiona contra Mario Bros, Mario Bros muere. Para este caso, vamos a combinarlo con el caso de uso 1, uniendo ambos comportamientos, dejando al enemigo que dada una situación, actúe según las reglas dadas. El diagrama de flujo que representa este comportamiento es el siguiente:

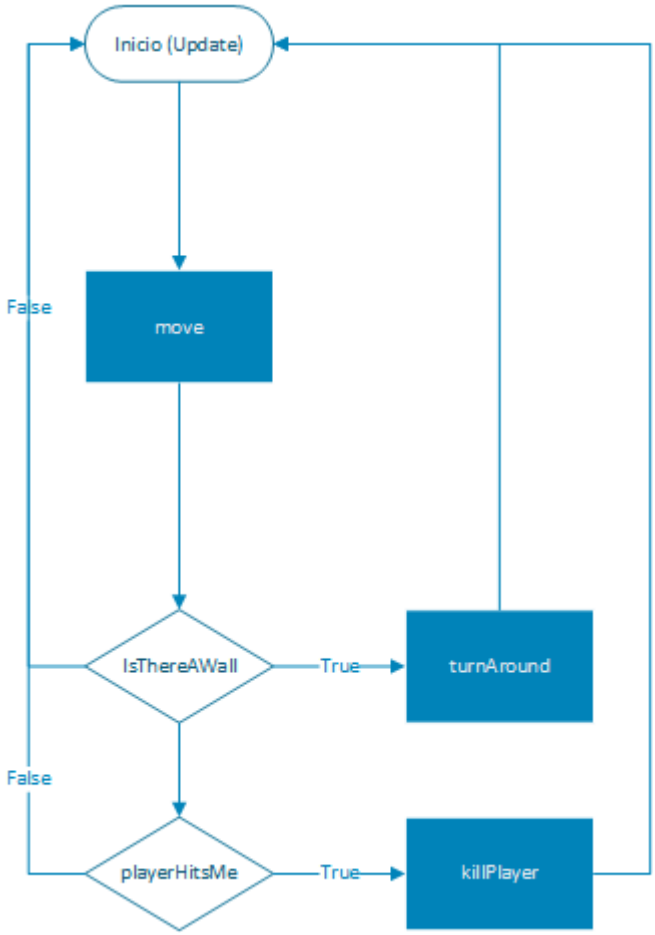


Fig. 52 Diagrama de flujo del caso de uso 2, Player Hits Me

Reaprovechando el modelo que hemos desarrollado anteriormente, vamos a añadirle una nueva condición y una nueva acción que irán asociadas al nuevo comportamiento.

Partiendo de lo que ya teníamos anteriormente, nos situamos sobre el nodo principal *Behaviour*, y añadimos un nuevo nodo *When*. A posteriori, nos situamos sobre el nodo *When* creado y añadimos un nuevo nodo *Condition*, en el que especificaremos como propiedad "*playerHitsMe*"

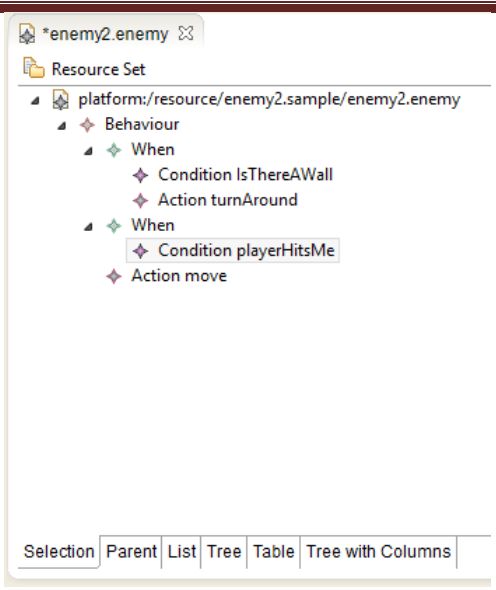


Fig. 53 Estableciendo propiedades en el caso de uso 2

Ahora, es turno de establecer qué acción se ejecutará tras producirse la condición anterior. En este caso, añadimos un nuevo nodo *Action* cuya propiedad será "*killPlayer*".

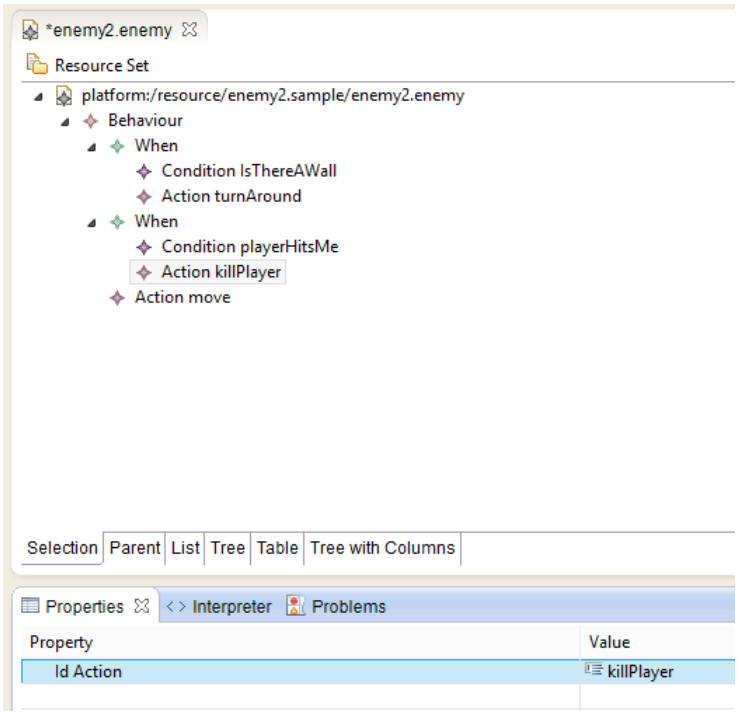


Fig. 54 Aspecto general del modelo en el caso de uso 2

Estos mismos pasos anteriores pueden realizarse mediante el editor, lo que nos daría como resultado el siguiente diagrama:

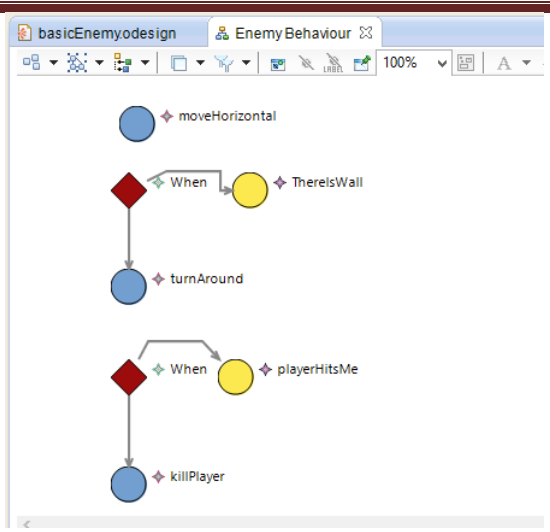


Fig. 55 Usando el editor para el caso de uso 2

Al igual que antes, podemos observar la especificación XMI generada para este comportamiento.

enemy2.enemy enemy2.enemy

```
<?xml version="1.0" encoding="UTF-8"?>
<Enemy:Behaviour xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:Enemy="Enemy">
  <whens>
    <condition idCondition="IsThereAWall"/>
    <actions idAction="turnAround"/>
  </whens>
  <whens>
    <condition idCondition="playerHitsMe"/>
    <actions idAction="killPlayer"/>
  </whens>
  <actions idAction="move"/>
</Enemy:Behaviour>
```

Fig. 56 Especificación XMI para el caso de uso 2

Ya sólo nos faltaría seguir los pasos de despliegue del patrón de comportamiento, contemplados en el punto 5.2.1 y comprobar el comportamiento que adquieren los enemigos para este patrón dado.

5.4 Caso de uso 3: Shoot Player

Para el caso de uso 3, vamos a replicar el comportamiento de un enemigo soldado del videojuego Metal Slug, en el cual si el enemigo está a una distancia considerable del personaje principal, le disparará.

Para este caso, vamos a combinarlo con el caso de uso 2, uniendo ambos comportamientos, dejando al enemigo que dada una situación, actúe según las reglas dadas. El diagrama de flujo que representa este comportamiento es el siguiente:

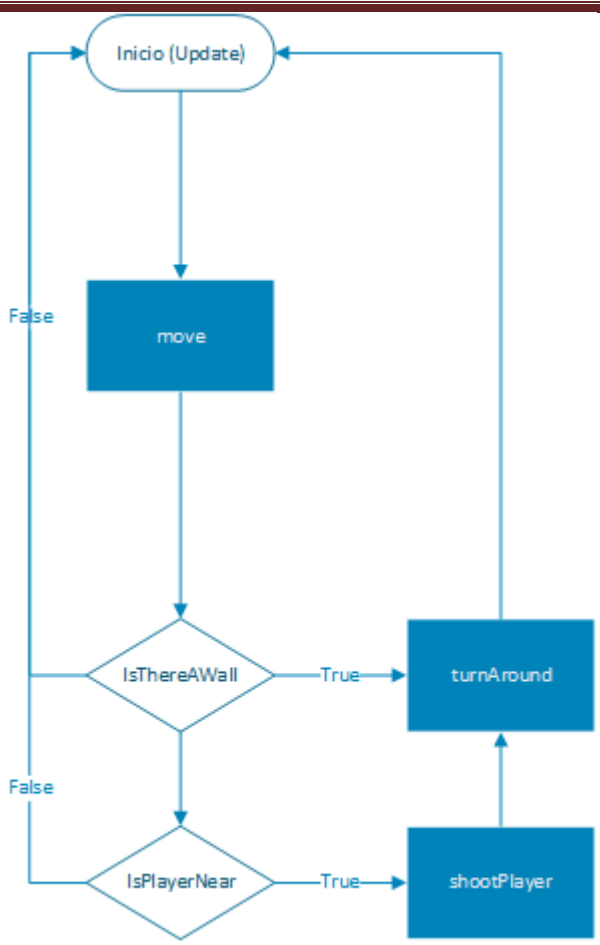


Fig. 57 Diagrama de flujo para el caso de uso 3, Shoot Player

Podríamos crear un comportamiento desde cero, pero en este caso vamos a modificar un comportamiento ya existente con el fin de averiguar cuán complejo es editar y cambiar un comportamiento ya definido.

Partiendo del patrón de comportamiento del caso de uso 2, cambiaremos la condición en la que aparece "*playerHitsMe*" y estableceremos "*IsPlayerNear*".

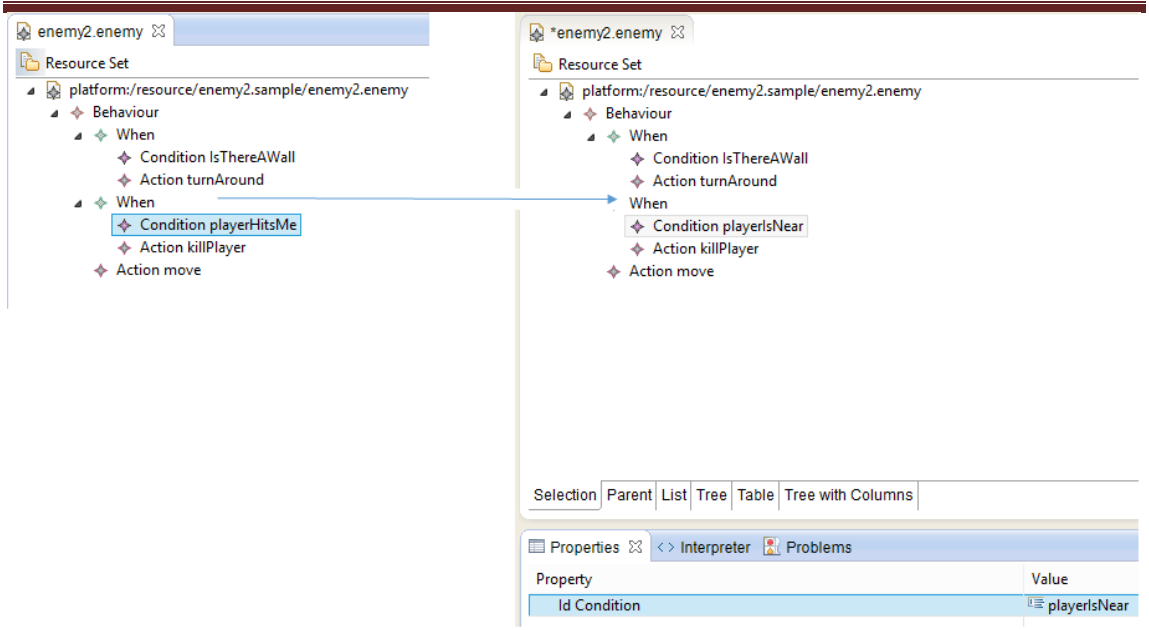


Fig. 58 Creando acciones en el modelo

En el editor, sería algo tan sencillo como situarnos sobre el nodo *Condition* y cambiar el valor de la propiedad:

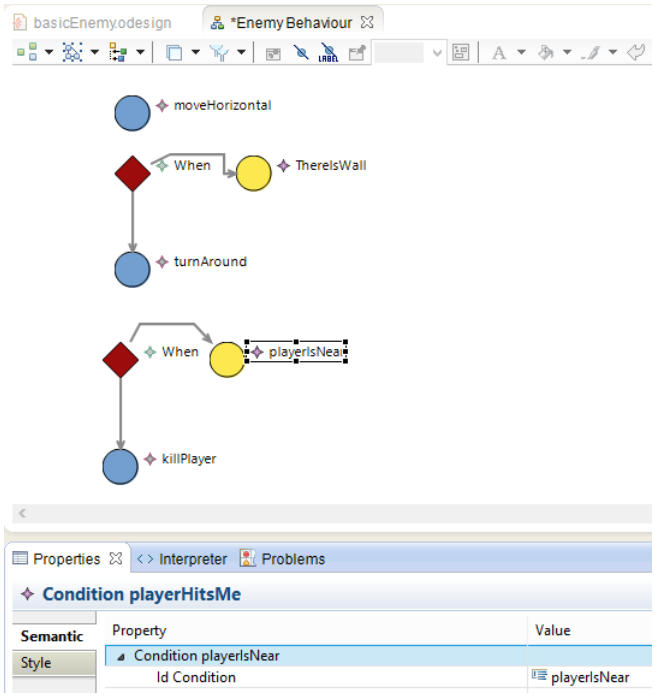


Fig. 59 Creando acciones en el modelo mediante el editor

Hacemos lo propio con la acción "*killPlayer*", cambiándola por "*shootPlayer*".

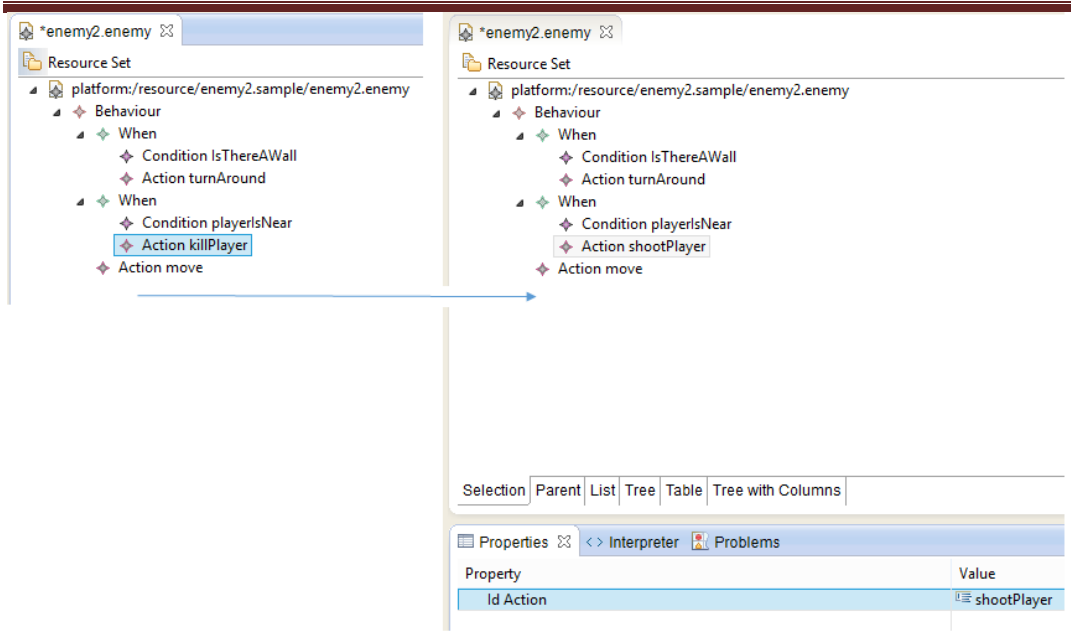


Fig. 60 Creando acciones en el modelo, asignando propiedades

Y por último añadimos la acción que nos falta para completar el patrón de comportamiento específico de un soldado dentro del videojuego Metal Slug.

Para ello, nos situamos sobre el último nodo *When*, y añadimos un nuevo nodo *Action*, cuya propiedad será "turnAround".

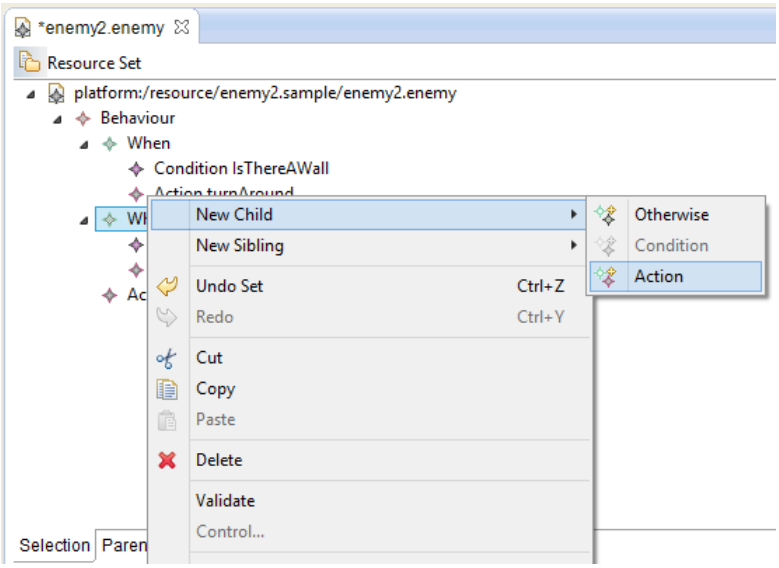


Fig. 61 Añadiendo nuevo nodo Action

En el editor, habrá que añadir un nodo *Action* y unirla con el nodo correspondiente *When*:

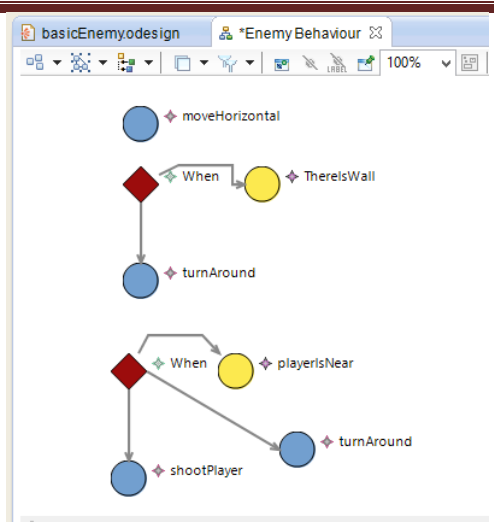


Fig. 62 Añadiendo nuevo nodo Action en el editor

Con esto, ya tenemos listo nuestro patrón de comportamiento, esta vez con la particularidad de, partiendo de un patrón de comportamiento dado, cambiar las acciones y el flujo que adquirirá a lo largo de su ejecución.

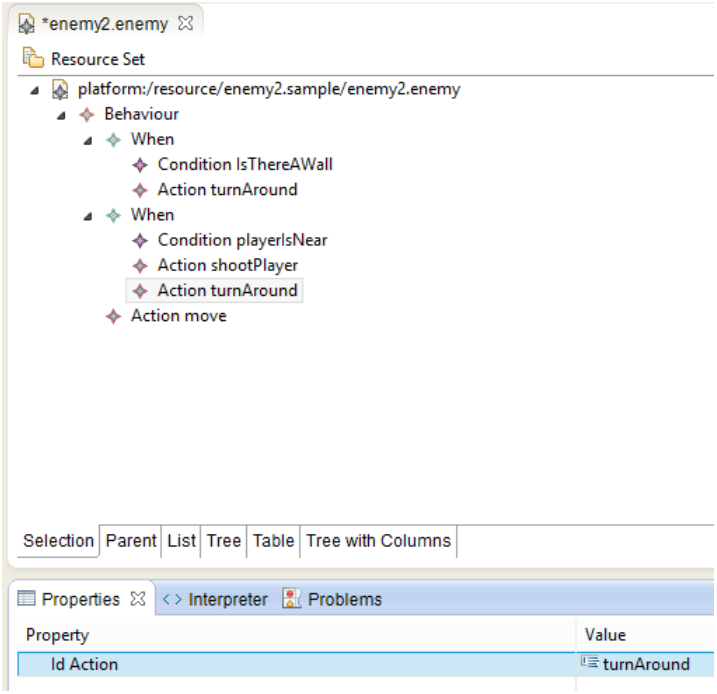


Fig. 63 Aspecto que presenta el modelo en el caso de uso 3

Comprobamos la especificación XMI resultante y ejecutamos siguiendo los pasos especificados en el punto 5.2.1, para comprobar que realmente este comportamiento ha cambiado.

enemy2.enemy | enemy2.enemy

```
<?xml version="1.0" encoding="UTF-8"?>
<Enemy:Behaviour xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:Enemy="Enemy">
  <whens>
    <condition idCondition="IsThereAWall"/>
    <actions idAction="turnAround"/>
  </whens>
  <whens>
    <condition idCondition="playerIsNear"/>
    <actions idAction="shootPlayer"/>
    <actions idAction="turnAround"/>
  </whens>
  <actions idAction="move"/>
</Enemy:Behaviour>
```

Fig. 64 Especificación XMI para el caso de uso 3

5.5 Caso de uso 4: Random Behavior

Para este caso de uso especial, vamos a incorporar varios elementos de la creación de patrones de comportamiento, sin un objetivo definido. Es decir, el comportamiento elegido es totalmente aleatorio, y puede no tener sentido en un videojuego real. El objetivo pues es comprobar las capacidades y el número de pasos que deberíamos realizar para alcanzar dicho patrón de comportamiento.

El diagrama de flujo que acompaña a este comportamiento es el siguiente:

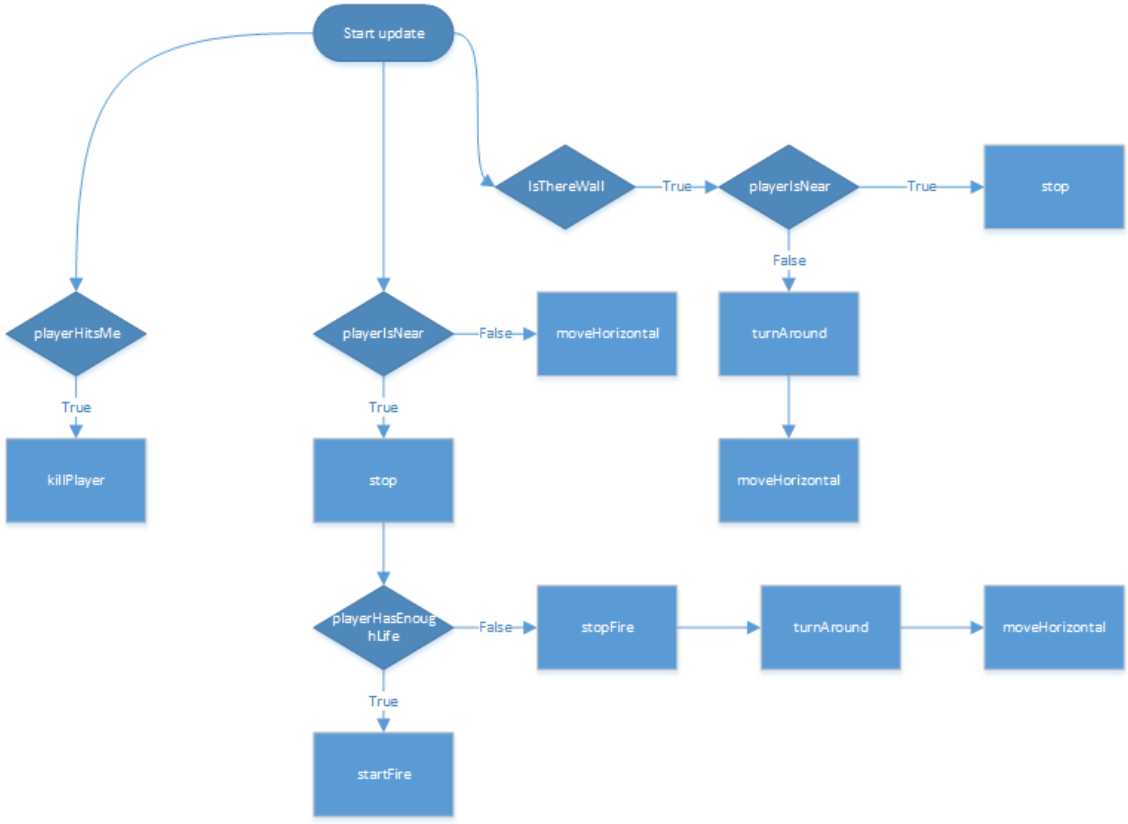


Fig. 65 Diagrama de flujo para el caso de uso 4, "Random Behavior"

Este diagrama daría como resultado el siguiente modelo.

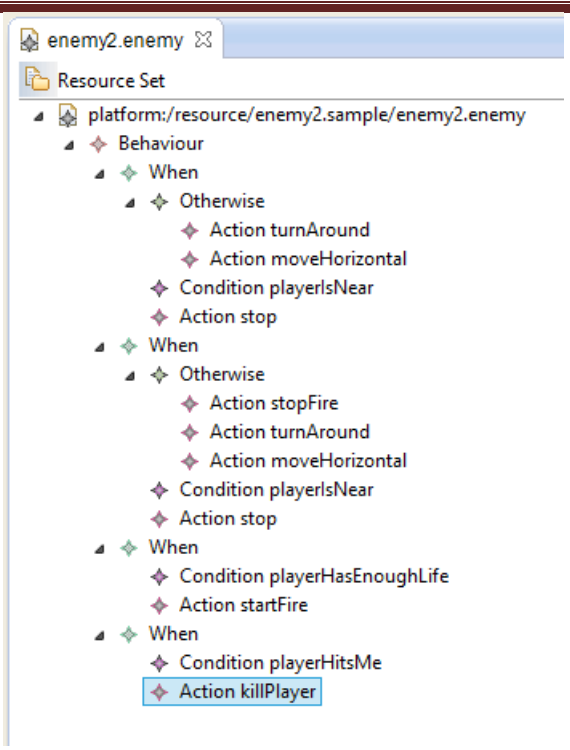


Fig. 66 Modelo para el caso de uso 4, "Random Behavior"

Usando el editor:

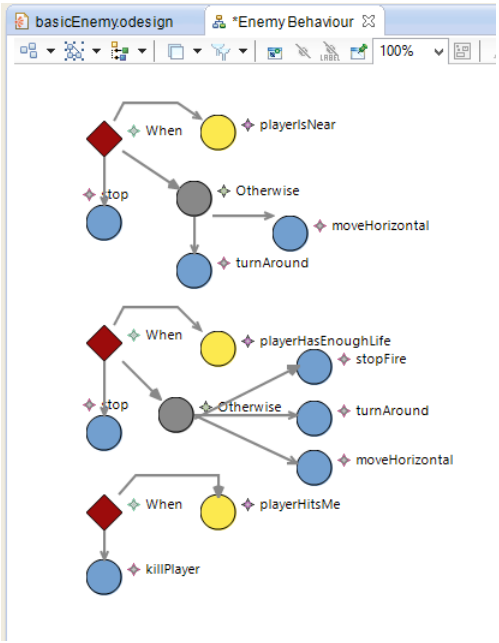


Fig. 67 Usando el editor para el caso de uso 4

El cual generará la siguiente especificación XMI.

enemy2.enemy enemy2.enemy

```
<?xml version="1.0" encoding="UTF-8"?>
<Enemy:Behaviour xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:Enemy="Enemy">
  <whens>
    <otherwise>
      <actions idAction="turnAround"/>
      <actions idAction="moveHorizontal"/>
    </otherwise>
    <condition idCondition="playerIsNear"/>
    <actions idAction="stop"/>
  </whens>
  <whens>
    <otherwise>
      <actions idAction="stopFire"/>
      <actions idAction="turnAround"/>
      <actions idAction="moveHorizontal"/>
    </otherwise>
    <condition idCondition="playerIsNear"/>
    <actions idAction="stop"/>
  </whens>
  <whens>
    <condition idCondition="playerHasEnoughLife"/>
    <actions idAction="startFire"/>
  </whens>
  <whens>
    <condition idCondition="playerHitsMe"/>
    <actions idAction="killPlayer"/>
  </whens>
</Enemy:Behaviour>
```

Fig. 68 Especificación XMI para el caso de uso 4, "Random Behavior"

Capítulo 6. Evaluación

Una vez hemos especificado cuál es el objetivo que perseguimos con nuestra propuesta y habiendo explicado detenidamente el funcionamiento de la misma, es hora de evaluar nuestra propuesta con respecto a las alternativas vigentes hoy en el mercado de creación de patrones de comportamiento.

Para realizar la comparación, vamos a seleccionar una lógica de negocio concreta, especificación de comportamiento para un enemigo. Los escenarios que se van a comparar son los siguientes:

1. Creación del comportamiento de un enemigo dentro de un videojuego de taxonomía plataformas obtenido del entorno de desarrollo *Game Maker Studio*, *Stencyl*, *Unity* 3D y *Blueprints*. El comportamiento elegido es la comprobación de que si un enemigo choca contra nuestro jugador, inmediatamente nuestro jugador es destruido (Player Hits Me).
2. Creación de un comportamiento totalmente aleatorio de un enemigo, exprimiendo las capacidades de nuestra solución (*Random Behavior*).
3. Posible alternativa al sistema de definición de patrones de comportamiento a partir de nuestra solución.

Nota: damos por supuesto que la parte gráfica ya ha sido abordada, y lo único que resta es la especificación del comportamiento del enemigo. Los personajes están definidos y presentes en el escenario.

6.1 Situación actual: comportamiento “Player Hits Me”

Actualmente, para la especificación del comportamiento de un enemigo para el punto 1 anterior hace falta lo siguiente:

6.1.1 Caso Game Maker Studio

Establecemos como ID del enemigo obj_Enemy dentro de las propiedades del enemigo. A continuación, deberíamos escribir el siguiente fragmento de código:

```
var inst;
inst = instance_place(x, y, obj_Enemy);
if inst != noone
{
with (inst) instance_destroy();
}
```

A posteriori, ejecutaremos el videojuego para ver su resultado.

6.1.2 Caso Stencyl

Una vez tenemos definidos los enemigos, tenemos que crear un nuevo grupo que llamaremos Enemies. Después, en la parte de eventos, crearemos un evento básico donde le especificaremos qué actores están comprometidos en ese evento.

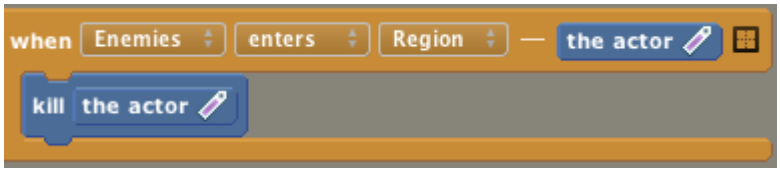


Fig. 69 "Player Hits Me" realizado con la herramienta Stencyl

A posteriori, ejecutaremos el videojuego para ver su resultado.

6.1.3 Caso Unity 3D

Ya establecidos todos los elementos que forman nuestro videojuego, con los tags oportunos que identifican a cada uno de los grupos de actores que intervienen en la escena, accedemos al editor de scripts e insertamos el siguiente código:

```
var health = 200;
var damage = 200;
var wait_time = 2;
var lock = 0;
function OnCollisionEnter(hit: Collision){
     if(hit.gameObject.tag == "enemy" && lock == 0){
        KillPlayer();
     }
}

function KillPlayer(){
    lock=1;
    health=health-damage;
    yield WaitForSeconds(wait_time);
    lock=0;
}

function Update(){
    if(health == 0){
       Destroy(gameObject);
    }
}
```

Finalizaremos ejecutando la escena y comprobando que realmente el comportamiento es el esperado.

6.1.4 Caso Blueprints

Al igual que las anteriores alternativas, con todos los actores establecidos en la escena, es hora de especificar el patrón de comportamiento "Player Hits Me". Para ello, accedemos a la parte Blueprints del personaje principal, y establecemos la siguiente relación. Creamos una especie de trigger que cuando detecte que algo ha tocado la cabeza del personaje, éste se muere.

Para ello, hacemos uso del evento "*Event Point Damage*", estableciendo la variable "*Health*" como -1. De esta manera, siempre que algo choque contra la cabeza del personaje principal, la vida del personaje pasará a ser -1, para finalmente añadir la función "*Destroy Actor*" que hará el resto.

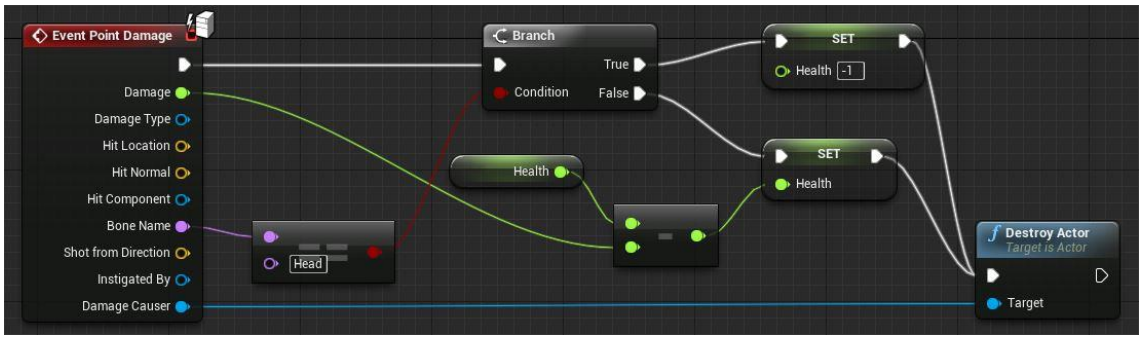


Fig. 70 "Player Hits Me" realizado con la herramienta Blueprints

6.1.5 Solución propuesta

Para la definición del patrón de comportamiento en el sistema propuesto hace falta:

Accedemos al modelo y creamos el patrón de comportamiento que deseemos, especificando los identificadores propios para cada entidad.

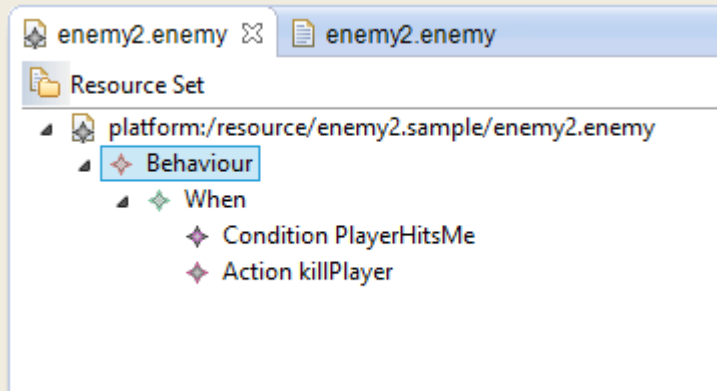


Fig. 71 Modelo para el caso de uso "Player Hits Me"

A través del editor obtendríamos el siguiente diagrama:

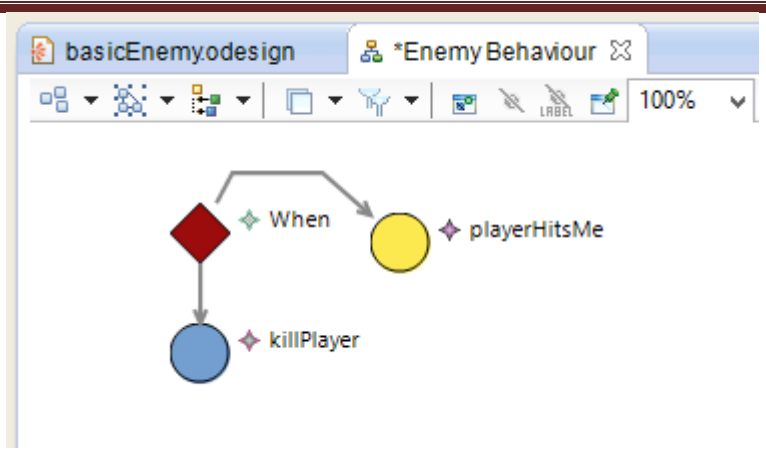


Fig. 72 Creando el caso de uso 1 "Player Hits Me" a través del editor

Una vez establecido el modelo, convertimos nuestro modelo en código ejecutable y entendible por la aplicación destino, en nuestro caso el videojuego de plataformas.

Al tratarse de un prototipo, hay operaciones intermedias que se deben hacer a mano. Obviamente en una versión comercial estas operaciones estarán ausentes y el proceso será automático, pasando directamente del modelo a código ejecutable y entendible por la aplicación destino.

Ya por último ejecutamos el videojuego para comprobar que realmente es el resultado esperado.

6.2 Situación actual: comportamiento "Random Behavior"

Para la creación de un comportamiento relativo al punto 2 anterior, tomamos como ejemplo los procedimientos llevados a cabo en el punto 6.1 para las diferentes herramientas.

Para este caso de uso hemos hecho una estimación basándonos en otras soluciones relativas al modelaje de comportamientos.

6.3 Interpretación de los resultados

A la hora de evaluar las alternativas, no está claro de qué forma hemos de hacerlo pues son muchos los factores que influyen en la edición de este tipo de patrones de comportamiento.

Para ello, vamos a proponer un sistema de evaluación que aglutina aquellos aspectos que creemos que influyen en la complejidad a la hora de realizar alguna operación dentro de la definición de patrones de comportamiento.

Tenemos 5 elementos claramente diferenciados que forman parte del proceso de construcción de patrones, que son:

- Caracteres escritos para llegar a un fin.
- Clases usadas/creadas para su consecución
- Método/s que provee el framework o solución en aras de ayudar a la consecución del patrón de comportamiento.
- Nodos o cajas usadas para la elaboración.
- Propiedades que se ven afectadas y/o han sido creadas para llegar a un fin.

Así mismo, hemos asignado unos pesos a cada uno de los elementos anteriores, pues el grado de dificultad de usar un elemento u otro es plausible.

A continuación se puede ver los resultados obtenidos para la medición del caso de uso "Player Hits Me":

	Caracteres	Clases	Métodos Framework	Nodos	Propiedades
Unity	233	1	3	0	2
Game Maker	71	1	1	0	2
Stencyl	15	0	0	4	5
T2Game	22	0	0	2	2
Blueprints	36	0	0	7	6

Fig. 73 Tabla de mediciones para el caso de uso "Player Hits Me" sin pesos asignados

La tabla resultante, con los pesos asignados, se muestra a continuación. El total representa la suma de los valores, pues se interpreta como la suma de las operaciones que se han de hacer para lograr un fin:

	Caracteres (0,3)	Clases (0,2)	Métodos Framework (0,3)	Nodos (0,4)	Propiedades (0,6)	Total
Unity	69,9	0,2	0,9	0	1,2	72,2
Game Maker	21,3	0,2	0,3	0	1,2	23
Stencyl	4,5	0	0	1,6	3	9,1
T2Game	6,6	0	0	0,8	1,2	8,6
Blueprints	10,8	0	0	2,8	3,6	17,2

Fig. 74 Tabla de mediciones para el caso de uso "Player Hits Me" con pesos asignados

El gráfico resultante de la comparación es el siguiente (en escala logarítmica):

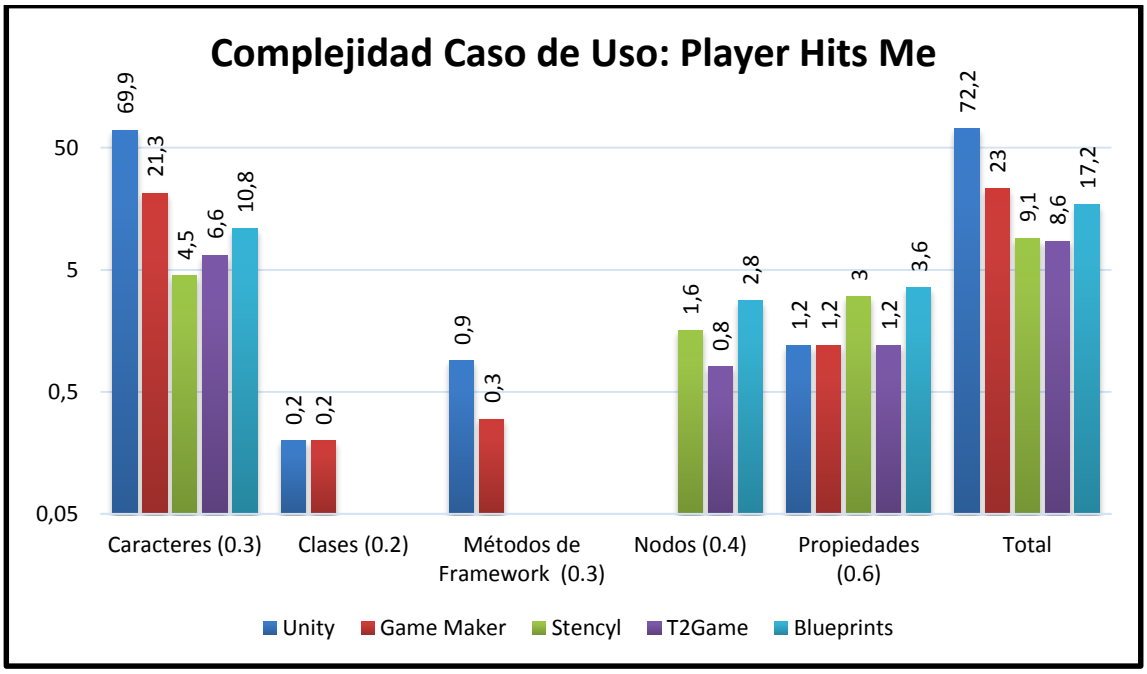


Fig. 75 Gráfico de comparación de alternativas para el caso de uso "Player Hits Me"

Para el caso de uso "*Random Behavior*", se muestran los siguientes resultados:

	Caracteres	Clases	Métodos Framework	Nodos	Propiedades
Unity	5495	1	8	0	8
Game Maker	1717	1	5	0	7
Stencyl	182	0	0	21	21
T2Game	165	0	0	16	15
Blueprints	272	0	0	28	20

Fig. 76 Tabla de mediciones correspondiente al caso de uso "Random Behavior"

A continuación se muestra la correspondiente tabla con los pesos asignados para cada operación llevada a cabo para completar la creación del comportamiento.

	Caracteres (0,3)	Clases (0,2)	Métodos Framework (0,3)	Nodos (0,4)	Propiedades (0,6)	Total
Unity	1648,5	0,2	2,4	0	4,8	1655,9
Game Maker	515,1	0,2	1,5	0	4,2	521
Stencyl	54,6	0	0	8,4	12,6	75,6
T2Game	49,5	0	0	6,4	9	64,9
Blueprints	81,6	0	0	11,2	12	104,8

Fig. 77 Tabla de mediciones con pesos asignados correspondiente al caso de uso "Random Behavior"

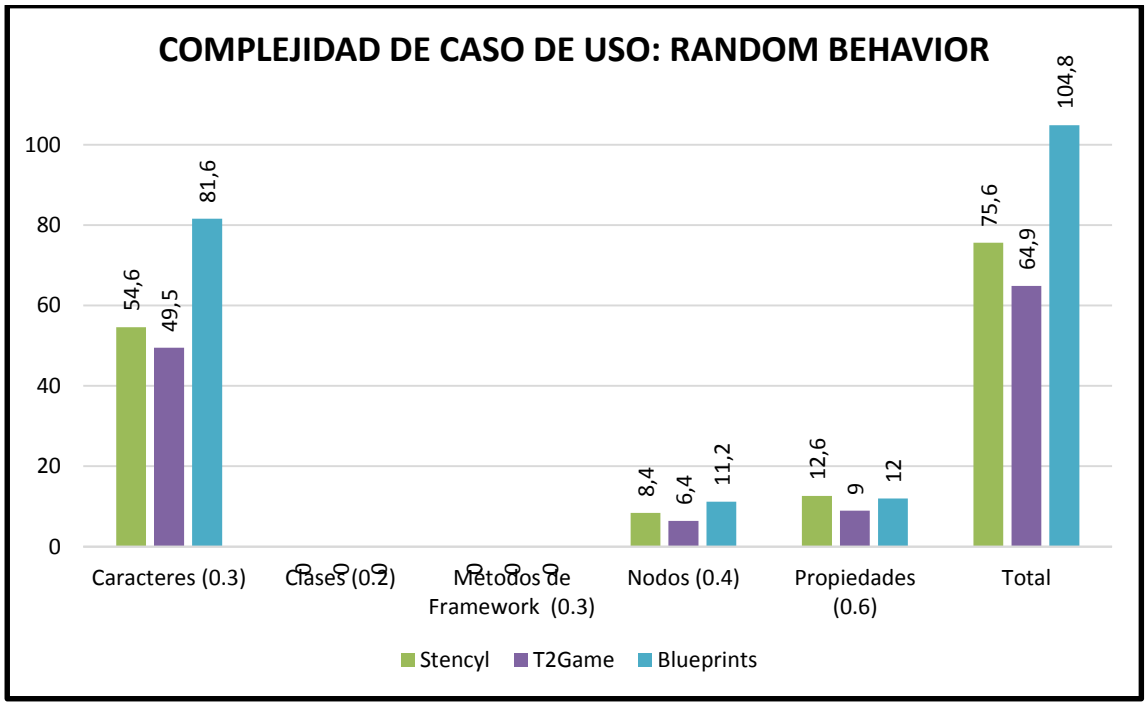


Fig. 78 Gráfico de comparación de alternativas para el caso de uso "Random Behavior"

Se ha hecho una estimación basándose en otros proyectos similares en lo que respecta a las soluciones obtenidas en *Stencyl* y *Blueprints*.

Se han excluido los resultados pertenecientes a las herramientas *Unity* y *Game Maker*, pues los valores sobresalían mucho con respecto a las herramientas *Stencyl*, *T2Game* y *Blueprints*. Este incremento excesivo de valores se debe principalmente al uso de las operaciones con caracteres, pues en una solución de este tamaño tienen mayor cabida.

Para ello, vamos a centrar nuestra comparación en las dos herramientas que más se asemejan a nuestra solución, que son *Stencyl* y *Blueprints*, mostrándolas en escala geométrica.

Al igual que en el caso anterior, en el cómputo general de pasos a seguir para la consecución de la creación de un comportamiento, se demuestra estadísticamente en porcentajes que nuestra solución, comparada con la herramienta *Stencyl*, es un 0,45% más sencilla a la hora de crear comportamientos.

De esta forma, podemos decir que nuestra solución presenta una ligera ventaja en cuanto a complejidad de creación, que para el caso de *Stencyl* estaría situado en torno al 0,42% de media.

Para el caso de *Blueprints*, esta herramienta hace un mayor uso de los nodos a la hora de establecer los diferentes eventos que van surgiendo a lo largo del videojuego.

Comparando ambas soluciones, podemos decir que nuestra solución es aproximadamente un 11% más sencilla que *Blueprints*, en términos de complejidad de creación.

Para el caso de herramientas como *Blueprints*, *Game Maker* o *Unity*, observamos que a medida que el comportamiento que queremos configurar crece en número de opciones, estas diferencias se incrementan bastante, lo que podría darnos pie a confirmar que el grado de complejidad para la creación de estos comportamientos puede ser relativamente más sencillo si se realiza mediante editores con asistente que si se realiza de manera programática, tal y como muestran nuestras mediciones.

Para el caso de *Stencyl*, la diferencia vemos que es pequeña en comparación con nuestra solución, pues se trata de un editor con asistente muy similar a nuestra propuesta. Si bien *Stencyl* se focaliza más en el uso de cajitas que juntas forman un puzle, en el cómputo global de operaciones vemos como se asemeja en parte a nuestra solución, obviamente con sus particularidades.

Capítulo 7. Conclusiones y Trabajo Futuro

En los escenarios desarrollados la propuesta ha sido utilizada satisfactoriamente para modelar varios comportamientos enemigos y generar juegos funcionales. En estos mismos escenarios, y según los análisis realizados y de acuerdo a los criterios establecidos, la utilización de la propuesta resultaría más simple que otras alternativas utilizadas en la actualidad, como por ejemplo se demuestra en el caso de Blueprints, donde nuestra solución es un 11% menos compleja en términos de creación.

Tras los resultados obtenidos, se presentan los aspectos positivos y negativos de la solución propuesta frente a los casos de uso anteriormente citados.

7.1.1 Aspectos positivos de la solución propuesta

Posibilidad de cambio de los patrones de comportamientos, ya que únicamente se centra en los patrones de comportamientos, aislándolos de la mecánica global del videojuego.

Posibilidad de entremezclar un conjunto finito de patrones de comportamiento. Estos comportamientos se pueden combinar y repetir, siendo ilimitado el uso de ellos.

Menor posibilidad de cometer errores, pues los comportamientos están delimitados inicialmente a un conjunto finito de ellos. Siguen unas reglas y deben especificarse según las posibilidades que brinda cada nodo.

Más rapidez en el proceso de creación en comparación con los editores que no proveen un asistente de creación de comportamientos, así como menor complejidad de creación para los diferentes casos de uso.

Fácilmente extensible. Al estar basado en tecnologías EMF, es ampliamente interoperable con otras herramientas.

7.1.2 Aspectos negativos de la solución propuesta

Al tratarse de un conjunto finito de patrones de comportamiento, estaremos limitados a las posibilidades que nos brinde el editor de comportamientos. Esta desventaja se puede paliar con la colaboración de desarrolladores que aporten sus nuevos patrones de comportamiento. No deja de ser una desventaja con respecto a otras plataformas siempre y cuando se tenga en cuenta la parte programática de éstos. Es sabido que programáticamente se pueden desarrollar más patrones de comportamiento, pero este no es nuestro objetivo.

7.2 Trabajo Futuro

El trabajo futuro está orientado de cara a la mejora y optimización de nuestra propuesta. Podríamos citar los siguientes campos para próximas investigaciones:

Diseñar nuevos elementos del lenguaje para dotarlo de más expresividad y que los usuarios puedan especificar de forma sencilla nuevos comportamientos. Un ejemplo de nuevos elementos que se van a añadir a nuestro lenguaje son los siguientes:

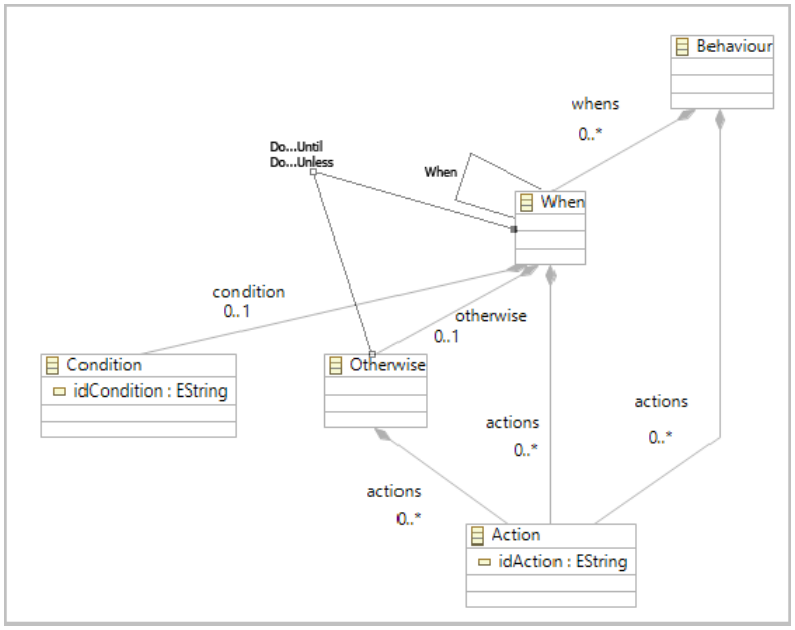


Fig. 79 Metamodelo con futuras incorporaciones

Considerar el uso de nodos *When* sobre sí mismo, así como permitir *Conditions* negativas, de cara a dotar el lenguaje de más expresividad, como ocurre con la inclusión de nuevos nodos *Do…Until* y *Do…Unless*, que podrían englobar acciones que se ejecutaran hasta una determinada condición de parada.

Discutir optimizaciones y mejoras a la estructura propuesta con el fin de hacerlos más entendibles de cara al usuario final, sencillos de modelar y con un nivel de abstracción lo suficientemente potente para permitir variaciones.

Nuevas y estudiadas propiedades que representen fielmente el comportamiento y sean fácilmente entendible por cualquier usuario, como pueden ser la regulación de la velocidad del enemigo, la velocidad de disparo del enemigo, que un enemigo sepa si te estás dirigiendo hacia él, etc. Principalmente, consistiría en parametrizar aquellas condiciones y acciones ya presentes en el lenguaje, para dotarlas de más expresividad.

Diversificar los patrones de comportamiento haciéndolos más específicos, de tal manera que cada enemigo tenga el suyo el propio, además de la parametrización de varios comportamientos. Se podría tener en cuenta la lógica borrosa para este fin.

Promover el uso de patrones de comportamiento no sólo para los enemigos, sino para los diferentes actores que componen un videojuego, como pueden ser ítems, el personaje jugador, elementos del escenario, personajes aliados, etc.

Nueva inmersión de patrones de comportamiento en videojuegos 3D. Un patrón de comportamiento debe poder ser aplicado a cualquier ente cualquiera que sea su plano de acción, planos bidimensionales o planos tridimensionales.

Nueva línea de investigación para patrones de comportamiento retroalimentados, que se alimenten según las condiciones que se presentan en un instante dado, pudiendo aprender del entorno.

Nueva línea de investigación sobre patrones de comportamientos que actúen según emociones, como puede ser la monitorización de los movimientos de un usuario, pudiendo extrapolarlo para nuevas especificaciones de patrones de comportamientos de enemigos.

Capítulo 8. Publicaciones derivadas

"T2Game: A New Specific Domain Language to Model Behavior Patterns on Games".

En la revista:"*Journal of Expert Systems with Applications - Elsevier*".

Indice de impacto 2014 – 2015: 2.240

Estado: **Enviado – esperando contestación**.

Principales revistas en las que el artículo podría haber sido publicado:

- "*Computer Science and Information Systems - Elsevier*"
  - Índice de impacto en 2014 – 2015: 1.456
- "*Journal of Computer Standards & Interfaces - Elsevier*"
  - Índice de impacto en 2014 – 2015: 1.18
- "*Journal of Computational Design and Engineering - Elsevier*"
  - Índice de impacto en 2014 – 2015: estimado en 1.52
- "*Journal of Computational Science – Elsevier*"
  - Índice de impacto en 2014 – 2015: 1.760
- "*Journal of Computing Surverys - Association for Computing Machinery (ACM)*"
  - Índice de impacto en 2014 – 2015: 4.04

Capítulo 9. Referencias Bibliográficas

Estilo de citación IEEE

[1] I. P. Trobo, Desarrollo de videojuegos con DSL, Oviedo: UNIOVI, 2009.

[2] «TIOBE programming community index,» TIOBE, [En línea]. Available: www.tiobe.com. [Último acceso: Mayo 2015].

[3] H. Lowood, History of Computer Game Design Bibliography, Standford, 2005.

[4] X. Tu y D. Terzopoulos, Artificial fishes: Physics, locomotion, perception, behavior, Orlando: In Proceedings of SIGGRAPH, 1994.

[5] J. Funge, X. Tu y D. Terzopoulos, Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Pedestrians, Los Angeles, 1999.

[6] S. Russell y P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education, 1995.

[7] T. S. H. G. Vidaver, Flexible an Purposeful NPC Behaviors using Real-Time Genetic Control, Vancouver, 2006.

[8] C.-N. ZHOU, X.-L. YU, J.-Y. SUN y X.-L. YAN, Affective Computation Based NPC Behaviors Modeling, Proceedings of the 2006 IEEE/WIC/ACM Internation Conference on Web Intelligence and Intelligence Agent Technology, 2006.

[9] D. B. a. G.Seeman, AI for Game Developers, Cambridge: O'Reilly Media, Inc., 2004.

[10] J. Solís-Martínez, J. Pascual Espada, B. C. Pelayo G-Bustelo y J. M. Cueva Lovelle, BPMN MUSIM: Approach to improve the domain expert's efficiency in business process modeling for the generation of specific software applications, Oviedo: Elsevier, 2014.

[11] R. France y B. Rumpe, Model-driven Development of Complex Software: A Research Roadmap, Minneapolis, 2007.

[12] E. Rolando Nuñez-Valdez, O. Sanjuan, B. C. Pelayo García-Bustelo, J. M. Cueva-Lovelle y G. Infante Hernández, Gade4all: Developing Multi-platform Videogames based on Domain Specific Languages and Model Driven Engineering, Oviedo: International Journal of Artificial Intelligence and Interactive Multimedia, 2013.

[13] E. Miotto y T. Vardanega, On the integration of domain-specific and scientific bodies of knowledge in Model Driven Engineering, Padua, 2009.

[14] J. Pascual Espada y V. García Díaz, Ingeniería dirigida por modelos, Oviedo: UNIOVI, 2014.

[15] L. De Seta, «Los Lenguajes específicos de dominio,» 2009. [En línea]. Available: http://www.dosideas.com/actualidad/487-los-lenguajes-especificos-de-dominio.html. [Último acceso: Marzo 2015].

[16] A. L. de M. Santos y A. W. B. Furtado, Applying Domain-Specific modeling to game development with the Microsoft DSL Tools, Pernambuco, 2006.

[17] J. de la Torre Escudero, «Páginas de docencia de Joaquina de la Torre Escudero,» [En línea]. Available: https://www.uam.es/personal_pdi/ciencias/joaquina/BOXES-POP/que_es_un_modelo.htm. [Último acceso: Febrero 2015].

[18] The Eclipse Foundation, «eclipse,» The Eclipse Foundation, 2015. [En línea]. Available: https://www.eclipse.org/modeling/emf/. [Último acceso: Febrero 2015].

[19] «Object Management Group,» 1999. [En línea]. Available: http://www.omg.org/spec/XMI/. [Último acceso: Febrero 2015].

[20] W3C, «Extensible Markup Language (XML),» W3C, 1998. [En línea]. Available: http://www.w3.org/XML/. [Último acceso: Febrero 2015].

[21] Object Management Group, «Object Management Group,» 1997. [En línea]. Available: http://www.uml.org/. [Último acceso: Febrero 2015].

[22] B. Lundell, B. Lings, A. Persson y A. Mattsson, «UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2,» *Model Driven Engineering Languages and Systems,* pp. 619-630, 2006.

[23] Sun Microsytems y Oracle Corporation, «Java,» Oracle, 1995. [En línea]. Available: https://www.java.com/es/. [Último acceso: Febrero 2015].

[24] Eclipse Consortium et al., Java Emitter Templates (JET), 2003.

[25] K. Krogmann y S. Becker, A Case Study on Model-Driven and Conventional Software Development: The Palladio Editor, Karlsruhe, 2007.

[26] A. W. B. Furtado y A. L. M. Santos, Using Domain-Specific Modeling towards Computer Games Development Industrialization, Pernambuco, 2006.

[27] H. Prähofer, D. Hurnaus y H. Mössenböck, Building End-User Programming Systems Based on a Domain-Specific Language, Linz, 2006.

[28] Carton, Andrew; et al., «Aspect-oriented model-driven development for mobile context-aware computing,» de *Proceedings of the 1st International Workshop on Software Engineering for Pervasive Computing Applicaitons, Systems and Environments*, Dublin, IEEE Computer Society, 2007, p. 5.

[29] Y. Hong y Z. Liu, «Preliminary Research on Decision Model Based on Bayesian Techniques For A NPC in Computer Games,» de *International Symposium on Computational Intelligence and Design*, 2010, p. 4.

[30] «Game Maker Studio,» YoYo Games, 15 Noviembre 1999. [En línea]. Available: https://www.yoyogames.com/studio. [Último acceso: Mayo 2015].

[31] J. Chung, «stencyl,» Stencyl, LLC, 31 Mayo 2011. [En línea]. Available: http://www.stencyl.com/. [Último acceso: Mayo 2015].

[32] «Construct 2,» Scirra LTD, 4 Febrero 2011. [En línea]. Available: https://www.scirra.com/construct2. [Último acceso: Mayo 2015].

[33] W3C, «W3C - HTML5,» Octubre 2014. [En línea]. Available: http://www.w3.org/TR/html5/. [Último acceso: Junio 2015].

[34] «Blueprints Visual Scripting,» Unreal Engine, 1998. [En línea]. Available: https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html. [Último acceso: Mayo 2015].

[35] D. Helgason, «unity,» Unity Technologies, 30 Mayo 2005. [En línea]. Available: https://unity3d.com/es. [Último acceso: Mayo 2015].

[36] «Maya,» Autodesk, Febrero 1998. [En línea]. Available: http://www.autodesk.com/products/maya/overview. [Último acceso: Abril 2015].

[37] O. Alon y J. Rimokh, «ZBrush,» Pixologic ZBrush, 1999. [En línea]. Available: http://pixologic.com/. [Último acceso: Abril 2015].

[38] Novell Inc., Ximian (Xamarin) y Mono Community, «Mono,» Novell Inc., Ximian (Xamarin), Mono Community, 30 Junio 2004. [En línea]. Available: http://www.mono-project.com/. [Último acceso: Mayo 2015].

[39] G. van Rossum, «Python Software Foundation,» 1991. [En línea]. Available: https://www.python.org/. [Último acceso: Mayo 2015].

[40] «Cocos2d-JS,» 29 Enero 2012. [En línea]. Available: http://www.cocos2d-x.org/wiki/Cocos2d-JS. [Último acceso: Mayo 2015].

[41] «Cocos2d game framework for xna,» [En línea]. Available: http://www.cocos2dxna.com/. [Último acceso: Mayo 2015].

[42] B. Cox, «Mac Developer Library,» Apple Inc., 1980. [En línea]. Available: https://developer.apple.com/library/mac/navigation/. [Último acceso: Mayo 2015].

[43] S. Jobs, S. Wozniak y R. Wayne, «Apple,» Apple Inc., 1 Abril 1976. [En línea]. Available: https://www.apple.com/. [Último acceso: Mayo 2015].

[44] «ECMAScript,» Netscape Communication Corp. ; Mozilla Fundation, 1995. [En línea]. Available: http://www.ecmascript.org/. [Último acceso: Mayo 2015].

[45] «Visual C#,» Microsoft, 2000. [En línea]. Available: https://msdn.microsoft.com/es-es/library/kx37x362.aspx. [Último acceso: Mayo 2015].

[46] Object Management Group, «OMG - Object Management Group,» Object Management Group, Inc., 1997. [En línea]. Available: http://www.omg.org/. [Último acceso: Mayo 2015].

[47] D. C. Schmidt, Model-Driven Engineering, IEEE Computer, 2006.

[48] «What is Sirius,» The Eclipse Foundation, [En línea]. Available: http://www.eclipse.org/community/eclipse_newsletter/2013/november/article1.php. [Último acceso: Mayo 2015].

[49] «Wikipedia. La enciclopedia libre.,» 7 Febrero 2002. [En línea]. Available: http://es.wikipedia.org/?title=GNU/Linux. [Último acceso: Mayo 2015].

[50] I. Google, «Google,» [En línea]. Available: https://play.google.com/store. [Último acceso: Mayo 2015].

[51] N. G. Menéndez, Desarrollo dirigido por modelos y basado en DSL´s de videojuegos de acción en 2D para dispositivos móviles, Oviedo: UNIOVI, 2012.

[52] «Android SDK,» Google Inc., 23 Septiembre 2008. [En línea]. Available: https://developer.android.com/sdk/index.html. [Último acceso: Mayo 2015].

[53] «Dart,» Google Inc., 2011. [En línea]. Available: https://www.dartlang.org/. [Último acceso: Mayo 2015].

[54] D. M. Ritchie y B. W. Kernighan, «The C Programming Language,» Eastern Economy Edition, 1972. [En línea]. Available: http://www.amazon.com/Programming-Language-Brian-W-Kernighan/dp/8120305965/ref=la_B000AQ6LVG_1_1/192-5691159-4167335?s=books&ie=UTF8&qid=1432572699&sr=1-1. [Último acceso: Mayo 2015].

[55] «Ingeniería dirigida por modelos - Universidad de Oviedo,» Universidad de Oviedo, [En línea]. Available: http://grupos.uniovi.es/web/mderg. [Último acceso: Mayo 2015].

[56] GamerDic, «GamerDic: diccionario online de términos sobre videojuegos y cultura gamer,» 2015. [En línea]. Available: http://www.gamerdic.es/termino/gameplay. [Último acceso: Mayo 2015].

Capítulo 10. Apéndice