Construcción de un Generador de Escenas usando técnicas DSDM

Proyecto Fin de Carrera.

FACULTAD DE INFORMÁTICA

UNIVERSIDAD DE MURCIA

Simón González González

Proyecto dirigido por:
Francisco Javier Bermúdez Ruiz

Departamento de Informática y Sistemas.
Universidad de Murcia.

Julio 2010

Agradecimientos.

Conseguir llegar al término de una Ingeniería Superior no es únicamente el resultado de una labor y un esfuerzo personal, precisa también del apoyo de muchísima gente con la que se comparten esos años. Por este motivo quisiera dedicar unas líneas a las personas que han compartido mi camino universitario.

A mis padres, por inculcarme la importancia de realizar una carrera y por otorgarme la posibilidad de estudiar lo que quería, en especial a mi madre, por entender a la perfección la dificultad que conlleva. A mi hermana, por su apoyo incondicional en todo este tiempo. A mis abuelos, porque son las personas más ilusionadas con mi titulación. A mis compañeros de facultad, en especial a mis compañeros de prácticas a lo largo de la carrera: José Luís Abellán, Arturo Mellado, Julio Soler, David Juan Tudela, María del Mar Rubio… y un largo etcétera, por su ayuda y por todos los momentos vividos. A mis amigos de toda la vida, por su confianza en mí, en especial a mi gran amigo Juan José Cerón, también futuro Ingeniero, por prestarme el portátil para hacer este proyecto, aunque eso es lo de menos… A mi novia, por ser el impulso que me faltaba para terminar la carrera. Y a Francisco Javier Bermúdez Ruiz, director de este proyecto y el mejor que he podido tener.

A todos, GRACIAS.

En la vida, no hay que temer nada, solo tratar de comprender.

Marie Curie.

Índice.

Índice de Figuras.

1. Introducción.

1.1 Contexto

La automatización es un concepto clave que aparece en la evolución del desarrollo del software y que se ha perseguido desde sus principios. El modo en que se ha desarrollado el software a lo largo de los años ha sufrido principalmente dos cambios paradigmáticos que se presentan como hitos. El primero fue la aparición de los lenguajes de programación a principios de los sesenta. Lenguajes como FORTRAN o COBOL permitieron una programación de alto nivel independiente de la máquina, con un nivel de expresividad muy superior al proporcionado por los lenguajes ensambladores. Los compiladores fueron el primer paso en el camino de la automatización transformando el código fuente del lenguaje de programación en código máquina a partir de la gramática del lenguaje.

El segundo hito fue la aparición del Lenguaje Unificado de Modelado (UML) a mediados de los noventa. Este lenguaje se ha convertido en un estándar de facto para el modelado de sistemas software y ha incrementado de forma significativa el interés por el modelado en todas las etapas del desarrollo del software. UML abrió las puertas al Desarrollo de software Dirigido por Modelos o Ingeniería de Modelos, un nuevo paradigma en el que los desarrolladores primero crean modelos independientes de la plataforma de implementación (PIM, Platform Independent Model) y compiladores de modelos transforman esos modelos en otros que incluyen los detalles propios de la plataforma (PSM, Platform Specific Model). La especificación Model Driven Architecture (MDA), propuesta por el consorcio OMG, establece las bases de este nuevo paradigma. El Desarrollo Dirigido por Modelos supone un nuevo paso para aumentar la automatización. A partir de la definición de modelos PIM los compiladores de modelos generan los modelos PSM y el código para un lenguaje de programación concreto. Por tanto, se aumenta el nivel de abstracción que maneja el desarrollador y el grado de automatización. Se consigue una independencia del sistema operativo y otros detalles de la máquina mediante el uso de los compiladores de lenguajes de programación, y de la plataforma de implementación (por ejemplo EJB, .NET, etc) mediante el uso de compiladores de modelos.

Un concepto importante en MDA es el de transformación, el cual juega un papel fundamental para conseguir construir cada uno de los modelos que guían el desarrollo. Las herramientas de transformación permiten convertir automáticamente un modelo PIM en un modelo PSM, así como convertir los modelos PSM en código. Los modelos utilizados durante el proceso de desarrollo deben estar definidos en un determinado lenguaje, que debe tener una sintaxis y una semántica bien definida. Este lenguaje que define al modelo recibe el nombre de metamodelo, que es otro concepto importante en el contexto de MDA. Su importancia es doble, pues dado que las transformaciones trabajan con modelos, deben poder analizarlos y verificarlos para así poder realizar las transformaciones que, por lo tanto, se apoyan en los metamodelos para poder trabajar con los modelos.

Eclipse es una plataforma extensible basada en Java destinada al desarrollo de software. Fue creada por un consorcio de empresas liderado por IBM. Su arquitectura extensible permite ampliar su funcionalidad a base de plugins, los cuales se integran en la plataforma con facilidad. En la actualidad existen un gran número de proyectos que desarrollan y mantienen plugins para la plataforma. Algunos de los proyectos de la plataforma están orientados o facilitan el Desarrollo Dirigido por Modelos, como por ejemplo EMF o GMF.

En la actualidad, existen generadores de escenas, principalmente gráficas, que permiten la generación de animaciones. Como pionera en este campo cabe resaltar la compañía Macromedia (posteriormente adquirida por Adobe). Macromedia estuvo dedicada a proporcionar dinamismo a un entorno inicialmente tan estático como eran las páginas web. Dentro de esta política, su producto más representativo fue Flash y ofrecía además una herramienta que permitía generar animaciones comprimidas especialmente para transmitirse con mayor agilidad por Internet.

Además de mencionar a la empresa Adobe Macromedia y su generador de escenas para animaciones de Flash, existen numerosos usos para los cuales se puede adaptar el concepto de escena y el de generador de escenas (escenarios en realidad virtual, viñetas o storyboards en dibujo, representaciones de modelos atmosféricos en meteorología, etc.) En este proyecto se hablará de la escena desde un punto de vista simple y genérico para permitir que dicho concepto se pueda adaptar a cualquiera de los ámbitos anteriormente mencionados, siempre que se pretenda (y sea posible) una generación automática de escenas.

1.2 Objetivos

El objetivo principal del proyecto es la construcción de un generador de escenas empleando técnicas DSDM. Se pretende obtener un generador que sea reutilizable y extensible para la aplicación de la generación de escenas en dominios particulares. Se trata de una herramienta que permita generar una secuencia de escenas a partir de una escena inicial y un conjunto de acciones que suceden en la secuencia de escenas en instantes de tiempo concretos. Con dicha escena inicial y con las acciones, puesto que el tipo de escena que se genera está supeditado a un conjunto de reglas que determinan el comportamiento de los componentes de la escena frente a las acciones que se producen, el generador será capaz de construir una secuencia de escenas en un rango de instantes de tiempo.

Indicar también que como complemento al generador de escenas se requiere de un visualizador de escenas, que por otro lado también permita la necesaria edición de las escenas (al menos, de la escena inicial).

Comentar que otro objetivo del proyecto es que el alumno aprenda los conceptos que subyacen al DSDM y experimente con tecnologías DSDM para la construcción de software.

Por último, comentar que como objetivo secundario del proyecto está el aplicar el generador de escenas sobre un dominio concreto de prueba. En este proyecto se ha

buscado representar las escenas en un dominio simple, donde los elementos que compongan la escena sean simples y claros, donde las acciones que puedan realizar los elementos también sean simples y bien definidas, y donde las reglas que rigen a los elementos de la escena y sus acciones estén bien acotadas Dicho dominio simple lo constituye el escenario del juego de carreras denominado Lightcycle (que se puede encontrar en el film Tron [11] de Walt Disney Productions).

1.3 Metodología

El proyecto se ha llevado a cabo en dos periodos de tiempo separados con dedicación completa en lo que se refiere a la parte del desarrollo de la aplicación. El primero de estos periodos transcurrió desde mediados de septiembre hasta principios de diciembre, mientras que el segundo periodo transcurrió desde mediados de febrero hasta mediados de abril. Por su parte, la redacción de la memoria comprendió un periodo desde finales de abril hasta finales de junio, dedicando exclusivamente los fines de semana.

La primera fase consistió en estudiar y comprender el desarrollo de software dirigido por modelos, así como el estudio de las tecnologías EMF y GMF. Esta fase llevó un par de semanas. Una vez adquiridos los conocimientos en el desarrollo de metamodelos y modelos, se dedicaron un par de semanas a la definición de un metamodelo inicial para la aplicación al que se le efectuaron sucesivos refinamientos. Esta fase llevó otras dos semanas.

En la tercera fase se inició el desarrollo de un visualizador/editor (de ahora en adelante, visualizador) de escenas con GMF, al que se le fueron haciendo también sucesivos cambios, que obligaron a rehacer varias veces el visualizador desde el principio. Esta fase supuso algo más de un mes, con lo que no se terminó hasta finales de noviembre. La última semana y media de este primer periodo de trabajo del proyecto se dedicó a estudiar los ficheros generados por el visualizador de escenas y también, aunque después no se ha llegado a utilizar en el proyecto, se buscó información para automatizar la forma de desarrollar el visualizador de escenas mediante Java.

En este punto del proyecto, con el visualizador de escenas terminado, se dejó un tiempo el proyecto y se volvió a retomar a mediados de febrero. Lo primero que se hizo fue estudiar las distintas posibilidades para hacer un visualizador de escenas extensible, aunque finalmente se descartó esta parte. Este periodo llevó unas dos semanas.

A continuación se procedió al estudio de RubyTL para desarrollar el generador de escenas, lo cual llevó una nueva semana. Esta parte del desarrollo del generador llevó otro mes en el cual se hicieron las transformaciones modelo-modelo y modelo-código en RubyTL, además de dedicar varios días al estudio y uso de la API de EMF.

Finalmente se dedicó una última semana a perfilar las dos herramientas desarrolladas. Esta parte finalizó a mediados de abril. La última fase del proyecto vino con la documentación de la memoria que llevo desde finales de abril hasta finales de junio.

1.4 Organización del Documento

En lo que sigue, este documento se encuentra estructurado de la siguiente manera:

En el Capitulo 2 se explica la problemática del proyecto que se va a desarrollar.

En el Capítulo 3 se describe la solución implementada, explicando la arquitectura generativa definida en el proyecto.

En el Capitulo 4 y en adelante, se describe de manera detallada la solución técnica del proyecto. En concreto, en este capítulo se hace una introducción del significado y uso de metamodelos Ecore y se detalla la creación del metamodelo base así como el sentido de cada uno de sus elementos.

En el Capitulo 5 se explica cómo se ha desarrollado la herramienta del ***visualizador de escenas***, indicando los pasos a seguir, las limitaciones encontradas y los resultados obtenidos. Como introducción se describe la tecnología empleada: GMF

En el Capitulo 6 se explica cómo se ha llevado a cabo el desarrollo del ***generador de escenas***. También incluye un apartado introductorio sobre el lenguaje de transformación RubyTL y el uso de la API de EMF, tecnologías ambas empleadas en la construcción del generador.

El Capitulo 7 indica las conclusiones obtenidas tras la conclusión del proyecto, y las posibles vías futuras derivadas del mismo.

En el Capitulo 8 se indican las referencias utilizadas para llevar a cabo el proyecto.

Finalmente, en el Capitulo 9 se incluyen una serie de anexos con información adicional del proyecto.

2. Descripción del Problema.

En la actualidad, el Desarrollo de Software Dirigido por Modelos dispone de multitud de tecnologías y herramientas. En este proyecto se busca resolver los objetivos del mismo aplicando en la medida de los posible dichas tecnologías, aprovechando sus ventajas en el contexto que delimita el proyecto. La aplicabilidad de las tecnologías DSDM puede ser muy variada. En este proyecto se hará empleo de lenguajes de transformación de modelos y para la generación de código para la construcción de un generador de escenas. Por lo tanto, es necesario abordar la experimentación de dichas tecnologías en el problema que nos ocupa.

En este proyecto se selecciona un dominio de aplicación para dichas tecnologías DSDM. Se trata de construir un generador de escenas con una serie de características deseables, tales como la extensibilidad y la reutilización. Un generador de escenas es una herramienta que permite la generación de una secuencia de escenas en un rango de instantes de tiempo, partiendo de una escena inicial y de un conjunto de acciones que tendrán lugar temporalmente dentro de las escenas, y de acuerdo a unas reglas que rigen el comportamiento del tipo de escena.

Se hace necesario ahora definir el concepto de escena. Una escena comprende un conjunto de objetos y acciones que existen y tienen lugar en un determinado instante de tiempo. Los objetos pueden ser de cualquier naturaleza (físicos, conceptuales…) y las acciones que se producen en la escena están directamente ligadas con el tipo de objetos que define la escena. El generador de escenas debe construir una secuencia de escenas, ordenada por los instantes de tiempo que definen a cada escena.

Desde esta definición tan amplia del concepto de escena se pueden recoger muchos dominios de aplicación, como pueden ser dominios cuyas escenas estén conformadas por objetos físicos, con posiciones espaciales (existe por tanto un espacio de escena) y donde las acciones de la escena estén orientadas a las relaciones físicas entre los objetos de la escena y la propia escena. Esta definición de escena a un nivel poco detallado sería el marco de partida para el dominio que se aplicará en el proyecto como ejemplo para el generador de escenas.

Pero, por otro lado, se pueden recoger otros dominios de aplicación más conceptuales. Por ejemplo, el cuadro sintomático de un paciente. Se trataría de una escena donde los elementos que la componen son conceptuales. Donde existen relaciones físicas entre los elementos, pero no de naturaleza espacial (no existe el concepto de espacio en la escena). Donde dichos elementos poseen relaciones temporales (por ejemplo, orden de aparición de los síntomas) y donde las acciones están orientadas al tratamiento de dichos síntomas. En este caso, un generador de escenas permitiría evaluar el comportamiento de los síntomas de un paciente ante los distintos tratamientos que pudieran aplicársele (suponiendo que se es capaz de definir reglas para la escena que determinen la respuesta del paciente ante las acciones/tratamientos: por ejemplo, ante la aplicación de un medicamento antitérmico en ciertas cantidades, un paciente con ciertas características físicas sufre una bajada de temperatura en una cantidad de grados). Como

se puede comprobar, el concepto de escena puede ajustarse a muchos ámbitos de aplicación.

En el caso del proyecto que nos ocupa, el concepto de escena para su posterior generación ha sido muy simple. Se ha buscado un dominio de aplicación que pruebe el funcionamiento del generador y se ha optado por la perspectiva de escena como representación delimitada de objetos físicos con posiciones bidimensionales. Dichos objetos físicos pueden ser móviles o fijos. En concreto, se ha utilizado en este proyecto el concepto de escena para representar cada uno de los instantes de tiempo que tienen lugar en el juego de carreras denominado Lightcycle, que se puede ver en el film Tron. En este juego, básicamente existen un número indeterminado de objetos móviles que son lo que se denominan "motos de luz". Durante cada instante de la carrera dicho objeto genera un desplazamiento (siempre) y deja tras de si un rastro que actúa como objeto inmóvil. El objetivo del juego es mantenerse en la carrera, intentando no colisionar con ningún objeto móvil ni inmóvil. Por lo tanto en este problema, la escena en cada instante de tiempo contiene objetos móviles, fijos y se producen además acciones por parte de los objetos móviles (los únicos que generan acciones en este dominio de problema). Dichas acciones son de cambio de dirección. Puesto que en este juego de carreras, los objetos solo se desplazan en 4 direcciones (no existen las diagonales), las acciones solo pueden ser de cambio de dirección: arriba, abajo, izquierda y derecha.

Una vez se conoce el concepto de escena (y consecuentemente, secuencia de escenas), y se ha descrito el tipo de escena (Lightcycle) que se empleará como prueba, se va a entrar en detalle en las necesidades del generador de escenas que se requiere construir. El generador de escenas es una herramienta que permite la generación de secuencias de escenas (ordenadas temporalmente) a partir de una escena inicial y un conjunto de acciones que sucederán en diversos instantes de tiempo. Aunque el generador debería permitir interacción en cuanto a la determinación de las acciones conforme se van generando la secuencia de acciones, en esta primera aproximación que supone el proyecto, se busca obtener un generador que reciba la lista de acciones que sucederán como entrada del mismo. El generador debe tener un dominio de aplicación establecido (que en nuestro ejemplo será el Lightcycle). Esto se traduce en que dicho generador tendrá una definición de escena concreta y unas reglas de comportamiento que determinarán en cada instante de tiempo (escena) cual es la siguiente escena, según los objetos y las acciones que se produzcan, y en resumen, según el estado de la escena actual. Por lo tanto el generador de escenas puede ser percibido como una caja negra que se configura para un dominio de aplicación, recibe una escena inicial y una lista de acciones en diversos instantes de tiempo y genera la secuencia de acciones correspondiente.

Además de un generador de escenas, se requiere construir como herramienta de apoyo un visualizador/editor de escenas (realmente de secuencia de escenas) para disponer de una suite que permita de manera integrada la definición de las entradas del generador (escena inicial y acciones) y visualizar la salida del generador (secuencia de escenas).

Una de las claves en la construcción del generador de escenas es conseguir que este sea extensible y reutilizable (tanto generador como visualizador) sea cual sea la interpretación del concepto de escena aplicado (dominio de la secuencia de escenas). Es decir, ser capaz de generar una arquitectura generativa (de secuencia de escenas) que

sea extensible. Así pues sería deseable que el generador de escenas pudiera extender un dominio de aplicación inicial con nuevos conceptos, reutilizando los conceptos base del dominio inicial e incorporando nuevos conceptos, pudiendo reutilizar funcionalmente todo lo creado para el dominio inicial. Surge entonces la idea de crear el generador de escenas como un framework que permita ser extendido con dominios de aplicación concretos. Y aun más, que los dominios puedan ser reutilizables o incluso exista el concepto de herencia entre los dominios.

Otra de las características deseables en la construcción de la solución sería que tanto el generador como el visualizador se encontraran integrados y permitieran la interacción del usuario durante la generación de la secuencia de escenas. Esto permitiría que el generador de escenas recibiera las acciones en diversos instantes durante la generación de la secuencia de escenas, permitiendo mayor interacción con el usuario del generador.

3. Diseño de la Solución.

Tal y como se comentó en el apartado anterior, para llevar a cabo el proyecto se deben crear dos herramientas distintas, un generador y un visualizador de escenas. Para el caso del generador de escenas, básicamente consistirá en una herramienta que dada una escena inicial definida por el usuario con los elementos de los que se quiera que conste, y definiendo una serie de acciones que ejecutan los elementos de la escena a lo largo del tiempo, la herramienta será capaz de devolver la secuencia entera de escenas que se han producido en un tiempo determinado. Para ello la herramienta del generador de escenas debe tener definidas una serie de reglas que dictaminen que ocurrirá al producirse las acciones o simplemente bajo unas condiciones determinadas del dominio del problema. Por su parte, el visualizador de escenas será una herramienta que permitirá visualizar de manera gráfica la secuencia de escenas generada, e incluso, permitirá su edición. A continuación se muestra un esquema donde aparecen las principales tareas que se deben llevar a cabo para desarrollar el proyecto.

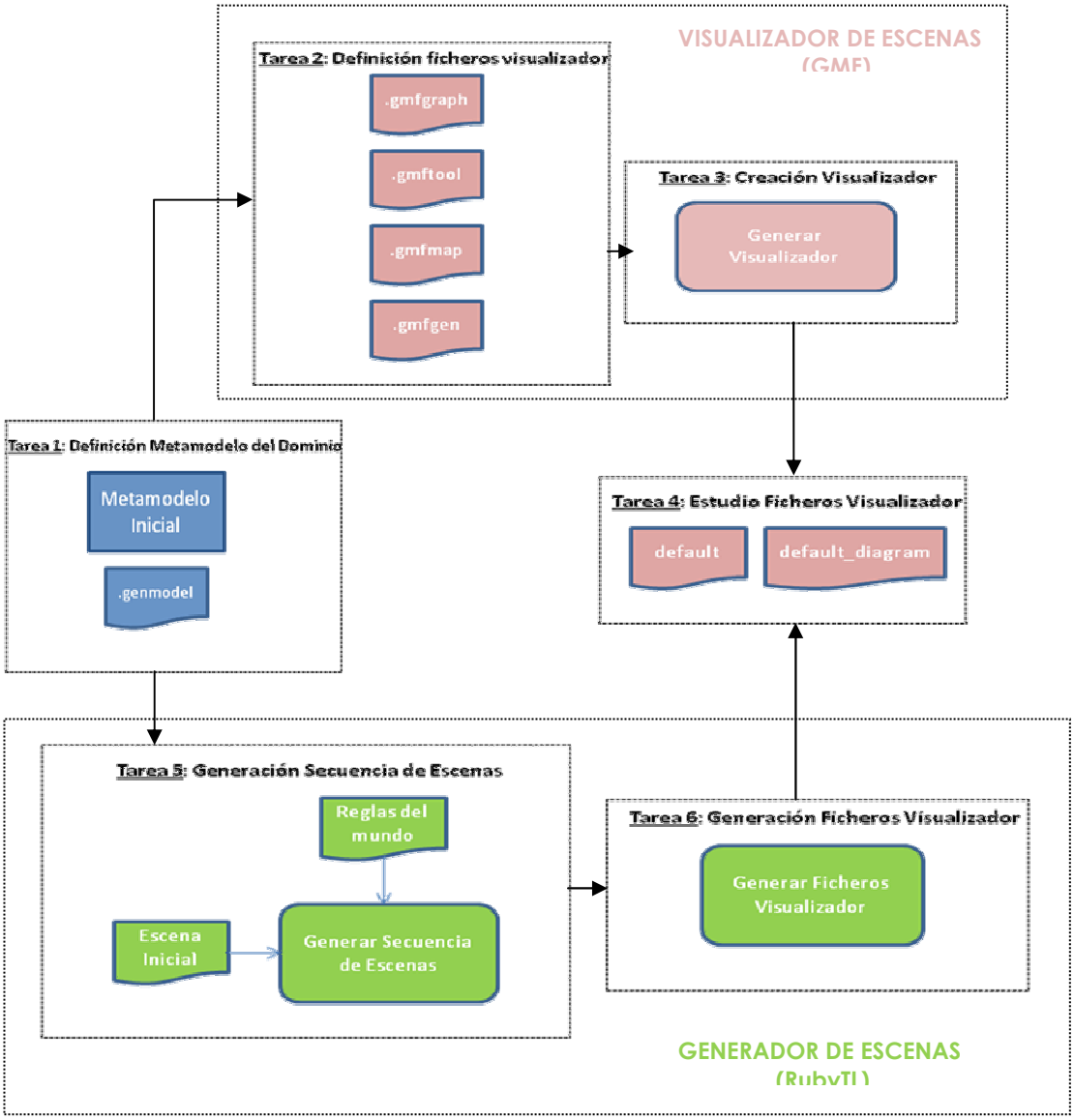


Figura 1. Esquema Global

Como se observa en el esquema, ambas herramientas parten de un punto común que es la primera tarea del proyecto. En dicha tarea se debe definir mediante EMF el metamodelo del dominio de la aplicación. En primer lugar se generaría en formato Ecore y a continuación se generará de forma automática su correspondiente fichero .genmodel.

Una vez que tenemos el metamodelo inicial se pasaría a la parte de creación del visualizador de escenas, que sería la segunda tarea. En ella se deben crear 4 ficheros (.gmfgraph, .gmftool, .gmfmap y .gmfgen) en base al metamodelo de partida, necesarios para la creación del visualizador, todo ello mediante GMF.

Ya en la tercera tarea se generará el visualizador propiamente dicho, para ello y gracias a los ficheros generados anteriormente se generará automáticamente el código del visualizador invocando dos métodos también de GMF.

Con el visualizador de escenas creado, que también puede ser utilizado como un editor de escenas, se harán una serie de pruebas en la cuarta tarea con el fin de entender la sintaxis y semántica de un par de ficheros que son la base para poder visualizar una secuencia de escenas. Estos ficheros son “default” y “default_diagram”. Una vez entendidos se pasaría a la creación del generador de escenas que será el encargado de crear una secuencia de escenas y definirla en base a estos dos ficheros los cuales se pasarán al visualizador para mostrar la secuencia de manera visual.

Es en la quinta tarea donde se comienza a crear el generador, en esta tarea lo que se hace es definir una escena inicial en base al metamodelo del dominio de la aplicación la cual se pasará a un proceso implementado en RubyTL que consiste en una sucesión de transformaciones modelo-modelo que generarán la secuencia de escenas final, basándose además para ello en unas reglas definidas para el dominio del problema. Además de RubyTL se hará uso en este punto del API de EMF para determinar cuando concluye dicho proceso de generación de escenas.

Finalmente en la sexta tarea, lo que hace el generador de escenas es transformar la secuencia de escenas generada en la tarea anterior en código y más concretamente en los ficheros “default” y “default_diagram” que se estudiaron en la parte del visualizador. Esta parte también se lleva a cabo mediante RubyTL y el uso de dos plantillas una por fichero a generar, que determinan la estructura de los ficheros. Una vez generados los ficheros simplemente quedaría copiarlos al visualizador para poder mostrarlos.

4. Metamodelo Base.

En el siguiente capítulo se describirá como se definió el metamodelo base del generador, empleando EMF.

4.1 Introducción EMF.

Eclipse Modeling Framework (EMF) es un framework de la plataforma Eclipse desarrollado con el objetivo de construir de manera sencilla aplicaciones basadas en modelos. Entre otras muchas cosas EMF facilita la labor de construcción de modelos ofreciendo un lenguaje sencillo como es el lenguaje Ecore. El cual permite la generación automática de código Java a partir de ellos u ofrece por ejemplo una API para recorrer de manera intuitiva la estructura de los mismos con el objetivo de acceder a partes del modelo de manera rápida. Es decir, es un plug-in para Eclipse que hace el modelado fácil.

EMF está dividido en dos partes: el framework Core, responsable de la gestión de los modelos, la generación de código básica y del entorno de ejecución; y el framework EMF.Edit que se construye sobre el framework Core y permite la construcción de editores gráficos de los modelos para la plataforma Eclipse. Además, EMF consta de otro componente importante: EMF Codegen. Este componente es el generador de código para los dos frameworks anteriores, y también se encarga de importar modelos desde otros formatos como Rational Rose, XML o Java.

El metamodelo Ecore es el lenguaje de metamodelado que permite definir metamodelos en el framework EMF. A su vez, el propio lenguaje está definido por un metamodelo Ecore, es decir, Ecore está definido en Ecore, por lo que se dice que es un meta-metamodelo. El lenguaje Ecore tiene sus raíces en UML y MOF, esto hace que posea muchos conceptos comunes con UML, como los conceptos de clase, atributo y asociación. Y aunque sea un subconjunto simplificado de MOF, soporta ciertos conceptos de alto nivel no incluidos directamente en Java, como contenciones, relaciones bidireccionales y herencia múltiple.

Los elementos más importantes de los que consta el lenguaje Ecore son: las metaclases en Ecore (el elemento EClass), las cuales poseen un conjunto de meta-atributos por medio de un conjunto de elementos EAttribute los cuales pertenecen a un tipo que viene definido por el elemento EDataType. Por otra parte una metaclase también puede estar relacionada con otras por medio de elementos EReference.

El framework EMF tiene como característica principal la generación automática del código Java que implementa el modelo. A partir de un modelo Ecore, el framework crea un modelo generador que controla los detalles de la generación automática de código. El modelo generador extiende al modelo Ecore incluyendo un conjunto de atributos que particularizan la generación de código. La generación de código puede dar lugar a un conjunto de clases e interfaces que implementan el modelo, un editor gráfico del modelo y un conjunto de clases de test para probar el modelo.

El framework ofrece una API para la recuperación y el almacenamiento de los modelos. EMF incluye una implementación por defecto, utilizando XMI (XML Metadata Interchange) como formato de persistencia para cualquier modelo. Sin embargo, si el modelo fue definido mediante un esquema XML, EMF permite guardar el modelo como una instancia XML del documento que sigue dicho esquema.

Para identificar únicamente a un elemento de un modelo Ecore, al propio modelo, o a un recurso, EMF utiliza una URI. Una URI (Uniform Resource Identifier) es una referencia estándar (parecida a una URL) para la localización de datos, que pueden estar en cualquier tipo de almacenamiento (ficheros, Internet, base de datos, etc.).

El componente EMF.Edit de EMF permite construir visores y editores gráficos para los modelos EMF. Los editores permiten mostrar y editar (copiar, pegar, arrastrar y soltar, deshacer, rehacer, etc.) instancias del modelo utilizando vistas y hojas de propiedades estándares de Eclipse.

4.2 Metamodelo del LigthCycle.

El punto de partida del proyecto es la definición del metamodelo del LigthCycle, es más, tal y como se vio, será el punto de partida de las dos herramientas que se van a generar: el visualizador de escenas y el generador de escenas. Por ello, es crucial que la definición del metamodelo sea correcta. Han sido numerosos los refinamientos que se han ido realizando a lo largo del proyecto sobre el metamodelo inicial con el fin de ir adaptándolo convenientemente a las necesidades que iban surgiendo de la aplicación o a las limitaciones que iban apareciendo a lo largo del proyecto. Tanto es así que la idea inicial no era la de partir de un metamodelo específico del problema del LigthCycle, sino un metamodelo genérico de escenas el cual se pudiese extender o concretar en un metamodelo específico de un juego, cómic, película… El problema que surgió con esta idea de partir de un metamodelo genérico, es que en EMF existe la opción “Load Resource…” que permite precisamente esto que se pretendía hacer, extender un metamodelo genérico. Pero al utilizar el metamodelo en GMF para construir el visualizador, GMF no es capaz de entender como está definida la extensión en EMF, es decir, no es capaz de enlazar el metamodelo genérico que se extiende con el metamodelo específico que hereda de él, más concretamente, no es capaz de localizar las clases que se heredan del metamodelo genérico. Este inconveniente supuso el primer gran motivo para desviarse un poco de la idea inicial y enfocar el proyecto en la obtención de un caso concreto y específico como es el problema del LigthCycle.

Por ello el punto de partida del proyecto es la creación de un metamodelo del problema del LigthCycle. A continuación se describe el metamodelo que se empleó para las dos herramientas desarrolladas:

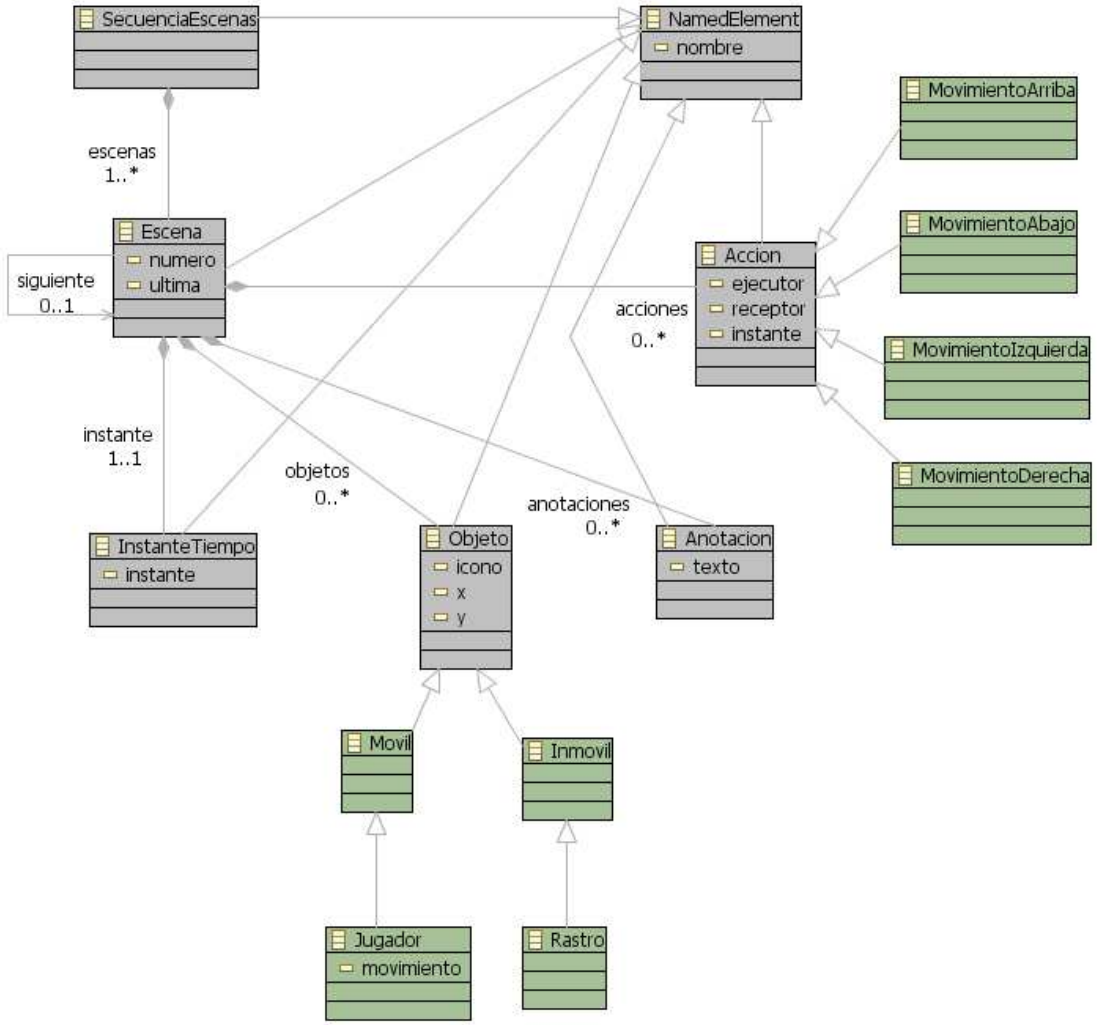


Figura 2. Metamodelo LigthCycle

Como se puede observar en la figura, existen clases de dos colores distintos (gris y verde). En tono gris aparecen las clases que deberían pertenecer al metamodelo genérico, mientras que en tono verde aparecen las clases concretas del metamodelo que se va a desarrollar (Metamodelo del LygthCycle). Como ya se ha comentado, lo ideal y como se comenzó a desarrollar el proyecto fue definiendo un metamodelo básico que contaba simplemente con las clases en tono gris, mientras que para cada problema nuevo que se acometiera, habría que definir un metamodelo nuevo, el cual extendiese el metamodelo base e incorporase las clases especializadas del problema concreto.

El metamodelo tiene como elemento raíz "SecuenciaEscenas" que representa el conjunto de todas las escenas. A continuación aparece el elemento escena propiamente dicho, que reflejará la situación del dominio del problema en un momento concreto. Como se puede ver, contiene dos atributos: un atributo de tipo entero que marca el número de la escena y un atributo denominado "ultima" de tipo booleano que indica si se trata de la última escena de la secuencia (la escena final de la partida). Además una escena tiene una referencia a sí misma ("siguiente") que enlaza una escena con la siguiente en la secuencia en caso de haberla. Cada escena está formada por una serie de

objetos, los cuales pueden ser móviles o inmóviles. Para el dominio del problema del LigthCycle van a ser “Jugadores” (normalmente 2) para el caso de los móviles y objetos “Rastro” por parte de los inmóviles. Estos objetos representan el rastro dejado por los jugadores en cada movimiento. Los objetos tienen los atributos “x” e “y” que indican la posición del objeto dentro de la escena y un atributo icono el cual se explicará más adelante, pero que va a permitir asignarle a los objetos un icono con el fin de obtener una representación mejor. Dicho atributo, a pesar de ser de tipo string, no va a contener ningún valor.

Además de los objetos, una escena puede contener una serie de anotaciones. Estas anotaciones aparecerán en la escena cuando ocurra algo que dé por terminado el problema con el fin de indicar qué ha ocurrido. Por ejemplo, aparecerán anotaciones que indiquen un choque entre jugadores, o un choque de un jugador con un rastro, o la salida de un jugador de los límites de la escena. Todos estos hechos serán informados mediante un mensaje en la escena.

Otro de los elementos de la escena es el “InstanteTiempo”, el cual indica el momento concreto (instante) en que se produce la escena, es decir, el típico reloj de todos los juegos que va marcando el tiempo transcurrido. Este elemento es muy importante porque en función del instante en el que se produzca la escena se aplicará una acción concreta o no.

Precisamente, el elemento “Accion” es el último elemento que puede contener la escena. Sobre una escena inicial se pueden declarar una serie de acciones, las cuales se irán realizando a medida que el problema avance y se llegue al momento oportuno. El atributo instante de “Accion” indica precisamente en qué momento se debe ejecutar la acción. Este atributo será comparado con el atributo “instante” de “InstanteTiempo” y si son iguales se ejecutará la acción. Los otros dos atributos de “Accion” son “ejecutor” y “receptor” que indican quién realiza la acción y quién la recibe respectivamente en caso de que existan. Para el caso concreto del LigthCycle, en principio se han definido cuatro acciones, las cuales no van a tener elemento receptor, son acciones de movimiento que indican un cambio de movimiento del jugador que las ejecuta. Las cuatro acciones son: “MovimientoArriba”, “MovimientoAbajo”, “MovimientoIzquierda” y “MovimientoDerecha”.

Finalmente solo queda comentar que todos los elementos heredan de una clase denominada “NamedElement” que contiene un atributo “nombre” y es que todos los elementos del problema necesitan ser identificados en base a su nombre.

4.3 Creación del Metamodelo del LigthCycle.

Para crear el metamodelo del problema del LigthCycle en Eclipse se creó un fichero .ecore. Para ello se selecciona la opción:

New -> Other -> Eclipse Modeling Framework -> Ecore Model

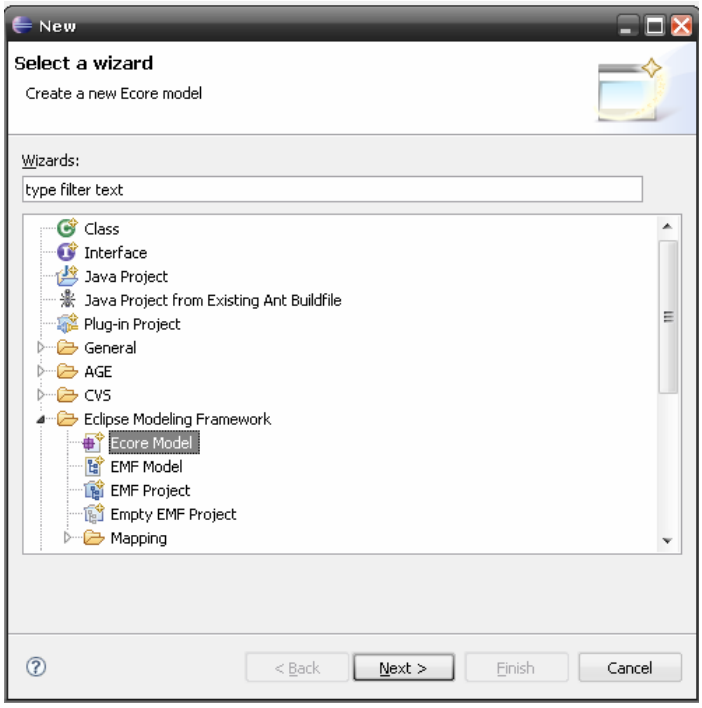


Figura 3. Creación Metamodelo Ecore

Al fichero se le puso de nombre Tron.ecore y se incluyó en la carpeta “metamodelos” del proyecto “VisualizadorEscenas”. Una vez creado el fichero, si se accede a él, lo primero que se tiene que configurar son dos propiedades sin las cuales no se puede crear el metamodelo ya que aparece un error. La primera es el nombre que se le va a poner al metamodelo (se le puso “Tron”) y, en segundo lugar, se le asignó la URI única que debe tener cada modelo Ecore. Para ello se le asignó como prefijo “Tron” y como URI “http://www.example.org/Tron”. Una vez hecho esto ya se está en disposición de crear el metamodelo.

Se creó una clase por cada elemento del metamodelo (las clases son de tipo EClass). Se añade su nombre en las propiedades y, puesto que todas las clases van a heredar de la clase “NamedElement”, se asignó para cada clase en la propiedad “ESuper Types” la clase “NamedElment”. A continuación se crearon los atributos de las clases (EAttribute) y se les asignó su tipo de datos (EType). Para el metamodelo vamos a utilizar tres tipos distintos de datos: EString, EInt y EBoolean. Hay que tener cuidado porque existen para el caso de los enteros y los booleanos unos tipos de datos que son EIntegerObject y EBooleanObject que parecen funcionar bien para almacenar enteros y booleanos respectivamente pero luego dan problemas cuando se trabaja con dichos atributos mediante RubyTL, ya que RubyTL no es capaz de realizar operaciones aritméticas

simples con enteros de este tipo, ni tampoco comparar adecuadamente los valores “true” y “false” utilizando este tipo de booleanos.

Finalmente se crearon las relaciones (EReference). Para las relaciones se deben tener en cuenta varias propiedades. En primer lugar se le asignó un nombre a la relación, a continuación se modificó el atributo EType para determinar con qué elemento se establece la relación. Seguidamente se indicaron los intervalos del rango de la relación mediante los valores LowerBound y UpperBound (el infinito se pone con -1), y finalmente si se trata de una relación de contenido, se marca la casilla Containment.

Como detalles finales para dejar el metamodelo correctamente definido, se indicó que las clases “Objeto”, “Movil”, “Inmovil” y “Accion” son clases abstractas. Para ello simplemente en las propiedades de cada una de dichas clases se seleccionó la opción Abstract.

Otro detalle importante es que la relación entre “Escena” e “InstanteTiempo” es una relación de contenido (como se puede apreciar en el metamodelo mostrado en la Figura 2) a pesar de ser más lógico que fuese una relación normal, ya que una escena no contiene un instante de tiempo, sino que se produce en un instante de tiempo. El hecho de representar esa relación como una relación de contenido es porque al visualizar las escenas de la secuencia final, se quiso que el “reloj”, que es como se va a representar el “InstanteTiempo”, aparezca dentro de la escena y no fuera de la misma. Es por ello que se definió una relación de contenido, pues de lo contrario no se podía visualizar luego la escena como se quería.

El último detalle destacable reside en el tipo de datos de los atributos “ejecutor” y “receptor” de la clase “Accion”. Se podían haber definido como referencias a objetos del problema ya que lo normal es que los actores de las acciones sean entidades de la escena, pero apareció un problema al final del proyecto a la hora de generar los ficheros finales, en concreto el fichero “default” necesario para el visualizador. Dicho fichero, como se verá más adelante, contendrá el modelo final de la secuencia de escenas y de entre todos los elementos que definirá estarán las acciones entre ellos. Si se definen los atributos “ejecutor” y “receptor” como referencias y se visualiza el código del fichero para comprobar cómo las define, se ve que utiliza por ejemplo para el caso del ejecutor la siguiente nomenclatura: ejecutor=”//@escenas.0/@objetos.0”. El problema radica en que para generar ese fichero en el generador de escenas desarrollado, se utilizó una plantilla donde hay un campo a rellenar con el contenido del atributo “receptor” o “ejecutor” en este caso y debería contener la misma nomenclatura, pero RubyTL al leer el modelo de entrada de la plantilla y llegar a dichos atributos no imprime las referencias a objetos de esa forma, de hecho no es capaz de imprimir una referencia, como mucho es capaz de imprimir un número identificativo de la misma, pero en ningún caso la cadena propia del fichero “default” necesaria para que el fichero sepa identificar bien las referencias y saber a qué objetos ejecutor y receptor hacen referencia los atributos. Este hecho supuso definir estos atributos de tipo “string” ya que para el caso de atributos de tipo “string” el fichero “default” no utiliza ninguna sintaxis particular, es simplemente el valor del atributo, consiguiendo con ello poder definir claramente los actores de las acciones escribiendo el nombre de éstos.

A continuación se muestra una imagen orientativa de cómo queda definido el metamodelo inicial del LigthCycle en EMF:

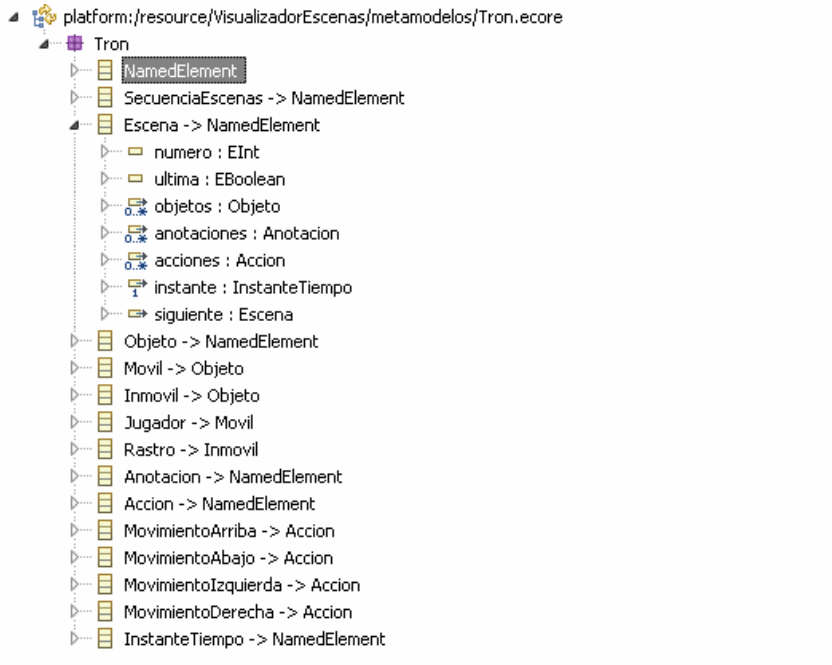


Figura 4. Metamodelo Ecore LigthCycle

Una vez que se terminó de definir el metamodelo, se creó un modelo con la definición de la generación del modelo (fichero .genmodel) para especificar como el framework EMF genera las clases Java que implementan el metamodelo. En realidad, éste es un modelo que "decora" el metamodelo con un número de propiedades extra para personalizar la generación del código del modelo y de la vista de propiedades. La forma de crear dicho fichero es la siguiente: File -> New -> Other -> EMF Model.

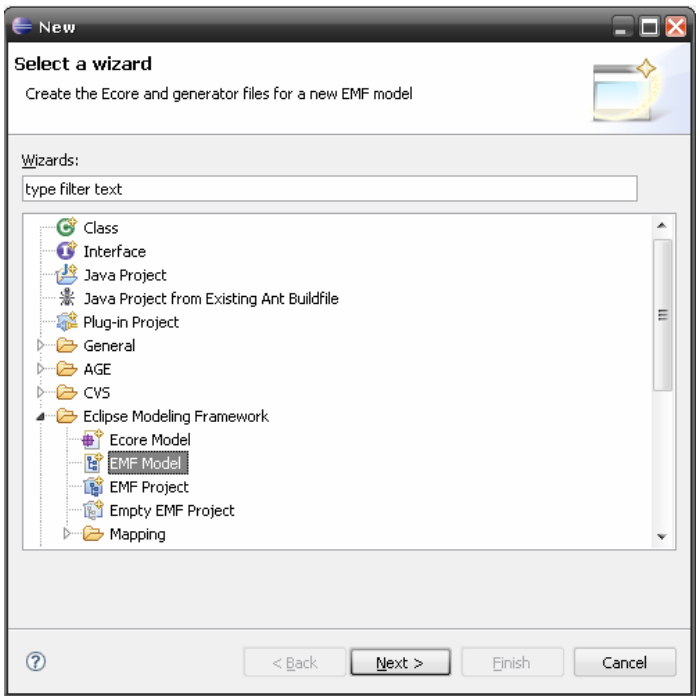


Figura 5. Creación Metamodelo Emf

El fichero se creó con el nombre Tron.genmodel y completa la carpeta “metamodelos” del proyecto “VisualizadorEscenas”, poniendo punto y final a la parte de generación de metamodelos necesarios para el proyecto. Como se ha mencionado anteriormente, este fichero es el que posibilita la generación automática del código Java del metamodelo del LigthCycle. Esto se consiguió simplemente abriendo el metamodelo Tron.genmodel recién creado, seleccionando el paquete Tron (raíz del árbol que aparece) y pinchando en Generate All. Con esto se generaron una serie de clases que permiten el acceso al metamodelo mediante código. Este proceso de generación del código del metamodelo también será necesario en el generador de escenas, en concreto cuando se haga uso de la API de EMF.

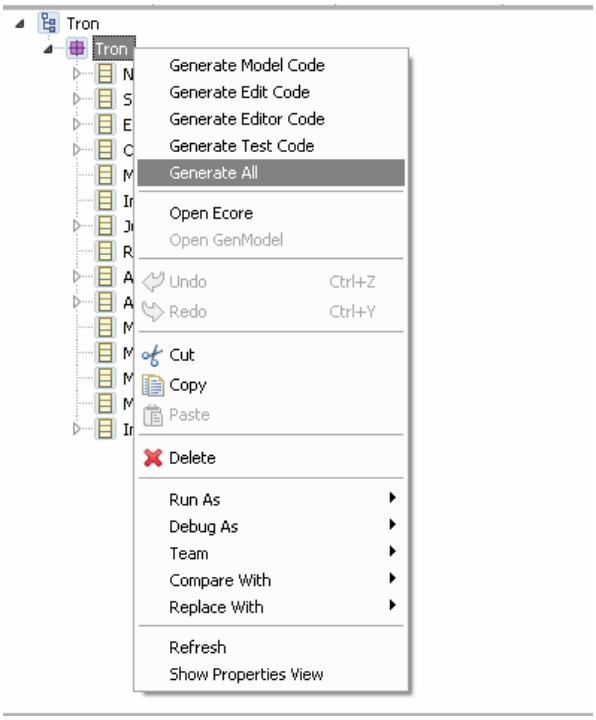


Figura 6. Generación código Java del Metamodelo

A continuación se explica la creación del visualizador propiamente dicho, haciendo uso esta vez de GMF. Durante el proceso se generarán varios ficheros, por lo que se creó en primer lugar en el proyecto una carpeta “visualizador” que contendrá los mismos.

5. Creación del Visualizador de Escenas.

En el siguiente capítulo, se describe la construcción del visualizador de escenas, empleando GMF como tecnología para el desarrollo de un visualizador/editor gráfico de escenas.

5.1 Introducción GMF.

Graphical Modeling Framework (GMF) es un framework de la plataforma Eclipse que permite dotar a los metamodelos descritos en Ecore de un mecanismo de edición gráfica. Para ello utiliza dos frameworks: el framework GEF como base para la parte diagramática y EMF (visto anteriormente) para gestionar los modelos y metamodelos. Ofrece, por lo tanto, una interoperabilidad entre EMF y GEF para la construcción de editores gráficos, así que puede decirse que GMF es un puente entre EMF y GEF.

El framework EMF ofrece la infraestructura necesaria para guiar la construcción del editor por medio de modelos como veremos a continuación. Este framework no sólo es utilizado para definir el metamodelo del editor gráfico que se quiere construir con GMF, sino que también es utilizado por el propio GMF para describir un conjunto de modelos necesarios para definir las diferentes partes que componen el editor a generar.

La parte relativa a la interfaz de usuario es aportada por el framework GEF que facilita un marco de trabajo para construir cualquier tipo de editores. Graphical Editing Framework (GEF) es un plug-in para Eclipse que permite el desarrollo de representaciones gráficas para un modelo. Los editores permiten modificar el modelo mediante funciones gráficas comunes como arrastrar y soltar, copiar y pegar, y acciones invocadas desde menús y/o barras de herramientas de Eclipse. Antes de la existencia de GMF la construcción de editores gráficos en Eclipse debía hacerse manualmente con GEF, lo cual era bastante tedioso.

Como se ha comentado anteriormente el desarrollo de editores gráficos con GMF se realiza mediante la definición de diversos modelos. Los modelos de GMF están descritos por el metamodelo Ecore y se ofrece un conjunto de herramientas y asistentes para editarlos. A continuación se describen brevemente:

- **Modelo de dominio (.ecore)**: también conocido como modelo semántico, es el punto de partida del editor. Es un metamodelo Ecore que describe la capa del modelo de la aplicación y actúa de metamodelo del modelo que el usuario podrá editar gráficamente.

- **Modelo de definición gráfica (.gmfgraph)**: contiene la información relativa a los elementos gráficos que aparecerán en el editor. En él se definen los elementos a representar (nodos, enlaces, etiquetas de texto, etc) y la galería de figuras (rectángulos, círculos, flechas, etc) asignadas para la representación de los mismos.

- **Modelo de herramientas (.gmftool)**: define la paleta de herramientas de nuestro editor gráfico con el fin de permitir al usuario añadir elementos al modelo. Además ofrece la posibilidad de asignar iconos a los elementos de nuestro editor.

- **Modelo de mapping (.gmfmap)**: relaciona el modelo de dominio con el modelo gráfico y el modelo de las herramientas. Es el modelo más importante del desarrollo en GMF pues en él se aúnan las definiciones de los demás modelos y se da lugar al modelo generador de código de GMF. Además de almacenar las correspondencias entre los elementos del modelo y los elementos de la vista, permite definir una serie de atributos para caracterizar el editor gráfico a generar.

- **Modelo generador (.gmfgen)**: Una vez definidos los modelos anteriores, GMF construye un modelo generador a partir del modelo de mapeo. Es un modelo que extiende al modelo de mapping para especificar los parámetros de la generación automática de código

Además de estos modelos básicos hay que comentar que un metamodelo Ecore no ofrece todos los requisitos que exige el framework GEF al modelo. En concreto, los aspectos como la posición, el tamaño de los elemento en el editor se escapan de la definición semántica. Para suplir esta carencia GMF introduce un modelo para describir las características de los diagramas llamado modelo de notación. Esta información es usada para representar y persistir los elementos visuales en los diagramas. Este modelo de notación es el que se genera en el fichero “default_diagram” y que será junto con el fichero “default” que almacena el modelo semántico, los ficheros necesarios para visualizar la representación gráfica de la secuencia de escenas.

Como se ha expuesto, el desarrollo en GMF conlleva la utilización de diferentes herramientas ofrecidas por el framework para la construcción de una serie de modelos que describan las características del editor gráfico a construir. Exactamente los pasos a seguir para el desarrollo son:

1. Crear el modelo de dominio mediante un metamodelo Ecore.

2. Crear un modelo de definición gráfica.

3. Crear el modelo de las herramientas.

4. Crear un modelo de mapeo.

5. Generar el editor gráfico. Este paso es ofrecido por el framework GMF y se realiza de forma automática ya que durante el proceso se obtiene el modelo generador.

6. Mejorar el editor gráfico modificando su código. Este paso se deja libre al desarrollador para personalizar los resultados ofrecidos por GMF en el paso anterior.

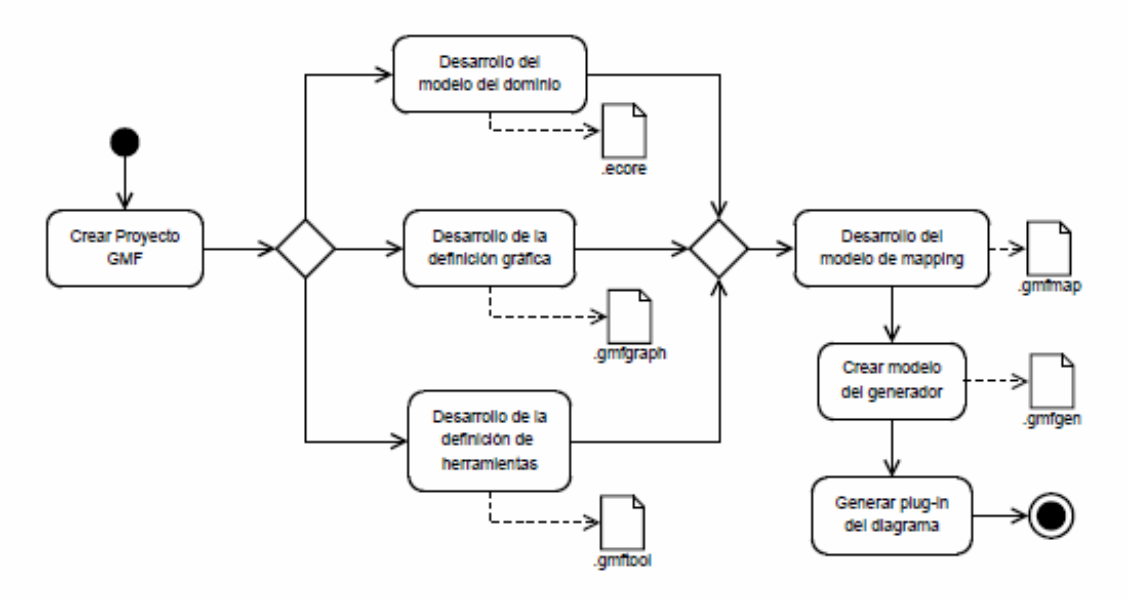


Figura 7. Esquema de Creación del Editor de Escenas

La creación de los modelos del dominio, de definición gráfica y de herramienta se puede efectuar en paralelo. Una vez que se tienen estos tres modelos se puede crear la definición del mapping. Luego, a partir de la definición del mapping se genera la definición del generador del editor. Por último, se genera el plug-in con el editor gráfico a partir de la información del generador del modelo. Esta secuencia se muestra en la Figura 7. En realidad, el desarrollo de una aplicación con GMF no se realiza de una forma estrictamente secuencial, sino que está sujeta a numerosas realimentaciones y modificaciones entre los modelos.

5.2 Visualizador de Escenas del LigthCycle.

En la introducción previa que se ha hecho de GMF se ha comentado que su objetivo es la creación de editores gráficos para generar modelos en base a un metamodelo de partida de una manera sencilla. Esta idea es cierta, pero en este caso se le ha dado un nuevo enfoque al propósito de generar un editor.

Más que un editor gráfico con el que ir generando modelos que representen las escenas del LigthCycle, lo que se va a generar es un visualizador de escenas del LigthCycle, es decir, como se verá más adelante los modelos de las escenas del LigthCycle se generarán automáticamente (mediante RubyTL) y serán esos modelos ya generados los que se le pasarán al visualizador para que los muestre de forma gráfica, con el fin de no tener que ir generando una a una las escenas que compongan el dominio del problema.

Por tanto, aunque es cierto que lo que se va a generar es un editor gráfico porque se puede utilizar como tal, el uso real que se le va a dar, sin embargo, es el de un visualizador de modelos o más concretamente de escenas del LigthCycle. Es cierto no obstante, que para saber qué tipo de modelos hay que generar, de tal forma que luego puedan ser visualizados, se realizaron una serie de pruebas previamente usando el editor

de escenas como tal, con el fin de generar distintos tipos de escenas y entender que modelos iba generando el editor internamente.

Una vez entendido el propósito del visualizador, se van a seguir los pasos descritos en la introducción previa de GMF para poder generarlo.

5.2.1 Modelo de Definición Gráfica.

El primer paso fue crear un modelo gráfico a partir del modelo Ecore que se generó con EMF. Para ello se seleccionó la opción: File -> New -> Other -> Simple Graphical Definition Model.

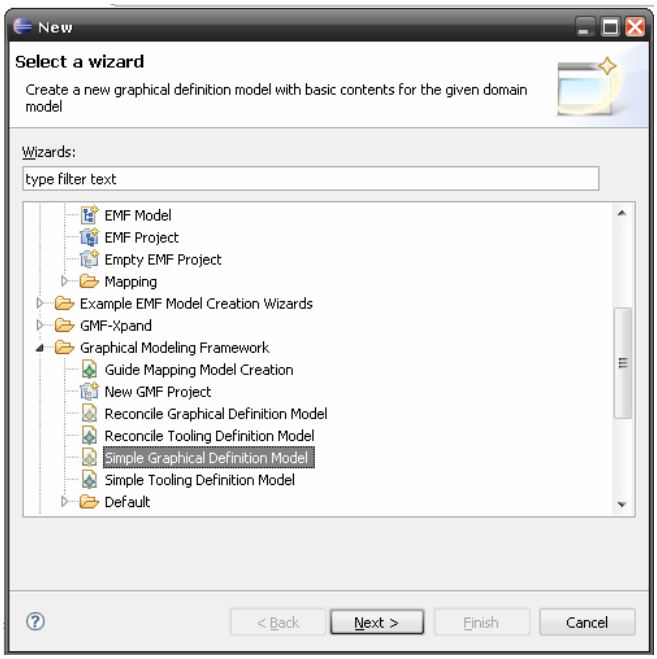


Figura 8. Creación del Modelo de Definición Gráfica

Esto abre un asistente donde se deben seleccionar aquellos elementos del metamodelo Ecore que se quiere que tengan una representación gráfica. En concreto para el problema del LigthCycle que se está desarrollando no se van a querer mostrar todos los elementos que se han definido, con lo que únicamente se seleccionaron aquellos elementos que se quisieron ver en la escena. Por ejemplo, no se quiere ver el elemento "NamedElment", ni la "SecuenciaEscenas", ya que esto viene implícito viendo las escenas (que sí se marcaron, junto a su atributo número, ya que se quiere ver en qué escena se está) y sus enlaces "siguiente" con la siguiente escena. Como se puede ver, las clases marcadas como abstractas en el metamodelo inicial no se pueden mostrar (es obvio), pero sí aquellas clases concretas que heredan de éstas. En este caso las clases "Objeto", "Movil" e "Inmovil" al ser abstractas no se mostrarán, pero las clases "Jugador" y "Rastro" que heredan de ellas sí que se pueden mostrar y de hecho así se hizo. De éstas además se quiere ver únicamente su atributo "icono" que, como ya se comentó, es una "excusa" para poder ver representados los objetos con un icono. El problema es que aunque se seleccione que se quiere representar una clase, si esa clase

no contiene ningún atributo a representar de ella, no se le podrá asignar ningún icono, ya que el icono solo aparece al mostrar los atributos de la clase. Se podría decir que los iconos van ligados a los atributos y por cada atributo aparece un icono. De esta forma a toda clase que se quiere visualizar con un icono se le tiene que asignar solo un atributo (si no, se duplicaría el icono) y además debe ser un atributo donde no se le vaya a dar ningún valor, porque si no aparecería por pantalla. Por ello se definió el atributo "icono". Las anotaciones también se mostrarán por si ocurre algo a lo largo del problema, para que se vea un mensaje por pantalla, y de ellas solo se mostrará su atributo "texto". Lo mismo ocurre con el "InstanteTiempo" que se representará con un reloj, y se mostrará su atributo "instante" para saber en qué instante se está desarrollando la escena del problema. Y finalmente las acciones no se van a representar, ya que las sucesivas escenas irán mostrando lo que va ocurriendo en cada momento.

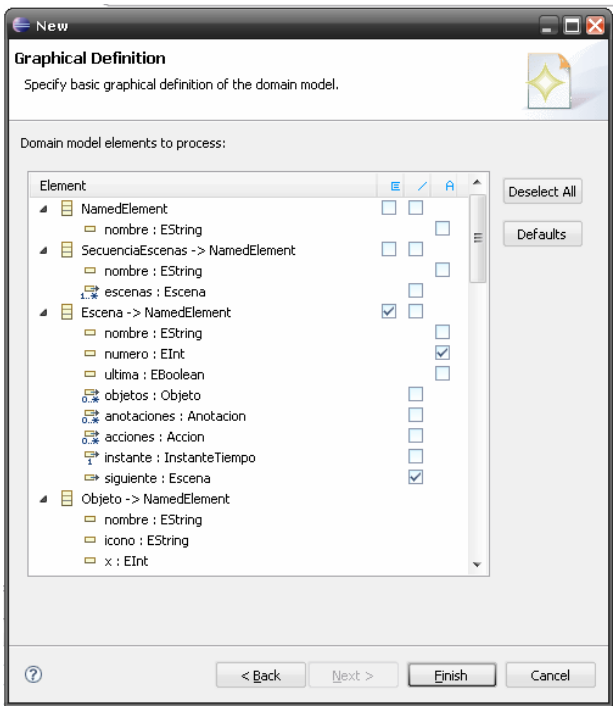


Figura 9. Asistente Modelo de Definición Gráfica

El modelo gráfico generado tendrá una extensión .gmfgraph. Y se le tuvieron que hacer una serie de modificaciones para que la representación gráfica quedase lo más parecida posible a la del dominio del problema. Como se puede observar, el modelo tiene una raíz Canvas de la cual se derivan cuatro tipos de hijos. Por un lado, están los elementos "Node": existe un nodo por cada una de las clases que se seleccionaron para mostrar en el visualizador de escenas. En segundo lugar, aparecen los elementos "Connection", que representan los enlaces que se indicaron que se querían representar (en este caso, solo aparece uno, que es el enlace "siguiente" de las escenas). Y al final están los elementos "Diagram Label", que son los atributos que se indicaron anteriormente que se querían mostrar. Se ha dejado para el final el primer elemento que aparece en el modelo, "Figure Gallery Default", que contiene la representación que se le va a dar a todos los elementos comentados (nodos, conectores y etiquetas).

El primer cambio que se hizo fue añadir un nuevo tipo de elementos debajo de la raíz Canvas. En primer lugar se definió que ciertos objetos van a contener a otros en la representación. En concreto, la escena va a contener una serie de elementos, los cuales estarán repartidos en dos compartimentos. Un primer compartimento donde se mostrará la escena en sí, con los jugadores, los rastros y el reloj, y un segundo compartimento justo debajo donde aparecerán las anotaciones si las hubiese, informando de algún hecho que se ha producido en la partida como el final de la misma. Tomando como ejemplo el compartimento primero se llevaron a cabo los siguientes pasos:

- Se creó un hijo de tipo “Compartment” en la raíz (Canvas).

- Se editó la propiedad Figure con el valor “Figure Descriptor EscenaFigure” para indicar que la representación gráfica que corresponde a la Escena dispondrá de un compartimento.

- Finalmente se estableció la propiedad Name con el valor “EscenaObjetoCompartmentNode”. En el caso del segundo compartimento se le dio de nombre “EscenaAnotacionCompartmentNode”. Se indicó en el nombre en primer lugar la clase contenedora y en segundo lugar la clase contenida.

Con esto ya se tiene la distribución de los objetos definida en la escena. Lo que falta a continuación es modificar la apariencia de los elementos a representar. Para ello dentro de Figure Gallery Default se modificaron los descriptores de figura de los distintos elementos. Cosas que se modificaron:

- *Escena*
  - Background Color, Constant Color: darkGrey
  - Preferred Size Dimension: [350, 350]

- *Anotacion*
  - Background Color, Constant Color: darkGrey

- *Jugador*
  - Background Color, Constant Color: darkGrey
  - Preferred Size Dimension: [32, 32] (tamaño del icono)
  - Line Border, atributo Width 1 (Color darkGrey)
  - LabelJugadorIconoFigure, atributo Text lo ponemos en blanco para que no aparezca nada en pantalla.

- *Rastro*
  - Todo igual que jugador salvo que el tamaño es del icono [16,16]

- *InstanteTiempo*
  - Background Color, Constant Color: darkGrey
  - Line Border, atributo Width 1 (Color darkGrey)

En GMF se les asigna por defecto a todas las figuras una representación basada en rectángulos en blanco. En este aspecto GMF está algo limitado porque no ofrece muchas posibilidades de representación, ya que solo permite rectángulos, rectángulos redondeados y elipses, aunque como ya se ha comentado, sí permite definirle un icono a los atributos de los objetos. Esto obligó a maquillar este inconveniente para poder visualizar algunos objetos como iconos.

Básicamente lo que se hizo con estas características que se modificaron es darle en primer lugar a todos los elementos un color de fondo igual con la propiedad "Background Color" y un color de línea de los rectángulos igual al color de fondo con la propiedad Line Border, con ello se consigue no ver esos rectángulos que representan a los elementos. En segundo lugar lo que se hizo es prefijar el tamaño de esos rectángulos con la opción "Preferred Size Dimension" para el caso del Jugador y del Rastro, asignándoles el mismo tamaño que tendrán los iconos que se les van a poner a los elementos, con el fin de ocupar el rectángulo al completo y tener la sensación de que el rectángulo no existe, que solo existe el icono. Y finalmente se dejaron las etiquetas "icono" de Jugador y Rastro en blanco para que no aparezca ningún texto junto al icono.

Una imagen resumida de lo que es el modelo se muestra a continuación:

Figura 10. Modelo de Definición Gráfica

5.2.2 Modelo de Herramientas.

Una vez que se ha definido el modelo gráfico, el siguiente paso es definir el modelo de herramientas. Con este modelo simplemente se va a indicar que elementos se quieren tener disponibles en la paleta de herramientas con el fin de poder incluirlos posteriormente en el editor de escenas. Para iniciar su creación se seleccionó la opción: File -> New -> Other -> Simple Tooling Definition Model

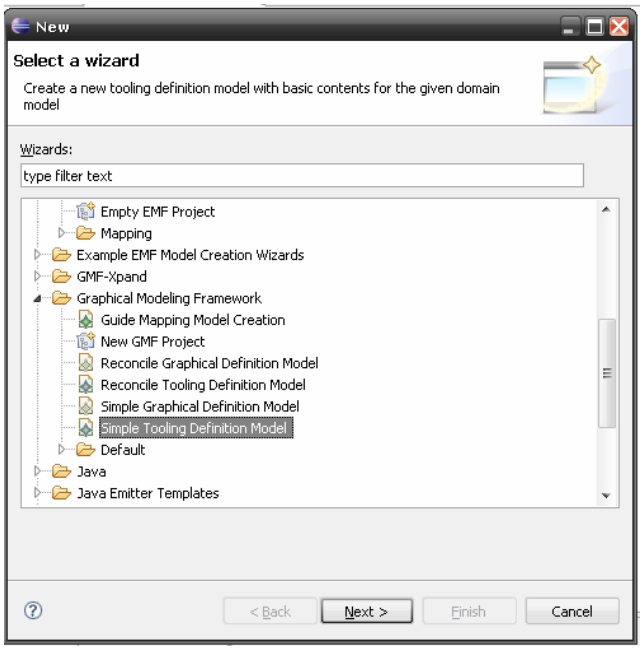


Figura 11. Creación Modelo de Herramientas

De nuevo como en el caso del modelo gráfico aparece una nueva ventana con un asistente donde se deben seleccionar los elementos que se quieren incluir en la paleta. Los elementos que se seleccionaron fueron: Escena, el enlace a escena siguiente, Jugador, Rastro, Anotacion e InstanteTiempo.

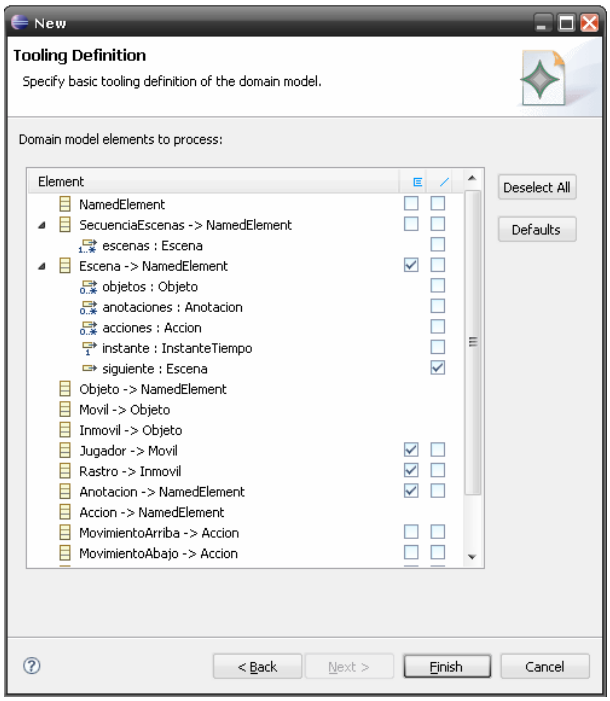


Figura 12. Asistente Modelo de Herramientas

El modelo generado tendrá una extensión ".gmftool" y contendrá los elementos que han sido seleccionados. Si se inspecciona cualquiera de los elementos creados por el asistente, se puede ver que contienen dos elementos "DefaultImage" cada uno. Esto quiere decir que el icono de cada uno de los elementos, tanto el de la paleta como el que aparecerá posteriormente cuando se inserte el elemento en el editor, va a ser un icono por defecto. Existe la posibilidad de asignar el icono que se quiera tanto en la paleta como en el diagrama, para ello se deben borrar estos elementos por defecto y crear los elementos "Small Icon Bundle Image" y "Large Icon Bundle Image" para poner el icono que se quiera en ambos sitios respectivamente, indicando tan solo la dirección del icono. Sin embargo, por más que se probó no se consiguió de esta manera. Por tanto, lo que se hizo fue asignar los iconos indicándolos directamente en el código que se genera una vez que está creado el visualizador.

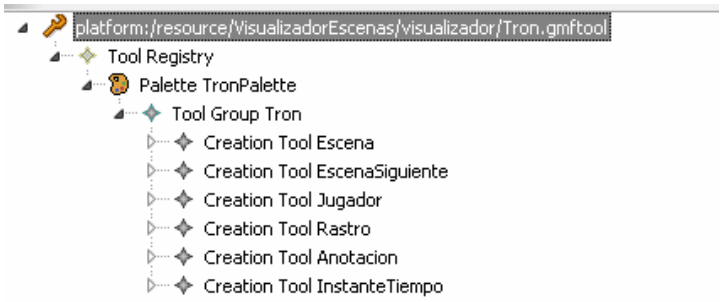


Figura 13. Modelo de Herramientas

5.2.3 Modelo de Mapping

El siguiente modelo a generar es el modelo de mapping. Este modelo lo que pretende es enlazar el metamodelo Ecore inicial, el modelo gráfico y el modelo de herramientas. Para generarlo se seleccionó la opción: File -> New -> Other ->Guide Mapping Model Creation

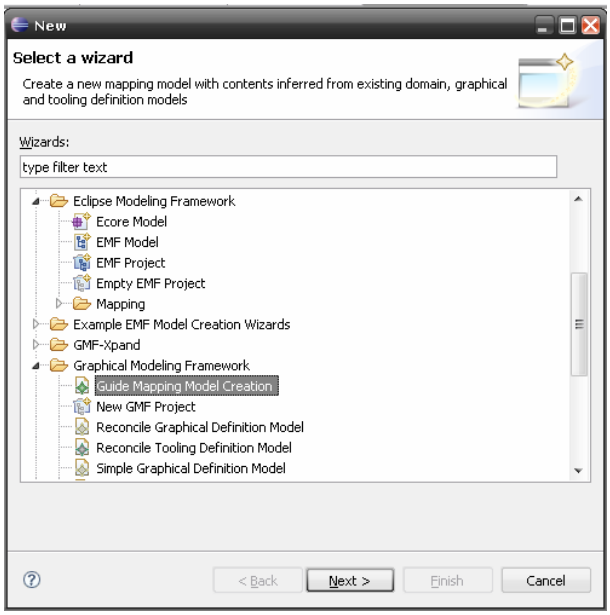


Figura 14. Creación Modelo de Mapping

Una vez seleccionada la ruta de los distintos modelos que va pidiendo el asistente, se llegó a una ventana donde se debe indicar qué elementos del metamodelo son clases y cuáles por su parte son enlaces, además de mostrar algunas propiedades de los mismos. Este punto no es muy crucial para el caso concreto que se está desarrollando, puesto que el hecho de tener compartimentos definidos en el modelo gráfico va a suponer construir este modelo casi desde cero una vez generado, ya que el asistente no es capaz de generar bien los compartimentos.

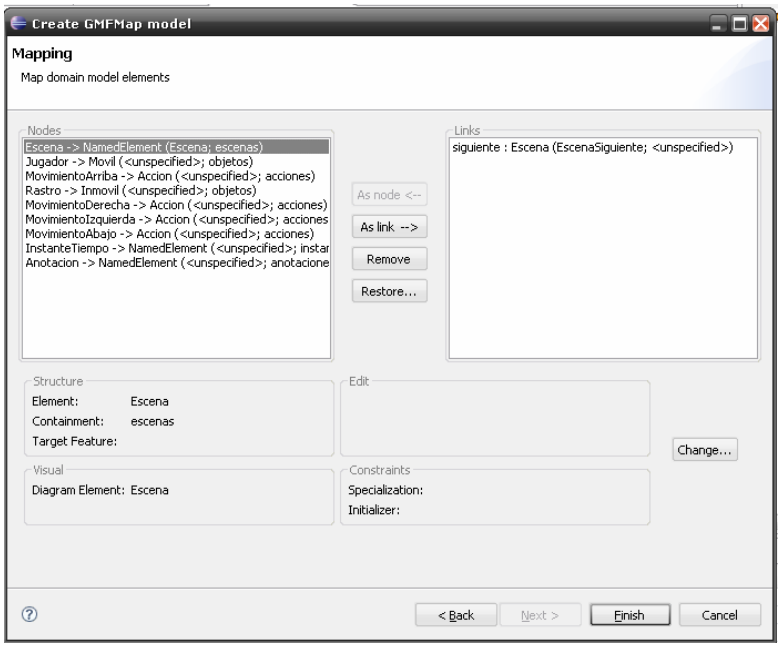


Figura 15. Asistente Modelo de Mapping

El asistente crea un modelo ".gmfmap", el cual se tuvo que volver a crear casi desde cero. En primer lugar se borraron todos los elementos creados salvo el "Top Node Reference" correspondiente a Escena y el "Link Mapping" correspondiente a Escena siguiente. Dado que la escena va a contener los elementos Jugador, Rastro, InstanteTiempo y Anotacion se fueron creando uno a uno dichos elementos dentro de la escena:

- En primer lugar se creó dentro del Node Mapping Escena un elemento Child Reference por cada uno de los cuatro elementos. Y dentro de cada uno se creó un Node Mapping. En cada Node Mapping se tienen que editar los valores de las propiedades: Element (el elemento del modelo que se mapea), Diagram Node (el elemento gráfico que se mapea) y Tool (la herramienta de la paleta que lo crea).

- Una vez definidas las propiedades del Node Mapping se cambió una propiedad del objeto Child Reference que lo contiene. En primer lugar se verificó que el atributo Child hace referencia a su propio Node Mapping y seguidamente se estableció la propiedad Containment Feature con el nombre de la relación que asocia el elemento Escena con este elemento.

- A continuación se deben crear tantas etiquetas dentro de cada Node Mapping como atributos se hayan querido representar en el editor por cada elemento. Para

ello se creó dentro de cada Node Mapping hijos del tipo Feature Label Mapping. Dos son las propiedades que se establecieron por cada etiqueta: Features, la cual identifica el atributo y Diagram Label, que identifica su representación.

- Finalmente se tuvieron en cuenta los elementos que contienen a otros. Para ello se creó un hijo de tipo Compartment Mapping del nodo que va a contener a otros. En el caso particular que se está desarrollando este elemento contenedor es Escena que, de hecho, va a tener como ya se comentó dos compartimentos. Una vez creado se configuró su propiedad Compartment con el nombre que se utilizó en el modelo gráfico y la propiedad Children con los elementos que va a contener. En este punto surgió un pequeño problema que hizo variar el metamodelo inicial: si se ve el metamodelo, el enlace que une Escena con InstanteTiempo es un enlace de contenido, lo cual no tiene mucho sentido, ya que una escena no contiene un instante de tiempo, sino que se desarrolla en un instante de tiempo. Esto supuso que en un principio esa relación del metamodelo fuese una relación normal, pero en este punto justo del modelo del mapping, solo permitía el modelo incluir como elementos Children del elemento Compartment Mapping de Escena elementos que aparezcan contenidos dentro de Escena en el metamodelo inicial, con lo cual se tuvo que definir la relación entre Escena e InstanteTiempo como una relación de contenido.

Una vez definidos los nodos, se comprobaron los enlaces. Hay que tener un elemento Link Mapping para enlazar las escenas. Los valores que toman especial atención en este punto son: Target Feature (establece la relación del modelo), Diagram Link (elemento gráfico que se mapea) y Tool (la herramienta de la paleta que la crea). Por último, se comprobaron el resto de elementos mapeados automáticamente siguiendo la misma filosofía, el único elemento restante creado por el asistente es el nodo “Top Node Reference”.

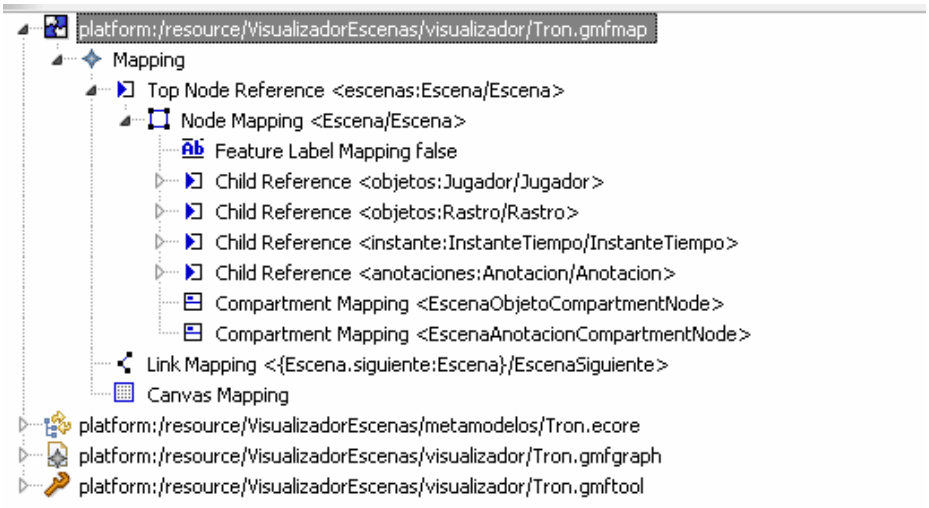


Figura 16. Modelo de Mapping

5.2.4 Modelo generador.

El único modelo que falta por crear es el modelo generador. Este modelo se generó automáticamente. Para ello, se pulsó el botón derecho del ratón sobre el modelo de mapping recién creado y se seleccionó la opción “Create generator model…”. Esto generará un fichero de tipo .gmfgen.

GMF ofrece un asistente que crea dicho modelo a partir del modelo de mapping. En el asistente se puede elegir si se quiere crear el editor como un plug-in Eclipse o como una aplicación rica de la plataforma (RCP Application). La diferencia reside en que la segunda opción genera una aplicación independiente con el editor gráfico, es decir, una aplicación basada en la plataforma Eclipse pero sin incluir la funcionalidad de Eclipse.

Una vez generado el modelo se hizo una modificación para que los objetos de la escena se puedan mover libremente en ella. En el modelo recién creado, se desplegó el GenDiagram y en el GenCompartment, es decir, en el compartimento donde se quiere que los elementos que contiene la escena se puedan mover libremente se seleccionó y se estableció la opción ListLayout a “false”.

Hay un último problema que también hay que solucionar y es que el .gmfgen pone en los paquetes que va a generar un prefijo con el nombre del proyecto que se está utilizando. Esto es un problema ya que a la hora de generar posteriormente el código del editor va a provocar bastantes errores en el código. Para solucionarlo se seleccionó el elemento raíz del modelo generador (Gen Editor) y en la propiedad Package Name Prefix se eliminó el nombre del proyecto. Solucionado el problema, ya se está en disposición de generar el visualizador de escenas.

5.3 Generación del Plug-in.

Una vez definidos todos los modelos necesarios, para generar el plug-in simplemente se pinchó con el botón derecho del ratón sobre el fichero Tron.gmfgen recién creado y se seleccionó la opción “Generate diagram code”. Esto generó cuatro proyectos: .diagram, .edit, .editor y un .tests, de los cuales solo interesan los dos primeros para el problema.

Con esto ya estaría el Visualizador de Escenas generado. Antes de ejecutarlo se tuvo que hacer una última modificación: establecer los iconos que se querían para los elementos, que como ya se comentó, se hizo modificando el código generado. Los iconos se guardan dentro del workspace del proyecto y más concretamente dentro del proyecto .edit que se acaba de generar automáticamente. Este proyecto contiene la carpeta ”icons” y dentro de ella se encuentra la carpeta “full\obj16” que es donde se encuentran los iconos que ha asignado GMF automáticamente. Simplemente basta con depositar en esa carpeta los nuevos iconos. Sabiendo esto se pueden hacer dos cosas, o bien se reemplazan los que hay en la carpeta utilizando el mismo nombre, o por el contrario, se copian los iconos a la carpeta con el nombre y la extensión que se quiera, se va a Eclipse de nuevo, se accede al proyecto .edit y, si se abre la carpeta “src”, se puede observar cómo contiene un fichero .java por cada elemento del metamodelo (estos ficheros se nombran con el sufijo ItemProvider). A continuación se accede al fichero del elemento cuyo icono se quiere modificar (por ejemplo, en el caso del

jugador se accedería al fichero “JugadorItemProvider.java”). Todos estos ficheros contienen un método público en su interior de nombre “getImage”. En el interior de dicho método se debe indicar el nombre del icono que se quiere utilizar. Como se ve en el fichero, la ruta del icono se especifica a partir de la carpeta “full/obj16”, que es donde se ha indicado que hay que ponerlo. La sintaxis del método es la siguiente:

```
public Object getImage(Object object) {
      return overlayImage(object,
            getResourceLocator().getImage("full/obj16/coche.gif"));
}
```

Como se puede ver, el icono que se asignó a los jugadores es “coche.gif”. Lo mismo se hizo por cada elemento al que se quiso modificar su icono (InstanteTiempo, Rastro y Anotacion).

Para probar el Visualizador de Escenas basta con lanzar una ejecución del proyecto y seleccionar que se abra en una nueva instancia de Eclipse. Se creó un nuevo proyecto Java normal y una vez creado, se creó un fichero de tipo Other -> Examples -> Tron Diagram. Se crearon dos ficheros, un fichero default.nombre que es el modelo que se va a crear. Y un fichero default.nombre_diagram que es la representación gráfica del modelo (modelo de notación). Estos son los ficheros que posteriormente se verán cómo generarlos de forma automática desde el generador y que serán pasados al visualizador para mostrar una secuencia de escenas del LigthCycle. Pero, aun así, se puede generar manualmente una secuencia de escenas con el editor recién generado para probarlo.

6. Creación del Generador de Escenas.

En el siguiente capítulo se describe como se construyó el generador de escenas, usando transformaciones entre escenas, soportadas por un lenguaje de transformación de modelos como RubyTL.

6.1 Introducción RubyTL.

RubyTL es un lenguaje de transformación basado en reglas, que ha sido construido como un lenguaje embebido dentro de Ruby. Es un lenguaje de transformación híbrido, cuya parte declarativa está basada en bindings, mientras que la parte imperativa viene dada por los constructores proporcionados por el propio Ruby.

Transformaciones Modelo-Modelo.

Una transformación de modelos es un proceso mediante el cual uno o más modelos origen son transformados en uno o más modelos destino. Para que una transformación de modelos pueda realizarse debe existir correspondencias entre los elementos del metamodelo origen y del metamodelo destino. Estas correspondencias se denominan mapping, y se expresan a nivel de los metamodelos origen y destino. Los lenguajes de transformación de modelos permiten especificar el mapping utilizando reglas de transformación. Estas reglas normalmente especifican cómo se relacionan las instancias de una determinada metaclase (del modelo origen) con instancias de otra metaclase (del modelo destino).

Se puede apreciar en la siguiente figura la sintaxis abstracta de RubyTL expresada como un metamodelo.

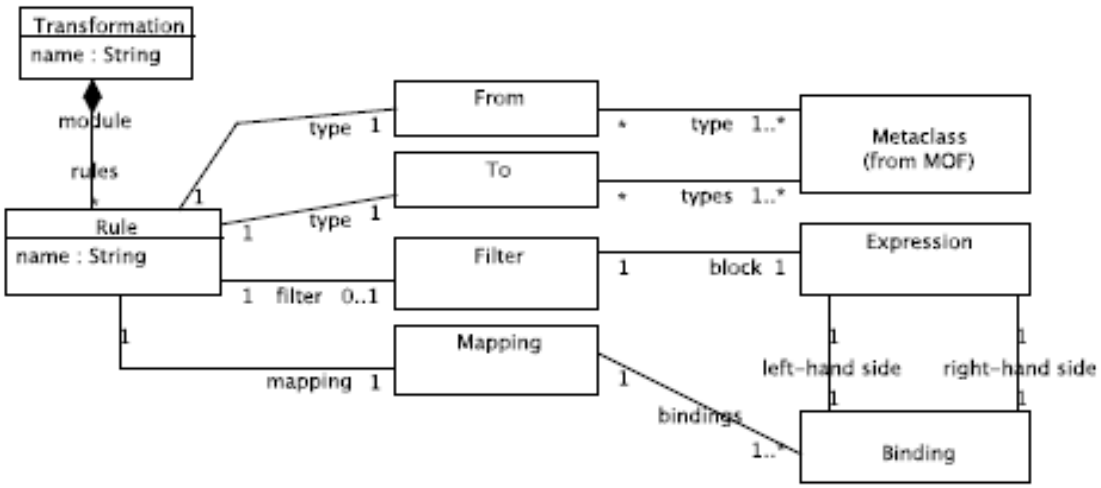


Figura 17. Sintaxis abstracta RubyTl

Como se puede observar, una definición de una transformación en RubyTL está formada por un conjunto de reglas de transformación empaquetadas en un módulo de transformación. Cada regla presenta un nombre y cuatro partes:

- from. Especifica una metaclase origen, que pertenece a un metamodelo origen.

- to. Especifica una o más metaclases destino, que pertenecen a un metamodelo destino.

- filter. Especifica una condición sobre el elemento origen. La regla solo se ejecuta si se satisface la condición. Esta parte es opcional y la regla siempre se lanzará si no presenta ningún filtro.

- mapping. Un bloque de código de Ruby, que se ejecuta cuando se aplica la regla. El bloque recibe un elemento origen y el elemento destino creado por la regla. Dentro del bloque es posible escribir cualquier código Ruby, aunque se recomienda un estilo declarativo basado en bindings.

El bloque de código de la parte mapping de una regla tiene un elemento origen y un elemento destino como parámetros. El primer parámetro es un elemento origen que conforma a la metaclase especificada en la parte from. El resto de parámetros corresponden a elementos destino que conforman con las metaclases destino especificadas en la parte to (en el mismo orden).

En la siguiente figura se puede observar la sintaxis concreta de RubyTL.

```
module <module-name> do

  rule <rule-name> do
    from <source-metaclass>
    to   {target-metaclass}

    filter do |source_element|
      <expression>
    end

    mapping do |<source_element>, {target_element}|
      {bindings}
      # bindings has the form:
      #     target_element.property = source_element.property
    end
  end

  # one or more rules
end
```

Figura 18. Sintaxis concreta RubyTl

Puede observarse que la sintaxis de RubyTL tiene algunas peculiaridades. Esto es debido a que RubyTL ha sido creado como un lenguaje embebido en Ruby, por lo que su sintaxis está, hasta cierto punto, determinada por la sintaxis del propio Ruby. Así, las estructuras del lenguaje que tienen elementos internos, como por ejemplo *rule*, *filter* o *mapping*, utilizan bloques de código como mecanismo de anidamiento. Un bloque de código Ruby es un trozo de código encerrado entre *'do – end'* o *'{ }'*. Los parámetros del bloque se especifican con '| |'. Es conveniente tener en cuenta la peculiaridad de que RubyTL es embebido, ya que supone tanto una ventaja como una desventaja.

- Existen restricciones relativas impuestas por la naturaleza del lenguaje. La restricción relativa a la sintaxis más destacable es que los nombres de las reglas deben escribirse entre comillas (simples o dobles), y la forma de especificar los bloques de código.

- Esta característica puede ser aprovechada para hacer uso de todas las construcciones imperativas de Ruby y de sus librerías, esto es, cualquier facilidad o librería de Ruby puede ser utilizada en RubyTL. Por ejemplo, es posible utilizar expresiones regulares dentro de una transformación.

Bindings.

La parte mapping de una regla está compuesta por un conjunto de bindings. El propósito de un binding es especificar una correspondencia entre elementos del modelo origen y elementos del modelo destino. Se escribe como una asignación de la forma *destino.propiedad = expresión-origen*, donde:

- *expresión-origen* es una expresión cuyo resultado es un elemento o una colección de elementos, que deben pertenecer al modelo origen. El tipo de la parte derecha viene dado por el tipo de la expresión. Si el resultado de la expresión es una colección, RubyTL iterará automáticamente sobre la misma al asignarla a la parte izquierda.

- *destino* es un parámetro del bloque de código del mapping que representa un elemento destino. Su tipo viene dado en la parte to de la regla.

- *propiedad* debe ser una propiedad de un elemento destino. El tipo de la parte izquierda de la asignación viene dado por el tipo de tal propiedad.

Es importante remarcar que el operador de binding (esto es, =) no tiene a la misma semántica que la asignación normal. En particular, si la propiedad destino es multivaluada (es una colección), entonces la semántica es "transformar y añadir".

Decoradores.

RubyTL permite extender las metaclases de un metamodelo con métodos de utilidad para cierta transformación. Esto es muy útil para hacer las transformaciones más legibles, normalmente factorizando el código de navegación en el decorador.

Un decorador está compuesto de uno o más métodos Ruby (especificados con *'def methodname – end'*). Los métodos podrán ser invocados por cualquier instancia de la metaclase o de cualquiera de sus subclases. Se pueden utilizar variables de instancia dentro de los decoradores pero no es una práctica recomendada.

Es una práctica recomendada escribir los decoradores antes de las reglas de transformación. También, es posible factorizar los decoradores en librerías. Las librerías se suelen guardar en el directorio *helpers*. Una librería es un fichero que contiene decoradores, y cuya extensión es *'.rb'*. Se utiliza la sentencia *'use library'* para cargar una librería desde una transformación. La sentencia *'use library'* toma una cadena que especifica la ruta de la librería como una URI. En particular, la URI *'helper:///'* busca directamente en la carpeta *helpers* del proyecto.

Reglas.

En RubyTL hay varios tipos de reglas, y cada una proporciona una funcionalidad distinta para tratar ciertos problemas de transformación. La siguiente figura muestra sus relaciones mediante una jerarquía de herencia. También se muestra la palabra clave que se usa para declarar una regla de ese tipo.

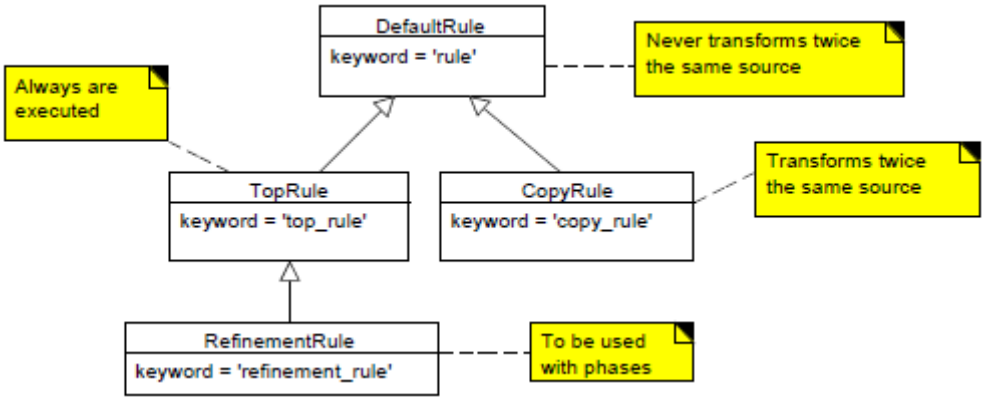


Figura 19. Tipos de Reglas en RubyTl

Default Rule.

La *regla por defecto* se nombra simplemente con rule. Tienen *from*, *to*, *filter* y *mapping* como se describe arriba. Una regla por defecto se ejecuta solo en las dos siguientes situaciones:

- Cuando la regla conforma a un *binding* y su filtro es satisfecho por la instancia origen. Se dice que la regla se ejecuta para resolver el *binding*.

- Cuando es la primera regla de una transformación y la transformación no tiene ninguna otra regla top.

La principal característica diferenciadora de este tipo de regla es que la misma regla nunca transforma dos veces el mismo elemento origen. Esto significa que si la regla es llamada por segunda vez (para resolver otro *binding*), se devuelve el resultado de la aplicación anterior.

Éste tipo de regla resuelve problemas de recursividad no transformando dos veces el mismo elemento fuente.

Top Rule.

Una regla top es una especialización del tipo de regla por defecto. Al igual que ésta, nunca transforma dos veces el mismo elemento origen. Sin embargo, siempre se ejecuta para cada instancia de la metaclase origen. Normalmente existe una regla top en cada transformación, a partir de la cual se inicia la misma, mientras que el resto de las reglas son “no top”. A partir de la ejecución de una regla top el resto de reglas se van ejecutando, como resultado de la evaluación de bindings. Si hay varias reglas top, se ejecutan en el mismo orden que aparecen en la definición de transformación.

Copy Rule.

Una regla de copia se comporta como una regla por defecto, excepto que sí que puede transformar más de una vez el mismo elemento fuente. Esto significa que si se aplica una misma regla de copia varias veces sobre el mismo elemento origen, se crearán varios elementos destino, uno por cada aplicación. Se debe usar esta regla solo cuando no sea posible utilizar reglas por defecto, ya que puede causar recursión infinita si hay ciclos en el grafo de llamadas.

Transformaciones Modelo-Codigo

Las transformaciones modelo a código se basan en plantillas o templates. Además, un fichero de configuración especial, llamado '2code', permite hacer el mapping entre las templates y los nombres de los ficheros a ser creados. Éste tipo de ficheros nos permiten iterar sobre un modelo y seleccionar los elementos del modelo que serán transformados a código. El fichero '2code' presenta la siguiente estructura:

```
main do
      compose_file ´file.sql´ do |file|
            apply_template ´templates/template.rtemplate´, :var => var
      end
end
```

Cada una de las partes se explicará a continuación:

- La palabra clave *'main'* especifica el punto de entrada de la transformación, y siempre debe existir.

- El método *'compose_file'* permite componer un fichero con el contenido que se obtiene como resultado de aplicar varias plantillas de transformación.

- La palabra clave *'apply_template'* se usa para especificar *mappings*. Presenta dos parámetros:

    - El nombre de la plantilla a ser aplicada. El resultado de aplicar dicha *template* será almacenado en el fichero que hay definido en el apartado *'compose_file'*.

    - *Mapping* de variables de entrada de la *template*. Son especificadas usando símbolos y variables. Este *mapping* le proporciona a la *template* el contexto necesario para realizar la generación de código.

Las *templates* son ficheros con la extensión *'rtemplate'*. Siguen la misma estructura que los ficheros JSP. Es decir, cualquier contenido de la *template* que no se enmarque dentro de los delimitadores '<% %>' será traducido como texto plano en el fichero de salida obtenido al aplicar dicha *template*, y cualquier contenido dentro de los delimitadores será traducido como código en Ruby y será ejecutado.

Todo el código de Ruby que se usa en la plantilla debe estar enmarcado dentro de los delimitadores indicados anteriormente. Dichos delimitadores presenta distintas opciones:

- *'<%= %>'*: es usado para ejecutar código de Ruby y mostrar en la salida (la salida será el fichero que queremos generar) el resultado obtenido de la ejecución del código como un string.

- *'<% %>'*: se usa para escribir partes de código de Ruby que no deben ser parte de la salida, es decir, que su resultado no aparecerá en el fichero generado.

- *'<% -%>'*: con la opción '-' se omiten los saltos de línea.

6.2 Escena Inicial

Hasta ahora se ha descrito la primera parte del proyecto, que consistía en la generación de un visualizador de escenas, el cual ya está confeccionado. En esta segunda parte lo que se va a tratar de explicar es como generar las escenas que se van a visualizar, lo cual se va a conseguir haciendo uso de la herramienta RubyTL aplicando una serie de transformaciones.

Toda esta parte se desarrolló en un workspace diferente ya que se trata de una aplicación distinta. Si en la primera parte se creó un Visualizador de Escenas, aquí se va a crear un Generador de Escenas, siendo este el nombre del nuevo proyecto. Como se ha dicho la base del nuevo proyecto es el uso de RubyTL, con lo que el proyecto creado debe ser de este tipo. Para crear un proyecto RubyTL se seleccionó la opción: File -> New -> Project -> New RubyTL Project

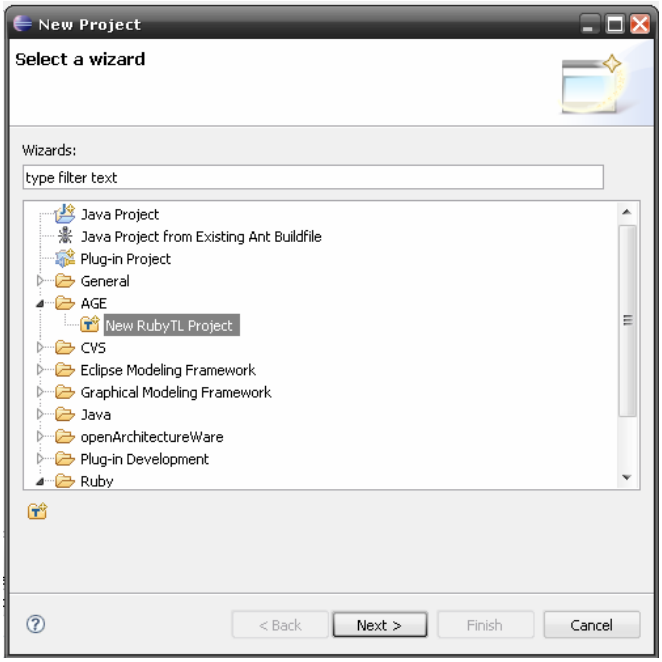


Figura 20. Creación Proyecto RubyTl

Este tipo de proyecto se crea automáticamente con seis carpetas distintas: helpers, metamodels, models, tasks, transformations y validations. Todas ellas se irán viendo y comentando para qué sirven. Pero antes de pasar de lleno a la utilización de RubyTL, se necesita un punto de partida. El punto de partida será el mismo que se utilizó para el Visualizador de escenas, es decir, un metamodelo del problema del LigthCycle, ya que lo que se pretende es generar escenas para dicho problema. Por tanto, lo que se hizo fue traer al nuevo proyecto el metamodelo que ya se creó en la primera parte y se metió en la carpeta metamodels. Una vez que se tiene el metamodelo, lo que se hizo fue crear una escena inicial del problema a partir de la cual se generarán el resto de escenas. Para ello, se deben indicar en la escena inicial las acciones que se va a querer que ocurran a lo largo de la secuencia.

Existen distintas formas de generar la escena inicial o, dicho de otra forma, de crear un modelo en base al metamodelo del LigthCycle. Quizás la más sencilla y de la que se hizo uso fue simplemente pinchando con el botón derecho del ratón en el elemento principal del metamodelo, en este caso concreto SecuenciaEscenas y seleccionando la opción "Create Dynamic Instance". Esto creó un modelo de tipo .xmi que se guardó en la carpeta models, el cual se fue modificando y adaptando a como se quería que fuese la escena inicial y las acciones que se quería que se desarrollasen a lo largo del problema.

La escena que se definió como ejemplo de escena inicial fue la siguiente:

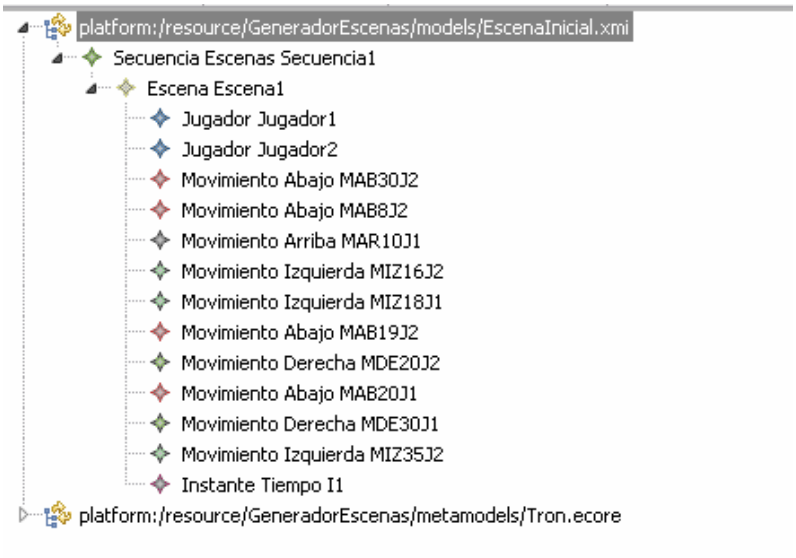


Figura 21. Modelo LigthCycle

6.3 Transformaciones Modelo-Modelo.

Una vez que se tiene la escena inicial ya diseñada, se está en disposición de generar el resto de escenas. La forma de hacerlo es utilizando RubyTL, en concreto, mediante transformaciones modelo-modelo. La idea es pasarle una secuencia de escenas a RubyTL, secuencia que solo contendrá una escena para el caso inicial, y RubyTL devolverá la secuencia pasada pero añadiéndole al final la escena siguiente generada. Este proceso se incorporó dentro de un bucle, de tal forma que se repita tantas veces como escenas se quieran generar, pudiendo acabar antes de tiempo debido a alguna causa que provoque el final del LigthCycle, ya sea un choque entre los jugadores, un choque de un jugador con un rastro dejado o la salida del jugador de los límites de la pantalla. En cada iteración del bucle, la secuencia generada en una iteración será la secuencia de entrada para la iteración siguiente y así sucesivamente.

Cabe destacar que, en el proceso de transformación modelo-modelo que se va a implementar, tanto el modelo inicial como el modelo final que se va a generar pertenecen al mismo metamodelo. Para realizar la transformación se deben tener en cuenta las "reglas del mundo" o, para este caso concreto, las reglas del problema del LigthCycle y el cómo tratarlas. En el LigthCycle se va a contar con solo unas pocas: básicamente los jugadores pueden realizar movimientos que lo que hacen es alterar la dirección que llevan en ese momento y dejan en la posición que ocupan un rastro que va

marcando el camino que siguen. En caso de no realizar ningún movimiento, el jugador sigue en la dirección que llevaba. Por otro lado el problema dará a su fin si dos jugadores se chocan entre sí, si se chocan con algún rastro dejado o si se salen de los límites de la pantalla.

Para definir la transformación se creó un fichero de tipo .rb, el cual se incluyó en la carpeta "transformations" generada por el proyecto RubyTL. El fichero con las transformaciones se llama "secuencia2secuenciasig.rb" y se puede decir que su función está dividida en dos partes. En una primera parte lo que hace es copiar íntegra la secuencia que le llega para transformar y en la segunda parte, lo que hace es generar la escena siguiente a la escena final de la secuencia de entrada.

A continuación se muestra por partes el fichero .rb que se creó:

```
transformation 'secuencia2secuenciasig'

top_rule 'secuencia2secuencia' do
  from SECUENCIA::SecuenciaEscenas
  to   SECUENCIASIG::SecuenciaEscenas
  mapping do |secuencia, secuenciasig|
    secuenciasig.nombre = secuencia.nombre
    secuenciasig.escenas = secuencia.escenas

…

rule 'escena2escena' do
  from SECUENCIA::Escena
  to   SECUENCIASIG::Escena
  mapping do |escena, escenasig|
      escenasig.nombre = escena.nombre
      escenasig.numero = escena.numero
      escenasig.siguiente = escena.siguiente
      escenasig.ultima = 'false'

      escenasig.instante = escena.instante
      escenasig.objetos = escena.objetos
      escenasig.anotaciones = escena.anotaciones
      escenasig.acciones = escena.acciones
  end
end

rule 'instante2instante' do
  from SECUENCIA::InstanteTiempo
  to   SECUENCIASIG::InstanteTiempo
  mapping do |instante, instantesig|
      instantesig.nombre = instante.nombre
      instantesig.instante = instante.instante
  end
end

rule 'jugador2jugador' do
  from SECUENCIA::Jugador
  to   SECUENCIASIG::Jugador
  mapping do |jug, jugsig|
      jugsig.nombre = jug.nombre
      jugsig.x = jug.x
      jugsig.y = jug.y
      jugsig.movimiento = jug.movimiento
```

```
  end
end

rule 'rastro2rastro' do
  from SECUENCIA::Rastro
  to   SECUENCIASIG::Rastro
  mapping do |rastro, rastrosig|
      rastrosig.nombre = rastro.nombre
      rastrosig.x = rastro.x
      rastrosig.y = rastro.y
  end
end

rule 'anotacion2anotacion' do
  from SECUENCIA::Anotacion
  to   SECUENCIASIG::Anotacion
  mapping do |anotacion, anotacionsig|
      anotacionsig.nombre = anotacion.nombre
      anotacionsig.texto = anotacion.texto
  end
end

rule 'MovimientoArriba' do
  from SECUENCIA::MovimientoArriba
  to   SECUENCIASIG::MovimientoArriba
  mapping do |accion, accionsig|
    accionsig.nombre = accion.nombre
    accionsig.ejecutor = accion.ejecutor
    accionsig.receptor = accion.receptor
    accionsig.instante = accion.instante
  end
end

rule 'MovimientoAbajo' do
  from SECUENCIA::MovimientoAbajo
  to   SECUENCIASIG::MovimientoAbajo
  mapping do |accion, accionsig|
    accionsig.nombre = accion.nombre
    accionsig.ejecutor = accion.ejecutor
    accionsig.receptor = accion.receptor
    accionsig.instante = accion.instante
  end
end

rule 'MovimientoIzquierda' do
  from SECUENCIA::MovimientoIzquierda
  to   SECUENCIASIG::MovimientoIzquierda
  mapping do |accion, accionsig|
    accionsig.nombre = accion.nombre
    accionsig.ejecutor = accion.ejecutor
    accionsig.receptor = accion.receptor
    accionsig.instante = accion.instante
  end
end

rule 'MovimientoDerecha' do
  from SECUENCIA::MovimientoDerecha
  to   SECUENCIASIG::MovimientoDerecha
  mapping do |accion, accionsig|
    accionsig.nombre = accion.nombre
    accionsig.ejecutor = accion.ejecutor
```

```
    accionsig.receptor = accion.receptor
    accionsig.instante = accion.instante
  end
end
```

Como se puede ver, la primera regla a la que se llama (top_rule) es la regla "secuencia2secuencia". En ella lo primero que hace la transformación es generar una secuencia de escenas nueva vacía, en la cual copia el nombre de la secuencia origen y acto seguido va haciendo llamadas para copiar las escenas de la secuencia de entrada. Simplemente va generando los objetos de las escenas que contiene la escena inicial y los copia a la escena nueva, copiando además los atributos de éstos. Esto simplemente se hizo con las dos primeras líneas de la regla "secuencia2secuencia" y una serie de llamadas al resto de reglas que aparecen, que son todas de copia de elementos, una por cada clase del metamodelo.

```
transformation 'secuencia2secuenciasig'

top_rule 'secuencia2secuencia' do
  from SECUENCIA::SecuenciaEscenas
  to   SECUENCIASIG::SecuenciaEscenas
  mapping do |secuencia, secuenciasig|
    secuenciasig.nombre = secuencia.nombre
    secuenciasig.escenas = secuencia.escenas

    escena = secuenciasig.escenas.last
    escenafin = SECUENCIASIG::Escena.new
    escena.siguiente = escenafin

    numero = escena.numero + 1
    escenafin.nombre = 'Escena' + numero.to_s()
    escenafin.numero = numero
    escenafin.ultima = 'false'

    instante = SECUENCIASIG::InstanteTiempo.new
    i = escena.instante.instante + 1
    instante.nombre = 'I' + i.to_s()
    instante.instante = i
    escenafin.instante = instante

    escena.anotaciones.each do |a|
      anotacion = a.clone
      escenafin.anotaciones = anotacion
    end

    escena.objetos.each do |o|
      objeto = o.clone
      escenafin.objetos = objeto
    end
```

Una vez que se tiene en la nueva secuencia una copia exacta de la secuencia de entrada (no hay más elementos que copiar), vuelve el curso de la transformación a la regla inicial y, más concretamente, a la tercera línea. A partir de aquí comienza la generación de la última escena de la secuencia, que será la siguiente a la última existente. Todo el proceso de generación de esta escena se encuentra definido a continuación dentro de la regla "secuencia2secuencia", que como se ve es bastante extensa.

En primer lugar lo que hace la regla es crear una escena nueva en la secuencia y la coloca en la posición final como escena siguiente a la última. A continuación define sus atributos y le crea a la escena un "InstanteTiempo" indicando el instante en el que se desarrolla la escena. Una vez hecho esto, lo siguiente que hace es una copia profunda de los elementos que contiene la escena anterior. Esto se hace porque la escena que se va a generar es idéntica a la anterior, contiene los mismos elementos, y lo único que habrá que hacer es modificar la posición de los jugadores, pero el resto de elementos (rastros, acciones, etc.) permanecen igual de una escena a otra. Por tanto, se hace una copia de la escena anterior, pero no una copia normal, sino una copia profunda haciendo uso del método "clone()" pues surgió un problema y es que si se hacía una copia normal desaparecían los elementos de la escena anterior. Por tanto se clonaron las anotaciones, los objetos y las acciones.

```
    escena.acciones.each do |a|
      accion = a.clone
      escenafin.acciones = accion
      if (accion.instante == escena.instante.instante)
        if (accion.kind_of? SECUENCIASIG::MovimientoArriba)
          escenafin.objetos.select {|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |j|
            if (j.nombre == accion.ejecutor)
              j.movimiento = 'arriba'
            end
          end
        elsif (accion.kind_of? SECUENCIASIG::MovimientoAbajo)
          escenafin.objetos.select {|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |j|
            if (j.nombre == accion.ejecutor)
              j.movimiento = 'abajo'
            end
          end
        elsif (accion.kind_of? SECUENCIASIG::MovimientoIzquierda)
          escenafin.objetos.select {|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |j|
            if (j.nombre == accion.ejecutor)
              j.movimiento = 'izquierda'
            end
          end
        else (accion.kind_of? SECUENCIASIG::MovimientoDerecha)
          escenafin.objetos.select {|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |j|
            if (j.nombre == accion.ejecutor)
              j.movimiento = 'derecha'
            end
          end
        end
      end
    end

    escenafin.objetos.select{|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |jugador|
      rastro = SECUENCIASIG::Rastro.new
      instante = escena.instante.instante
      avancejug = 15

      if(jugador.movimiento == 'arriba')
        jugador.y = jugador.y - avancejug
        rastro.nombre = 'RAR' + instante.to_s()
```

```
        rastro.x = jugador.x
        rastro.y = jugador.y + avancejug
      elsif(jugador.movimiento == 'abajo')
        jugador.y = jugador.y + avancejug
        rastro.nombre = 'RAB' + instante.to_s()
        rastro.x = jugador.x
        rastro.y = jugador.y - avancejug
      elsif(jugador.movimiento == 'izquierda')
        jugador.x = jugador.x - avancejug
        rastro.nombre = 'RIZ' + instante.to_s()
        rastro.y = jugador.y
        rastro.x = jugador.x + avancejug
      else(jugador.movimiento == 'derecha')
        jugador.x = jugador.x + avancejug
        rastro.nombre = 'RDE' + instante.to_s()
        rastro.y = jugador.y
        rastro.x = jugador.x - avancejug
      end
      escenafin.objetos = rastro
    end
```

En el caso concreto de las acciones se comprueba además si la acción se ejecuta en la escena que se está creando, en cuyo caso se modifica la dirección que lleva el jugador que ejecuta la acción. Una vez que se han copiado las acciones, ya se tienen todos los objetos de la escena anterior. Lo que se hace a continuación es mover los jugadores que aparecen en la escena, simplemente se avanza cada jugador en función de la dirección que lleve, dirección que se ha modificado previamente al clonar las acciones en caso de haber alguna acción, con lo que la dirección que llevan los jugadores es la correcta. Se estableció que los jugadores avancen de 15 en 15 pixels ya que menos es poco perceptible el movimiento, dado que el icono de los jugadores es de 32x32 pixels. Además, por cada movimiento del jugador lo que se hace es crear un nuevo objeto rastro y situarlo en la posición dejada por el jugador.

```
    #Comprobacion choque con rastro
    drastro = 10
    escenafin.objetos.select{|o| o.kind_of?
SECUENCIASIG::Jugador}.each do |jugador|
      escenafin.objetos.select{|o| o.kind_of?
SECUENCIASIG::Rastro}.each do |rastro|
        if (jugador.x >= rastro.x-drastro && jugador.x <=
rastro.x+drastro && jugador.y >= rastro.y-drastro && jugador.y <=
rastro.y+drastro)
          anotacion = SECUENCIASIG::Anotacion.new
          anotacion.nombre = 'ACR'
          anotacion.texto = 'Choque con rastro [' + rastro.x.to_s() +
',' + rastro.y.to_s() + ']'
          escenafin.anotaciones = anotacion
          escenafin.ultima = true
        end
      end
    end

    #Comprobacion choque jugadores
    djugador = 20
    jugador1 = escenafin.objetos.select{|o| o.kind_of?
SECUENCIASIG::Jugador}.first
    jugador2 = escenafin.objetos.select{|o| o.kind_of?
SECUENCIASIG::Jugador}.last
```

```
    if (jugador1.x >= jugador2.x-djugador && jugador1.x <=
jugador2.x+djugador && jugador1.y >= jugador2.y-djugador && jugador1.y
<= jugador2.y+djugador)
      anotacion = SECUENCIASIG::Anotacion.new
      anotacion.nombre = 'ACJ'
      anotacion.texto = 'Choque entre jugadores'
      escenafin.anotaciones = anotacion
      escenafin.ultima = true
    end

    #Comprobacion salida de pista
    pantallax = 350
    pantallay = 350
    if (jugador1.x <= 0 || jugador1.x >= pantallax || jugador1.y <= 0
|| jugador1.y >= pantallay ||
        jugador2.x <= 0 || jugador2.x >= pantallax || jugador2.y <= 0
|| jugador2.y >= pantallay)
      anotacion = SECUENCIASIG::Anotacion.new
      anotacion.nombre = 'ASP'
      anotacion.texto = 'Salida de Pista'
      escenafin.anotaciones = anotacion
      escenafin.ultima = true
    end

    secuenciasig.escenas = escenafin
  end
end
```

La última parte de la regla de transformación consiste en determinar si se ha llegado al final de la partida, para lo cual se realizan tres comprobaciones. En primer lugar se comprueba la posición de cada jugador con la posición de cada rastro para determinar si se ha chocado el jugador con algún rastro dejado. Se estableció un rango de 10 pixels para determinar si se ha producido un choque, con lo que si la posición del jugador es igual o inferior a 10 pixels con respecto a la del rastro se habrá producido un choque. Esto provocará la generación de una anotación diciendo lo ocurrido y se establecerá el atributo “ultima” de la escena a “false”. La siguiente comprobación es muy similar y lo que hace es comprobar las posiciones de los jugadores entre sí, si las posiciones están esta vez en un rango de 20 pixels se habrá producido un choque entre jugadores y de nuevo se generará una anotación y se modificara el atributo “ultima” como en el caso anterior. Finalmente lo que se hace es comprobar si el jugador se ha salido de la pantalla con lo que también daría por terminada la partida.

Con esto ya se tendría la transformación definida y la secuencia siguiente generada. Lo único que falta es implementar un bucle que ejecute dicha transformación tantas veces como se quiera. Para ello se creó una clase Java, la cual consta de un único método (método main) que va a llevar a cabo todo el proceso de generación de escenas. La primera parte del método es precisamente la ejecución de la transformación dentro de un bucle y la forma en la que se hizo fue la siguiente:

```
for (int i=0;;i++){

  //Creamos la transformacion
  Binding EscenaInicial = new Binding();
  if (i==0)
      EscenaInicial.addModel(modeloInicial);
  else
      EscenaInicial.addModel(modeloFinal);
  EscenaInicial.addMetamodel("SECUENCIA",metamodelo);

  Binding EscenaSiguiente = new Binding();
  EscenaSiguiente.addModel(modeloFinal);
  EscenaSiguiente.addMetamodel("SECUENCIASIG", metamodelo);

  TaskM2MConfigurationData m2mConf=new
     TaskM2MConfigurationData(ruby, rubyTl, proyecto, transformacion);
  m2mConf.addSourceBinding(EscenaInicial);
  m2mConf.addTargetBinding(EscenaSiguiente);

  //Creamos el rakefile
  TaskM2MWriter writer = new TaskM2MWriter();
  String result =
          writer.writeConfiguration(m2mConf,"secuencia2secuenciasig");

  //Guardamos el rakefile en un fichero
  File file = new File(rakefileMTM);
  try{
    BufferedWriter bw = new BufferedWriter(new
                                        FileWriter(rakefileMTM));
    bw.write(result);
    bw.close();
  }catch (IOException e) {
    e.printStackTrace();
  }

  //Ejecutamos la transformacion
  RakefileConfigurationData configData =
   new RakefileConfigurationData(ruby, rubyTl, proyecto, rakefileMTM);

  RakefileLauncher launcher = new RakefileLauncher(configData);
  launcher.execute("secuencia2secuenciasig");
```

Lo primero que se hace es crear la transformación. Para ello se crea un objeto de la clase 'TaskM2MConfigurationData', al cual en su constructor hay que indicarle cuatro parámetros: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto (que tomará como base para realizar la transformación) y el path donde estará el fichero que contendrá las reglas a ejecutar para realizar la transformación (el fichero de las reglas de RubyTL). Una vez que se ha creado el objeto que contendrá la transformación hay que añadirle los bindings source y target (podrá haber más de un source). Un objeto "Binding" establece la correspondencia entre el modelo de entrada con el metamodelo al cuál conforma, en caso de ser un binding source, o la correspondencia entre el modelo que se pretende crear con el metamodelo que debe conformar, en caso de ser un target. En este caso concreto tanto el metamodelo de entrada como el de salida va a ser el mismo. Además en el binding se indica el prefijo mediante el cual se referenciará al metamodelo en la transformación. El único detalle a destacar del código, es que en la primera iteración, el binding de entrada se

hace con la escena inicial creada, mientras que en el resto de iteraciones hay que coger el modelo de salida de la iteración anterior.

Una vez creada la transformación, lo siguiente que hay que hacer es crear un fichero rakefile que será el encargado de lanzar la ejecución de la transformación recién creada. Para ello, se llama al método "write" de la clase "TaskM2MWriter" para obtener en un string todo el contenido que debería tener el rakefile. A dicho método se le pasan como parámetros la transformación obtenida anteriormente y el nombre que deberá tener la transformación. El resultado se debe guardar en un fichero temporal ya que de lo contario no se podrá ejecutar. Con esto ya se tendría creado el fichero rakefile con el que lanzar la transformación modelo a modelo.

Finalmente se lanza la ejecución. Para poder ejecutar un rakefile obtenido mediante código hay que crear un objeto de la clase "RakefileConfigurationData". Para crear dicho objeto hay que pasarle en el constructor la siguiente información: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto y el path donde estará el rakefile que hemos creado anteriormente. El objeto "RakefileConfigurationData" se le pasa como parámetro al constructor de la clase "RakefileLauncher" para obtener el objeto capaz de lanzar una transformación. Este último objeto presenta un método para ejecutar una transformación llamado "execute". Al método "execute" solamente hay que pasarle el nombre que se le asignó a la transformación en el rakefile que se cargó en el "RakefileConfigurationData". Todo este proceso se irá repitiendo en cada iteración del bucle para ir generando las sucesivas escenas del problema.

6.4 API EMF.

A modo de paréntesis en lo que se refiere al uso de RubyTL se va a comentar brevemente la necesidad del uso de la API de EMF. Como se ha visto anteriormente en el código de creación y ejecución de las transformaciones modelo a modelo, se lanza una transformación en cada iteración del bucle, pero si se observa bien el código, se ve que el bucle no tiene fin, por lo tanto se estarían generando escenas indefinidamente aunque se haya producido el final del LigthCycle con anterioridad. Es en este hecho donde radica la necesidad del uso de la API de EMF. Gracias a esta API lo que se va a tratar de hacer es controlar cuándo se debe salir del bucle. En concreto se debe parar de generar escenas cuando se haya producido un choque entre jugadores, un choque de un jugador con un rastro o cuando un jugador se haya salido de los límites de la pantalla. En otras palabras, se debe mirar si la escena recién generada tiene el atributo "ultima" a "true" pues esto indicará que se ha producido alguna de las tres causas comentadas.

La forma de hacer esto es utilizando la API de EMF. Como se comentó en el apartado de EMF, es posible generar de manera automática las clases Java de un metamodelo mediante EMF con el fin de tener acceso a dicho metamodelo mediante código. Para ello, lo primero que se hace es generar el fichero .genmodel del metamodelo inicial, esto se hacía con la opción: File -> New -> Other -> EMF Model. Una vez generado el .genmodel, solo falta generar las clases Java del metamodelo, para ello se abre el fichero recién creado, se selecciona el paquete Tron (raíz del árbol que aparece), se pincha con el botón derecho del ratón sobre él y se selecciona la opción "Generate Model Code".

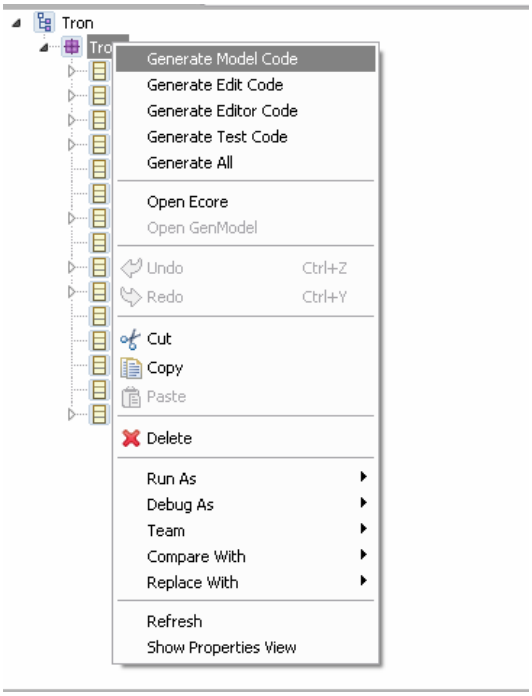


Figura 22. Generación Código Metamodelo

Esto crea una carpeta src en el proyecto con todas las clases Java del metamodelo del LigthCycle para poder acceder a él. Lo que se va a realizar a continuación es una comprobación mediante la API de EMF justo después de terminar la transformación modelo a modelo y por tanto de haber generado la escena nueva de la secuencia, en cada una de las iteraciones del bucle. Esta comprobación va a consistir en obtener la última escena de la secuencia, que será precisamente la escena recién generada en la transformación y comprobar el valor de su atributo "ultima". Si el atributo tiene su valor a "false" se continuaría con el bucle y por tanto se generaría una nueva escena, mientras que si el valor es "true" se saldría del bucle y no se generarían más escenas porque habría terminado el problema. La forma de implementar esto es la siguiente:

```
TronPackage generadorescenasPackage = TronPackage.eINSTANCE;

ResourceSet resourceSet = new ResourceSetImpl();

resourceSet.getResourceFactoryRegistry().getExtensionToFactoryMap()
                              .put("xmi", new XMIResourceFactoryImpl());
resourceSet.getPackageRegistry().put(TronPackage.eNS_URI,
                                                  generadorescenasPackage);

URI fileURI = URI.createFileURI(modeloFinal);
Resource resource = resourceSet.getResource(fileURI, true);
SecuenciaEscenas secuencia =
                        (SecuenciaEscenas)resource.getContents().get(0);
Escena escena = (Escena)secuencia.getEscenas().get(i+1);
if (escena.isUltima()){
      System.out.println("FIN DEL JUEGO");
      break;
}
```

Como se puede apreciar, lo primero que se hace es crear una instancia del metamodelo que se va a utilizar, usando para ello la clase TronPackage que se ha generado automáticamente con el .genmodel. A continuación se crea un objeto de la clase "ResourceSet", clase que se usará para acceder al modelo y a la cual hay que añadirle para utilizarla el tipo de codificación empleada en el modelo, en este caso "xmi", y el metamodelo que conforma y que se instanció en la primera línea. Finalmente se le pasa el modelo que va a tratar, o más concretamente la URI de dicho modelo, obteniendo un objeto del tipo Resource con el que ya se podrá acceder al contenido del modelo.

La forma de acceder al modelo es muy sencilla y simplemente se hará uso de las funciones implementadas por el metamodelo y que se generaron automáticamente con el .genmodel. Además, para el objetivo concreto que buscamos apenas hay que acceder al modelo. Lo primero que hay que hacer es recuperar la secuencia de escenas del modelo, que es el elemento raíz del modelo y se guarda en un objeto de su propia clase "SecuenciaEscenas" (clase también generada automáticamente con el .genmodel). A continuación, como solo se quiere ver la última escena de la secuencia, lo que se hace es obtener esa escena, que será la "i+1", siendo la variable "i" la que se veía en el bucle "for" de generación de transformaciones modelo-modelo en RubyTL, ya que esta parte hay que recordar que se está haciendo al final de dicho bucle (dentro de él).

Por último, una vez que se tiene la escena, lo único que queda es comprobar el valor de su atributo "ultima", para lo cual se invoca el método "isUltima" implementado por la clase. Si el valor del atributo es "true", se muestra un mensaje de fin del problema y se sale del bucle "for" dado que no habrá que generar más escenas, y si por el contario su valor es "false" se continúa con el bucle y se generará una nueva escena.

6.5 Transformaciones Modelo-Código.

Con todas las escenas ya generadas gracias a las transformaciones modelo-modelo y al API de EMF tan solo queda el último paso, que es transformar el modelo generado, es decir, la secuencia de escenas, en los dos ficheros ("default" y "default_diagram") que son necesarios para mostrar la secuencia en el Visualizador de Escenas.

Por cada secuencia generada en el Visualizador, éste genera dos ficheros para su correcta visualización. Un fichero de nombre "default.tron" que es un modelo muy similar al generado con las transformaciones modelo-modelo y que contiene todos los elementos que aparecen en la secuencia de escenas junto con el valor de sus atributos y un fichero de nombre "default.tron_diagram" que determina la posición de los elementos. La combinación de estos dos ficheros permite la correcta visualización de la secuencia de escenas.

El último paso que queda por tanto en el proyecto es convertir la secuencia de escenas generada mediante las transformaciones modelo-modelo, en los ficheros "default.tron" y "default.tron_diagram", para lo cual se hará uso de transformaciones modelo-código. Lo primero que se hizo para realizar una transformación modelo-código es crear un fichero ".2code" donde se indica tanto el fichero que se va a generar como la plantilla que se va a usar, así como el elemento del metamodelo que se utilizará como partida en la transformación. El fichero .2code creado es el siguiente:

```
main do
  compose_file 'default.tron' do |file|
    SECUENCIA::SecuenciaEscenas.all_objects.each do |secuencia|
      apply_template 'templates/default.rtemplate', :secuencia =>
secuencia
    end
  end

  compose_file 'default.tron_diagram' do |file|
    SECUENCIA::SecuenciaEscenas.all_objects.each do |secuencia|
      apply_template 'templates/default_diagram.rtemplate', :secuencia
=> secuencia
    end
  end
end
```

Como se ve en el código, se van a realizar dos transformaciones, ya que se van a generar dos ficheros, utilizando para la generación de cada uno de ellos una plantilla distinta. En ambos casos se le pasa como elemento de partida la secuencia de escenas.

El siguiente paso es generar las plantillas de ambos ficheros. Para ello se creó una nueva carpeta en el proyecto de nombre “templates” donde se guardó el código de ambas plantillas. La primera de ellas es “default.rtemplate” y va a guardar el modelo de dominio de la secuencia de escenas generada, es decir, va a contener todos los elementos incluidos en la secuencia junto con los valores de sus atributos. El código de la plantilla creada es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<Tron:SecuenciaEscenas xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:Tron="http://www.example.org/Tron" nombre="<%=
secuencia.nombre%>">
<%num = secuencia.escenas.length -%>
<%i=1-%>
<%secuencia.escenas.each do |escena| -%>
<%if (i<num)-%>
  <escenas nombre="<%=escena.nombre%>" numero="<%=escena.numero%>"
siguiente="//@escenas.<%=i%>" ultima="<%=escena.ultima%>">
<%else-%>
  <escenas nombre="<%=escena.nombre%>" numero="<%=escena.numero%>"
ultima="<%=escena.ultima%>">
<%end-%>
<%i=i+1-%>
<%escena.objetos.each do |objeto| -%>
<%if (objeto.kind_of? SECUENCIA::Jugador)-%>
    <objetos xsi:type="Tron:Jugador" nombre="<%=objeto.nombre%>"
x="<%=objeto.x%>" y="<%=objeto.y%>"
movimiento="<%=objeto.movimiento%>" />
<%else (objeto.kind_of? SECUENCIA::Rastro)-%>
    <objetos xsi:type="Tron:Rastro" nombre="<%=objeto.nombre%>"
x="<%=objeto.x%>" y="<%=objeto.y%>" />
<%end-%>
<%end-%>
<%escena.anotaciones.each do |anotacion| -%>
    <anotaciones nombre="<%=anotacion.nombre%>"
texto="<%=anotacion.texto%>" />
<%end-%>
```

```
<%escena.acciones.each do |accion| -%>
<%if (accion.kind_of? SECUENCIA::MovimientoArriba)-%>
    <acciones xsi:type="Tron:MovimientoArriba"
nombre="<%=accion.nombre%>" instante="<%=accion.instante%>"
ejecutor="<%=accion.ejecutor%>" />
<%elsif (accion.kind_of? SECUENCIA::MovimientoAbajo)-%>
    <acciones xsi:type="Tron:MovimientoAbajo"
nombre="<%=accion.nombre%>" instante="<%=accion.instante%>"
ejecutor="<%=accion.ejecutor%>" />
<%elsif (accion.kind_of? SECUENCIA::MovimientoIzquierda)-%>
    <acciones xsi:type="Tron:MovimientoIzquierda"
nombre="<%=accion.nombre%>" instante="<%=accion.instante%>"
ejecutor="<%=accion.ejecutor%>" />
<%elsif (accion.kind_of? SECUENCIA::MovimientoDerecha)-%>
    <acciones xsi:type="Tron:MovimientoDerecha"
nombre="<%=accion.nombre%>" instante="<%=accion.instante%>"
ejecutor="<%=accion.ejecutor%>" />
<%end-%>
<%end-%>
    <instante nombre="<%=escena.instante.nombre%>"
instante="<%=escena.instante.instante%>" />
  </escenas>
<%end-%>
</Tron:SecuenciaEscenas>
```

Esta plantilla, como se ve, genera un fichero muy similar al modelo obtenido en las transformaciones modelo-modelo y que actúa precisamente de entrada de esta transformación. Para ver mejor el resultado que devolvería esta plantilla, se muestra a continuación un ejemplo con una escena generada por la plantilla:

```
<?xml version="1.0" encoding="UTF-8"?>
<Tron:SecuenciaEscenas xmi:version="2.0"
xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:Tron="http://www.example.org/Tron" nombre="Secuencia1">
  <escenas nombre="Escena2" numero="2" siguiente="//@escenas.2"
ultima="false">
    <objetos xsi:type="Tron:Jugador" nombre="Jugador1" x="200" y="245"
movimiento="izquierda" />
    <objetos xsi:type="Tron:Jugador" nombre="Jugador2" x="70" y="60"
movimiento="derecha" />
    <objetos xsi:type="Tron:Rastro" nombre="RIZ1" x="215" y="245" />
    <objetos xsi:type="Tron:Rastro" nombre="RDE1" x="55" y="60" />
    <acciones xsi:type="Tron:MovimientoAbajo" nombre="MAB30J2"
instante="30" ejecutor="Jugador2" />
    <acciones xsi:type="Tron:MovimientoAbajo" nombre="MAB8J2"
instante="8" ejecutor="Jugador2" />
    <acciones xsi:type="Tron:MovimientoArriba" nombre="MAR10J1"
instante="10" ejecutor="Jugador1" />
    <acciones xsi:type="Tron:MovimientoIzquierda" nombre="MIZ16J2"
instante="16" ejecutor="Jugador2" />
    <acciones xsi:type="Tron:MovimientoIzquierda" nombre="MIZ18J1"
instante="18" ejecutor="Jugador1" />
    <acciones xsi:type="Tron:MovimientoAbajo" nombre="MAB19J2"
instante="19" ejecutor="Jugador2" />
    <acciones xsi:type="Tron:MovimientoDerecha" nombre="MDE20J2"
instante="20" ejecutor="Jugador2" />
    <acciones xsi:type="Tron:MovimientoAbajo" nombre="MAB20J1"
instante="20" ejecutor="Jugador1" />
```

```
    <acciones xsi:type="Tron:MovimientoDerecha" nombre="MDE30J1"
instante="30" ejecutor="Jugador1" />
    <acciones xsi:type="Tron:MovimientoIzquierda" nombre="MIZ35J2"
instante="35" ejecutor="Jugador2" />
    <instante nombre="I2" instante="2" />
  </escenas>
```

Como se puede observar, en primer lugar se incluyen dos líneas que contendrán todas las secuencias de escenas generadas. En la primera se muestra tanto la versión de xml utilizada como la codificación empleada, mientras que la segunda indica el tipo de metamodelo acorde al modelo a generar. Una vez incluidas estas dos líneas en el fichero se pasaría a ir definiendo una a una las escenas del modelo. Tan solo se ha puesto una escena en el ejemplo, pero para el resto sería lo mismo y se irían colocando a continuación. Por cada escena, tal y como se aprecia, se define un elemento "escenas" cuyos atributos son las propiedades de la escena "nombre", "numero", "siguiente" y "ultima". A continuación, dentro del elemento escena se añaden como elementos independientes los objetos que contiene la escena. En primer lugar se indican los jugadores, que serán elementos "objetos" de tipo Jugador, y se incluyen las propiedades del jugador como atributos (nombre, posición y movimiento que lleva el jugador). A continuación se definen los objetos Rastro, que también son elementos "objetos" y de tipo Rastro. Sus atributos son iguales que los del jugador pero sin el movimiento. Acto seguido se definen los distintos tipos de acciones que se van a aplicar, especificando para cada una de ellas el tipo que es y en sus atributos tanto el ejecutor como el receptor de la acción, así como el momento en el que se va a producir. Y finalmente se define el instante de tiempo como un elemento instante e indicando el instante en el que se está produciendo la escena. En la última línea se cierra el elemento "escenas" ya que no tiene más objetos definidos esta escena, si hubiese una segunda escena en la secuencia, se abriría de nuevo un elemento "escenas" y se definiría de la misma manera que la escena del ejemplo.

La segunda plantilla se llama "default_diagram.rtemplate" y va a contener el modelo de notación del modelo semántico generado en el fichero anterior, es decir, se utiliza para determinar las posiciones de los elementos en la pantalla, así como alguna otra propiedad como el tamaño de algún elemento concreto o su apariencia, detalles que no aparecen en el anterior fichero donde se limitaba a indicar qué elementos aparecían en la secuencia de escenas. El código creado en este caso es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<notation:Diagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:Tron="http://www.example.org/Tron"
xmlns:notation="http://www.eclipse.org/gmf/runtime/1.0.1/notation"
xmi:id="_tBbO4T-IEd-JZ4ucaEMtIQ" type="Tron"
name="default.tron_diagram" measurementUnit="Pixel">
<%escenax=35
numescena=0
secuencia.escenas.each do |escena| -%>
  <children xmi:type="notation:Node" type="1001">
    <children xmi:type="notation:Node" type="4005"/>
    <children xmi:type="notation:Node" type="5001">
<%numobjeto=0
escena.objetos.each do |objeto|
if (objeto.kind_of? SECUENCIA::Jugador)-%>
      <children xmi:type="notation:Node" type="2001">
        <children xmi:type="notation:Node" type="4001"/>
```

```
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Jugador"
href="default.tron#//@escenas.<%=numescena%>/@objetos.<%=numobjeto%>"/
>
        <layoutConstraint xmi:type="notation:Bounds" x="<%=objeto.x%>"
y="<%=objeto.y%>"/>
      </children>
<%else (objeto.kind_of? SECUENCIA::Rastro)-%>
      <children xmi:type="notation:Node" type="2002">
        <children xmi:type="notation:Node" type="4002"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Rastro"
href="default.tron#//@escenas.<%=numescena%>/@objetos.<%=numobjeto%>"/
>
        <layoutConstraint xmi:type="notation:Bounds" x="<%=objeto.x%>"
y="<%=objeto.y%>"/>
      </children>
<%end
numobjeto=numobjeto+1
end-%>
      <children xmi:type="notation:Node" type="2003">
        <children xmi:type="notation:Node" type="4003"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:InstanteTiempo"
href="default.tron#//@escenas.<%=numescena%>/@instante"/>
        <layoutConstraint xmi:type="notation:Bounds" x="240" y="9"/>
      </children>
      <styles xmi:type="notation:SortingStyle" />
      <styles xmi:type="notation:FilteringStyle" />
    </children>
    <%if (escena.ultima) -%>
    <children xmi:type="notation:Node" type="5002">
      <%numanotacion=0-%>
      <%escena.anotaciones.each do |anotacion| -%>
      <children xmi:type="notation:Node" type="2004">
        <children xmi:type="notation:Node" type="4004"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Anotacion"
href="default.tron#//@escenas.<%=numescena%>/@anotaciones.<%=numanotac
ion%>"/>
        <layoutConstraint xmi:type="notation:Bounds"/>
      </children>
      <%numanotacion=numanotacion+1-%>
      <%end-%>
      <styles xmi:type="notation:SortingStyle" />
      <styles xmi:type="notation:FilteringStyle" />
    </children>
    <% end-%>
<styles xmi:type="notation:ShapeStyle" />
    <element xmi:type="Tron:Escena"
href="default.tron#//@escenas.<%=numescena%>"/>
    <layoutConstraint xmi:type="notation:Bounds" x="<%=escenax%>"
y="50" width="350"/>
<%escenax=escenax + 460-%>
<%numescena=numescena +1-%>
  </children>
<%end-%>
  <styles xmi:type="notation:DiagramStyle"/>
  <element xmi:type="Tron:SecuenciaEscenas" href="default.tron#/"/>
</notation:Diagram>
```

A simple vista esta plantilla resulta bastante compleja de comprender, pero la clave para entenderla es que los atributos “type” seguidos con un número identifican a un elemento del metamodelo. Cada correspondencia entre elemento del metamodelo y número, se puede encontrar en el fichero “Tron.gmfgen” que se generó automáticamente en el proyecto anterior del Visualizador. Entendiendo esto, se ve que por cada escena se indican los elementos que contiene, incluyendo las etiquetas que muestra en cada momento y especifica el estilo y la posición de los elementos. Para el caso de las escenas, como su tamaño está predeterminado, también indica el tamaño de la misma. Finalmente para poder enlazar este fichero con el anterior se indica mediante los atributos “href” la correspondencia de los elementos de éste con los del fichero anterior.

Como sucedía en la plantilla anterior, se muestra a continuación el fichero generado para la misma escena que se definió en el ejemplo anterior:

```
<?xml version="1.0" encoding="UTF-8"?>
<notation:Diagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:Tron="http://www.example.org/Tron"
xmlns:notation="http://www.eclipse.org/gmf/runtime/1.0.1/notation"
xmi:id="_tBbO4T-IEd-JZ4ucaEMtIQ" type="Tron"
name="default.tron_diagram" measurementUnit="Pixel">
<children xmi:type="notation:Node" type="1001">
    <children xmi:type="notation:Node" type="4005"/>
    <children xmi:type="notation:Node" type="5001">
      <children xmi:type="notation:Node" type="2001">
        <children xmi:type="notation:Node" type="4001"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Jugador"
href="default.tron#//@escenas.1/@objetos.0"/>
        <layoutConstraint xmi:type="notation:Bounds" x="200" y="245"/>
      </children>
      <children xmi:type="notation:Node" type="2001">
        <children xmi:type="notation:Node" type="4001"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Jugador"
href="default.tron#//@escenas.1/@objetos.1"/>
        <layoutConstraint xmi:type="notation:Bounds" x="70" y="60"/>
      </children>
      <children xmi:type="notation:Node" type="2002">
        <children xmi:type="notation:Node" type="4002"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Rastro"
href="default.tron#//@escenas.1/@objetos.2"/>
        <layoutConstraint xmi:type="notation:Bounds" x="215" y="245"/>
      </children>
      <children xmi:type="notation:Node" type="2002">
        <children xmi:type="notation:Node" type="4002"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:Rastro"
href="default.tron#//@escenas.1/@objetos.3"/>
        <layoutConstraint xmi:type="notation:Bounds" x="55" y="60"/>
      </children>
      <children xmi:type="notation:Node" type="2003">
        <children xmi:type="notation:Node" type="4003"/>
        <styles xmi:type="notation:ShapeStyle"/>
        <element xmi:type="Tron:InstanteTiempo"
href="default.tron#//@escenas.1/@instante"/>
        <layoutConstraint xmi:type="notation:Bounds" x="240" y="9"/>
      </children>
```

```
      <styles xmi:type="notation:SortingStyle" />
      <styles xmi:type="notation:FilteringStyle" />
    </children>
    <styles xmi:type="notation:ShapeStyle" />
    <element xmi:type="Tron:Escena" href="default.tron#//@escenas.1"/>
    <layoutConstraint xmi:type="notation:Bounds" x="495" y="50"
width="350"/>
  </children>
<styles xmi:type="notation:DiagramStyle"/>
  <element xmi:type="Tron:SecuenciaEscenas" href="default.tron#/"/>
</notation:Diagram>
```

En primer lugar se definen de nuevo dos líneas introductorias con información de xml y del metamodelo que se va a utilizar, en este caso el metamodelo de notación. De nuevo esta parte es fija en todos los ficheros “default_diagram” generados. Hay que resaltar en la segunda línea el último atributo definido “measurementUnit”, el cual indica la unidad de medida empleada para posicionar los elementos en la escena. Se ha asignado como unidad de medida el pixel.

A continuación se define la escena propiamente dicha, en esta ocasión no se van a indicar todos los elementos que contiene la escena, sino solo aquellos que se van a representar. En primer lugar se define un elemento “children” de tipo 1001. Este elemento referencia a la escena. Para comprobarlo hay que acceder al fichero “Tron.gmfgen” que se generó durante la creación del Visualizador de Escenas. Dentro del fichero .gmfgen hay un elemento llamado “Gen Diagram SecuenciaEscenasEditPart” el cual si se despliega contiene en su interior todos los elementos que se definieron que se querían mostrar. Cada uno de estos elementos contiene una propiedad “Visual Id” que contiene un número que se corresponde con los números que aparecen en este fichero de notación. La escena contiene a su vez un nuevo elemento children de tipo 4005. Este elemento se corresponde con la etiqueta que muestra el número de la escena que, tal y como se definió en el modelo gráfico del visualizador de escenas, iba a ser el único atributo de la escena que se iba a representar.
Si se hubiesen indicado más atributos de la escena a representar se pondrían a continuación.

El siguiente elemento es de tipo 5001. Este elemento hace referencia al primer compartimento de la escena, el que va a contener los objetos (había un segundo compartimento que contendrá las anotaciones) y es que en este modelo también se indican este tipo de detalles ya que los compartimentos también se representan. Obviamente, lo que aparecerá en el fichero a continuación dentro de este elemento son los objetos Jugador y Rastro que aparecen en la escena. Los jugadores son los elementos de tipo 2001 y los rastros de tipo 2002, y en ambos casos se van a mostrar sus atributos icono para que se pueda ver, como ya se explicó en la parte del visualizador, un icono representativo para ambos objetos. Los elementos de tipo 4001 y 4002 se corresponden con dicho atributo en ambos casos. Para el caso de los jugadores y de los rastros se definen además un par de propiedades interesantes. En primer lugar se indica la propiedad “href” que, como se comentó antes, se utiliza para indicar a qué elemento del fichero “default” hace referencia el objeto. Y en segundo lugar se definen las posiciones de los objetos con los atributos “x” e “y”.

Esta es la clave y el objetivo primordial del modelo de notación: poder posicionar los elementos en pantalla, pues aunque en el modelo anterior del fichero “default”

aparecían los valores “x” e “y” definidos, en ese modelo simplemente aparecían a modo de información pero no posicionaban el elemento y es en este modelo donde sí se utilizan estos valores.

Una vez definidos los objetos Jugador y Rastro, si hubiese anotaciones definidas para la escena se pondrían a continuación utilizando el tipo 2004, y se pondrían dentro del segundo compartimento, es decir, del compartimento de tipo 5002. Al no haber anotaciones lo siguiente que se define es el Instante de tiempo, que se hace de la misma forma que los jugadores y los rastros.

Para terminar se definen las propiedades de la escena, indicando de nuevo mediante el elemento “href” a qué escena se corresponde del fichero “default”. Para el caso de la escena, además de indicar la posición en pantalla, se indica la anchura que va a tener ya que se prefijó en el modelo gráfico del Visualizador de escenas. Finalmente se indica la secuencia de escenas definida, también referenciando la secuencia de escenas del fichero “default”. Al igual que sucedía para el caso del fichero “default”, en caso de tener más escenas la secuencia, el procedimiento de definición de las mismas sería idéntico y simplemente se irían definiendo una a continuación de otra.

Con estos dos ficheros definidos ya se tendrían todos los elementos necesarios para realizar la transformación modelo-código, lo único que falta es lanzar la transformación. El código implementado para ello es muy similar al que se utilizó en la transformación modelo-modelo y está incluido a continuación de éste en el proyecto, es decir, a la salida del bucle donde se implementaron las transformaciones modelo-modelo, ya que salíamos del bucle cuando se había generado toda la secuencia de escenas. El código es el siguiente:

```
Binding SecuenciaFinal = new Binding();
SecuenciaFinal.addModel(modeloFinal);
SecuenciaFinal.addMetamodel("SECUENCIA",metamodelo);

TaskM2TConfigurationData m2tConf=new TaskM2TConfigurationData(ruby,
                        rubyTl, proyecto, toCode, ficheros);
m2tConf.addSourceBinding(SecuenciaFinal);

TaskM2TWriter writer = new TaskM2TWriter();
String result = writer.writeConfiguration(m2tConf,"model2code");

//Guardamos el rakefile en un fichero
File file = new File(rakefileMTC);
  try{
      //if (file.createNewFile()){
      BufferedWriter bw = new BufferedWriter(new
                                    FileWriter(rakefileMTC));
      bw.write(result);
      bw.close();
      //}
   }catch (IOException e) {
      e.printStackTrace();
}

//Ejecutamos la transformacion
RakefileConfigurationData configData =
            new RakefileConfigurationData(ruby, rubyTl, proyecto,
rakefileMTC);
```

```
RakefileLauncher launcher = new RakefileLauncher(configData);
launcher.execute("model2code");
```

El código, como se puede ver, es muy similar al de las transformaciones modelo-modelo salvo algunas diferencias. Para obtener una transformación modelo a código hay que crear un objeto de la clase "TaskM2TConfigurationData", al cual en su constructor hay que indicarle cinco parámetros: el comando para ejecutar Ruby, el path donde está instalado RubyTL, el path del proyecto (que tomará como base para realizar la transformación), el path donde estará el fichero "2code" y el nombre de la carpeta donde se deberán guardar los ficheros generados por la transformación. En este caso, la transformación únicamente tendrá un "binding", que será un "source" correspondiente a la secuencia de escenas final obtenida tras la ejecución del bucle. El fichero "rakefile" se obtiene de la misma manera que en la transformación modelo a modelo, es decir, llamando al método "write", pero el de la clase "TaskM2TWriter", mientras que la ejecución se realiza de la misma manera.

Con esto ya estarían generados en la carpeta "files" que se creó en el proyecto previamente, los ficheros necesarios para la visualización de la secuencia en el Visualizador de Escenas ("default" y "default_diagram"). Un ejemplo realizado para mostrar cómo se visualiza una secuencia es el siguiente:

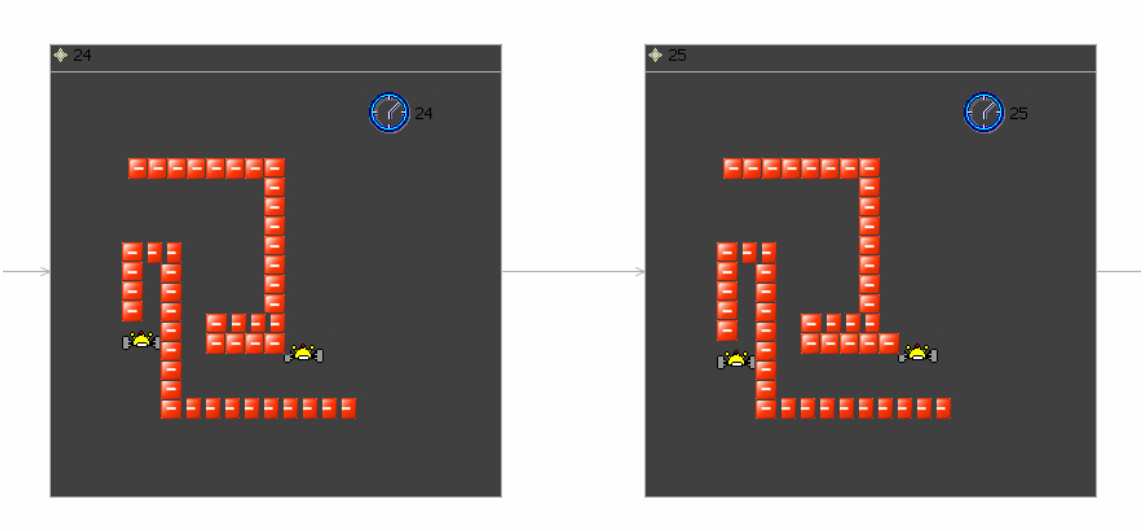


Figura 23. Escenas LigthCycle

7. Conclusiones y Vías Futuras.

Para concluir cabe mencionar que la aplicación de tecnologías DSDM en la construcción de un generador de escenas de esta naturaleza ha sido realmente satisfactoria. La construcción de un visualizador también ha sido muy productiva utilizando las herramientas disponibles desde los subproyectos de EMF.

Sin embargo al final no se han podido conseguir plenamente los objetivos marcados por el proyecto. Se consiguieron construir tanto un generador como un visualizador, conformando una suite completa para la generación de escenas. No obstante, se comenta que no se alcanzaron plenamente los objetivos por que finalmente no se consiguió construir los productos resultantes desde la perspectiva de un framework reutilizable y extensible, que era una de las cuestiones básicas que justificaban el empleo de tecnologías DSDM, y el uso de modelos y lenguajes sobre modelos. Como ya se ha comentado, a lo largo del proyecto han aparecido contratiempos que han hecho desviar la propuesta inicial del proyecto, pero esto no significa que las limitaciones encontradas sean insalvables sino que simplemente el acometerlas hacían aumentar demasiado la envergadura de un proyecto único. Pero sería muy interesante poder seguir este trabajo por esas vías dejadas. Hay que remarcar, por tanto, que se ha conseguido obtener la semilla del framework descrito en la descripción inicial del problema. Es decir, aunque no se ha conseguido genericidad en la solución, si se ha podido implementar una solución empleando modelos (para las transformaciones y la generación de código). Cabe resaltar la importancia del empleo de modelos en la solución por que permite que en el futuro la solución se adapte hacia la reutilización y extensibilidad de los dominios para la generación de escenas. Dicho empleo permite el aprovechamiento de tecnologías DSDM como los lenguajes de transformación o generación de código, y el uso de subproyectos como GMF, que al final repercuten en la productividad del desarrollo y en la calidad de la solución.

Por tanto, los objetivos que se buscaban con el proyecto quedan colmados, ya que se ha podido comprobar cómo dada una escena inicial se pueden generar todas las escenas que queramos siguientes a la misma en un futuro, y poder mostrarlas de forma gráfica. Pero aunque se ha llevado a cabo un gran paso inicial, la grandeza que esconde el proyecto vendría a continuación en posibles desarrollos futuros.

Lo primero y más importante, tal y como se planteó el proyecto en un principio sería hacer un visualizador de escenas en GMF extensible y reutilizable. Es decir crear un visualizador de escenas base, el cual dependiendo del dominio del problema se pudiese ampliar y sirviese para la visualización de escenas del problema específico. Para conseguir este propósito habría que acometer dos vías:

- En primer lugar se deberían tratar de hacer los metamodelos extensibles, es decir, tener un metamodelo de escenas básico que no sería otro que el metamodelo de las clases en gris mostrado en la Figura 2 y lograr extenderlo para cada caso concreto. Esta vía se vio truncada en el proyecto ante la imposibilidad ya comentada de hacer uso del Load Resource en GMF.

- Y en segundo lugar habría que tratar de hacer un generador de visualizadores. Lo primero que habría que hacer para lograr este hito sería metamodelar los 3 ficheros que es necesario construir en un visualizador, que como se ha visto son .gmfgraph, .gmftool y .gmfmap. y tratar de generarlos. Esta vía se comenzó a investigar y de hecho se consiguió metamodelar los tres ficheros. Con vistas a un futuro desarrollo, se muestran los metamodelos que se definieron para los tres ficheros en el apartado 9 de Anexos.

Estas dos vías serían sin ninguna duda las más interesantes de acometer por la extensibilidad de la que dotarían a la herramienta. Otra posible vía, de objetivo distinto a las anteriores pero muy interesante visualmente, sería la generación del código final de la secuencia de escenas en formatos "reproducibles", como por ejemplo Macromedia Flash. Como ya se ha comentado GMF tiene bastantes limitaciones en la representación y aunque el resultado final ha sido bueno, gracias a los "trucos" que se han aplicado (iconos, colores…), la utilización de técnicas como Flash conseguirían una mejora considerable en la representación.

Además, el proyecto al final ha construido un generador de escenas a medida del dominio de aplicación propuesto, y perdiendo de este modo la entidad de framework con la que se partía. No obstante, quedan sentadas las bases para adaptar (desde el punto de vista de los metamodelos) la solución al marco de un framework extensible y reusable.

Finalmente otra serie de vías de menor importancia que también se podrían acometer serían: utilizar el mecanismo de fases para las reglas de actuación o usar el API EMF de manera reflexiva para consultar el atributo "ultima" de la escena.

8. Bibliografía.

[Visualizador de Escenas]

"Bibliografía utilizada para el entendimiento de EMF y GMF"

- Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/.
- D. Steinberg, et al.,: *EMF: Eclipse Modelling Framework, 2d Edition.* Addison Wesley.
- Documentación de GMF: http://wiki.eclipse.org/GMF_Documentation
- Foro de GMF. http://dev.eclipse.org/newslists/news.eclipse.modeling.gmf/
- Proyecto Fin de Carrera: "**Desarrollo de herramientas MDA en la plataforma Eclipse**". Autor: **Javier Luis Cánovas Izquierdo**. Junio 2006. Universidad de Murcia. Facultad de Informática.
- Proyecto Fin de Carrera: "**myme: desarrollo dirigido por modelos con MyMobileWeb**". Autor: **Ángel Luis Calvo Ortega**. Julio 2007. Universidad de Murcia. Facultad de Informática.

[Generador de Escenas]

"Bibliografía utilizada para el entendimiento de RubyTL"

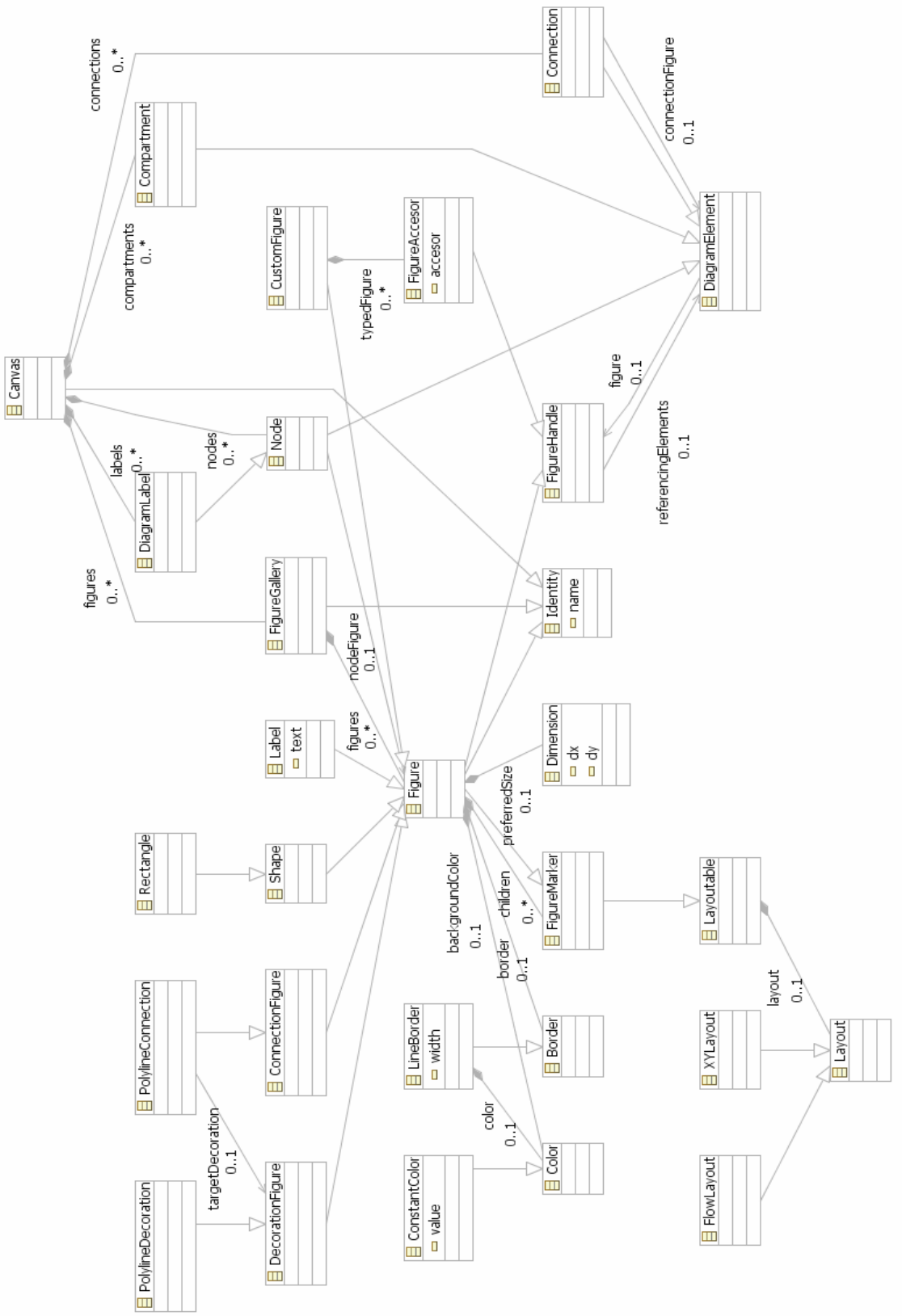
- Agile Generative Environment (AGE). http://gts.inf.um.es/trac/age
- Jesús Sánchez Cuadrado**:** Tutorial RubyTL 1.
- Jesús Sánchez Cuadrado**:** Tutorial RubyTL 2.
- Jesús Sánchez Cuadrado**:** RubyTL Crash Course.
- Ruby-Doc.org. http://ruby-doc.org/core/
- Proyecto Fin de Carrera: "**Datos de un Sistema de Informacón aplicando técnicas DSDM**". Autor: **Francisco Moya Arnao**. Febrero 2010. Universidad de Murcia. Facultad de Informática.

[Otros]

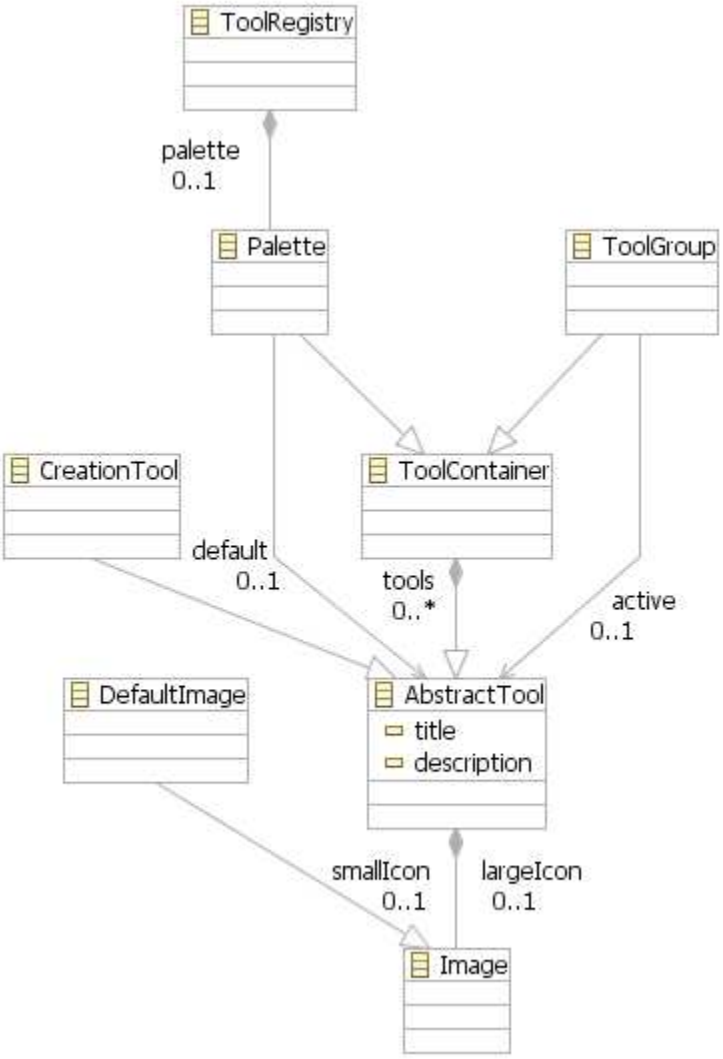
- Tron. Film de walt Disney Productions : http://es.wikipedia.org/wiki/Tron

9. Anexos.

9.1 Metamodelo .gmfgraph



9.2 Metamodelo .gmftool



9.3 Metamodelo .gmfmap