MDA y Desarrollo de Videojuegos

Emanuel Montero y José A. Carsí

Grupo de Ingeniería del Software y Sistemas de Información,
Departamento de Sistema Informáticos y de Computación,
Universitat Politècnica de València.
Camí de Vera s/n 46022 Valencia, España
{emontero, pcarsi}@dsic.upv.es

Resumen. El desarrollo de videojuegos representa un dominio de aplicación perfecto para la aplicación de *Model-Driven Architecture* (MDA). Presentamos un modelo PIM para el diseño de videojuegos que permite la especificación precisa de videojuegos 2D utilizando tres perspectivas fundamentales: jugabilidad, control e Interfaz Gráfica de Usuario (GUI). También se ofrece una infraestructura MDA para el desarrollo de videojuegos: meta-modelos PIM y PSM, una transformación QVT que transforma los conceptos PIM en conceptos PSM, y un compilador de modelos PSM que genera código fuente ejecutable para el motor de juegos *Microsoft XNA Game Studio*. Esta propuesta MDA para el desarrollo de videojuegos permite crear videojuegos 2D aprovechando todas las ventajas del desarrollo de software dirigido por modelos.

Palabras Clave: desarrollo de software dirigido por modelos, MDA, desarrollo de videojuegos.

1 Introducción

Desde sus inicios en la década de 1970, la industria del videojuego ha desarrollado software interactivo de entretenimiento para una creciente variedad de plataformas tecnológicas: PC, videoconsolas, navegadores Web y dispositivos móviles. Por ello, el desarrollo de videojuegos ha aumentado constantemente en complejidad tecnológica sin incrementar significativamente su nivel de abstracción [3]. Incluso con la ayuda de *middleware* específico y *game engines*, el desarrollo de juegos sigue dependiendo fuertemente de la programación en lenguajes orientados a objetos. Los desarrolladores de juegos carecen de un lenguaje de especificación para los videojuegos, por lo que los diseñadores de juegos tienen que escribir la documentación de diseño en lenguaje natural [6, 11, 12]. Los programadores traducen manualmente esta especificación imprecisa a código fuente, que es compilado y utilizado para realizar pruebas. Los *testers* prueban el juego y detectan errores que deben ser reparados. Algunos de estos errores pueden resolverse fácilmente a nivel de código, mientras que otros requieren cambios a nivel de diseño de juego. Como la documentación de diseño no es un artefacto software (modelo) sino un documento de texto en lenguaje natural, los cambios en el diseño son actualizados directamente a nivel de código (una práctica común conocida como el atajo del programador). Tras varias iteraciones de rediseño-programación-pruebas, el código del videojuego se

hace difícil de mantener. Nótese que el vacío semántico entre los conceptos de diseño y de programación del videojuego, hace que la tarea de implementación sea pesada y costosa (ver Figura 1 izquierda).

Con el objetivo de aliviar este escenario aplicando MDA, proponemos elevar el nivel de abstracción del desarrollo de videojuegos mediante un modelo PIM para el diseño de juegos. El desarrollo de software dirigido por modelos utiliza artefactos software (modelos) para capturar conceptos de un dominio concreto de aplicación, como el desarrollo de videojuegos. MDA en particular, promueve un incremento de la productividad simplificando el proceso de diseño mediante la automatización de tareas que tradicionalmente se realizan a mano, como la traducción de la documentación de diseño a código fuente que realizan los programadores. MDA también eleva el nivel de abstracción tecnológica, ya que los modelos se pueden beneficiar de lenguajes específicos de dominio (*Domain-Specific Languages*, DSLs), que permiten a los diseñadores modelar los artefactos software en sus propios conceptos de dominio [2, 8, 9]. Así mismo, MDA facilita la evolución tecnológica y el desarrollo de software multi-plataforma. A grandes rasgos, nuestra propuesta MDA para el desarrollo de videojuegos incluye:

- Un modelo PIM de alto nivel que especifica la estructura y el comportamiento del sistema (videojuego) sin considerar detalles tecnológicos de su implementación.
- Uno o más modelos PSM, cada uno describiendo cómo se implementa el modelo PIM en una plataforma tecnológica concreta.
- Una o más transformaciones automáticas entre modelos que transforman los modelos de alto nivel en modelos más concretos tecnológicamente. Típicamente, en MDA se transforma el modelo PIM en un modelo PSM, y a su vez, el modelo PSM en código fuente para una plataforma tecnológica específica (SDK, *game engine*, etc).

Dado que los videojuegos son artefactos software muy complejos, abordaremos el modelado de juegos desde distintas perspectivas de interés. La principal contribución de este artículo es un modelo PIM para el diseño de videojuegos que permite especificar las tres perspectivas fundamentales de juego: jugabilidad, control, e Interfaz Gráfica de Usuario (GUI), independientemente de la plataforma tecnológica de implementación. Además, presentamos una infraestructura MDA para el desarrollo de videojuegos, ofreciendo meta-modelos PIM y PSM, transformaciones automáticas y un compilador de modelos PSM para una plataforma tecnológica concreta: *Microsoft XNA Game Studio*. Por simplicidad, abordaremos únicamente el desarrollo de videojuegos 2D, siguiendo dos ejemplos motivacionales bien conocidos: *Pac-Man* [10] y *Street Fighter IV* [13].

La estructura de este artículo es la siguiente: la sección 2 ofrece una visión general de los avances en metodologías de desarrollo de juegos, con especial énfasis en el nivel de abstracción tecnológica. La sección 3 discute cómo aplicar MDA al desarrollo de videojuegos, detallando cada una de las perspectivas, modelos y transformaciones de nuestra propuesta. La sección 4 ofrece las conclusiones de la investigación y los trabajos futuros.

2 Estado del Arte

Existen alternativas al escenario tradicional de desarrollo de videojuegos en cascada. Furtado et al [7] han realizado esfuerzos para elevar la productividad del desarrollo de videojuegos mediante modelado específico del dominio y líneas de producto software, usando un pequeño DSL para capturar el conocimiento de los videojuegos de acción y aventuras en 2D. El uso de un DSL permite a los diseñadores de juegos trabajar con conceptos más próximos a su dominio de aplicación, a un elevado nivel de abstracción. Sin embargo, es discutible que un lenguaje tan limitado que solo permita realizar juegos de un pequeño sub-género como los juegos de acción y aventuras 2D pueda resultar de utilidad en un entorno industrial en que se desarrollan juegos de todo tipo.

Dobbe [4] aborda la definición de un DSL independiente del género para diseñar juegos. Este lenguaje permite a los diseñadores crear cualquier tipo de juego utilizando los siguientes aspectos: objetos, interacciones, reglas e historia. A pesar de que esta conceptualización del diseño de juegos puede resultar de gran interés para los diseñadores de juegos, resulta muy difícil adaptarla a un entorno industrial dado que carece de relaciones/transformaciones claras hacia conceptos de implementación.

MDA puede jugar un papel fundamental a la hora de elevar el nivel de abstracción tecnológica del desarrollo de videojuegos. En esta línea de investigación, Altunbay et al [1] ofrecen una primera aproximación al desarrollo MDA de juegos de tablero. En su conceptualización, un juego está compuesto por jugadores y un motor de juego. Los jugadores tienen objetivos que cumplir y controlan elementos de juego. El motor de juego consta de elementos de juego, estado, y un nivel con eventos, reglas y acciones. A pesar de que esta conceptualización es clara y muy útil para diseñar juegos de tablero, es difícil adaptarla al diseño de videojuegos. Sin embargo, representa la aproximación más notable al desarrollo MDA de juegos hasta la fecha.

3 Desarrollo de Videojuegos Dirigido por Modelos

Aplicar MDA al desarrollo de videojuegos puede cambiar el ciclo tradicional de vida del juego, haciendo que el diseño dirija todo el proceso de desarrollo (Figura 1 derecha). En un escenario MDA, los diseñadores de juegos especifican el juego en desarrollo mediante un modelo PIM. Nótese que el modelo PIM es un artefacto software en lugar de un documento de texto en lenguaje natural. Mediante transformaciones automáticas QVT se genera un modelo PSM a partir del modelo PIM. Opcionalmente, un modelador PSM complementa este modelo PSM con más detalles de implementación. Mediante un compilador de modelos PSM se transforma el modelo PSM en código fuente, que a su vez será complementado con detalles de bajo nivel por los programadores. El código fuente se compila y los binarios son utilizados por los *testers* para realizar pruebas. Al probarse el juego, se detectan errores o mejoras de la jugabilidad. Estos errores pueden ser refinados a todos los niveles de abstracción. Los cambios en el diseño de juegos se realizan sobre el modelo PIM, los cambios de implementación sobre el modelo PSM, y los cambios a bajo nivel se realizan sobre el código fuente. Como las transformaciones automáticas

ofrecen una clara trazabilidad entre artefactos software, los cambios a distintos niveles de abstracción pueden soportarse mediante anotaciones. Nótese que el esfuerzo tradicionalmente realizado a mano de implementación del videojuego ahora se realiza de forma semi-automática con transformaciones entre modelos. Además, al facilitarse la actualización de las mejoras de jugabilidad sobre el diseño de juegos, MDA favorece el diseño iterativo de juegos, esto es, mejorar la jugabilidad mediante sucesivo ciclos de pruebas y refinamientos, que ha sido resaltado como factor crítico para la calidad de los videojuegos [6].

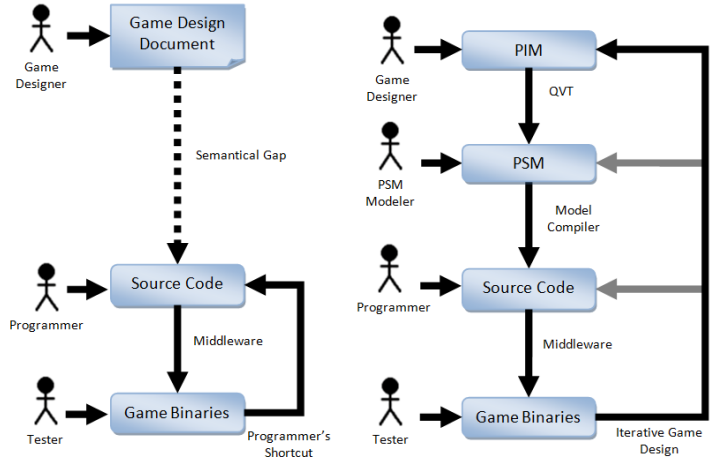

Fig. 1. (Izquierda) Desarrollo de videojuegos tradicional, y (derecha) desarrollo de videojuegos dirigido por modelos.

Las siguientes secciones ofrecen más detalle sobre cómo aplicar MDA al desarrollo de videojuegos. La sección 3.1 examina en profundidad los modelos PIM para la especificación de videojuegos. La sección 3.2 aborda el modelo PSM para una plataforma tecnológica concreta. Y la sección 3.3 detalla las transformaciones automáticas de PIM a PSM y de PSM a código fuente.

3.1 Modelo PIM para el Diseño de Videojuegos

El primer paso para aplicar MDA al desarrollo de videojuegos es capturar todos los conceptos del dominio en un meta-modelo PIM para el diseño de juegos. Un buen modelo PIM debe permitir a los desarrolladores expresar cualquier videojuego de forma precisa, con independencia de sus detalles de implementación subyacentes. Ya que los videojuegos son artefactos software muy complejos, capturar todos los conceptos del diseño de juego en un solo modelo no es viable. Por tanto, proponemos una descomposición de los videojuegos en distintas vistas de interés (perspectivas) que definen las principales áreas o sub-dominios del desarrollo de juegos.

Partiendo del ciclo de interactividad de Crawford [5], los videojuegos pueden considerarse sistemas interactivos donde los jugadores se comunican con el juego a través de acciones de entrada realizadas mediante controladores hardware, y a su vez

el juego se comunica con los jugadores a través de señales de salida en un dispositivo hardware de representación (monitor, pantalla, TV). Este ciclo entrada-proceso-salida se repite continuamente durante la interacción entre los jugadores y el juego. Por tanto, podemos diferenciar tres perspectivas fundamentales en el diseño de videojuegos: jugabilidad, control, e interfaz gráfica de usuario (*Graphical User Interface*, GUI). La perspectiva de la jugabilidad define la estructura interna y el comportamiento del juego: cómo se transforman las entradas en salidas. La perspectiva de control se centra en las acciones de los jugadores y los dispositivos de entrada. La perspectiva GUI se centra en las salidas visuales y los dispositivos de representación. Cada perspectiva del modelo PIM será abordada en detalle a continuación.

3.1.1. Perspectiva de Jugabilidad

El objetivo fundamental de la perspectiva de jugabilidad es definir de forma precisa la estructura y el comportamiento del juego. Para especificar la estructura interna del juego se utilizan dos diagramas basados en UML: el diagrama de contexto social y el diagrama de estructura de la jugabilidad. Para especificar el comportamiento del juego se utilizan reglas definidas mediante un pequeño subconjunto de OCL.

El **diagrama de contexto social** especifica la estructura externa del videojuego, esto es, cuántos jugadores interactúan con el sistema de juego. La Figura 2 muestra el meta-modelo de contexto social, en el que todas las primitivas del diagrama de contexto social se representan como meta-clases.

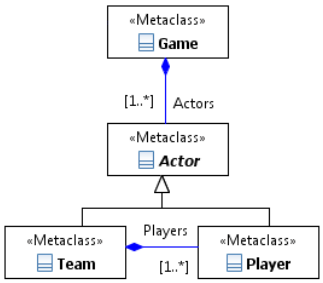

Fig. 2. Meta-modelo de contexto social.

Para facilitar el modelado visual a los diseñadores de juegos, se ofrece un DSL para especificar diagramas de contexto social. La Figura 3 muestra tres ejemplos de la sintaxis concreta del DSL de contexto social. Los jugadores se representan mediante el icono de una persona. Los equipos (grupos de jugadores que cooperan entre sí) se representan mediante círculos de línea discontinua. Tanto los jugadores como los equipos pueden complementarse con nombres de rol (que diferencian funciones en el juego) y cardinalidades (que expresan el número mínimo y máximo de jugadores que interactúan en el juego). El sistema de juego se representa mediante una caja coloreada.

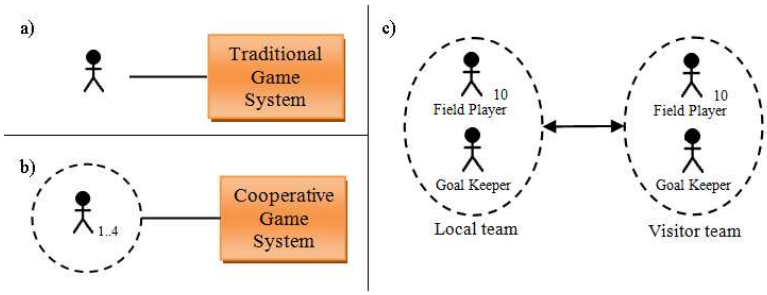

Fig. 3. Ejemplos de diagrama de contexto social: (a) juego para un único jugador, (b) juego cooperativo para cuatro jugadores, (c) juego competitivo para dos equipos: fútbol.

La Figura 3.a muestra el diagrama de contexto social de un juego en el que interactúa un único jugador. *Pac-Man* y muchos otros videojuegos se incluyen en esta categoría. La Figura 3.b muestra el diagrama de contexto social de un juego cooperativo para cuatro jugadores. Muchos videojuegos de la videoconsola *Wii* se incluyen en esta categoría. La Figura 3.c muestra el diagrama de contexto social del futbol, en el que compiten dos equipos formados por diez jugadores de campo y u portero (por simplicidad, el sistema de juego se ha eliminado del diagrama). Nótese que el diagrama de contexto social permite especificar todo tipo de juegos, tanto digitales como no digitales, representando una potente herramienta de trabajo para los diseñadores de juegos.

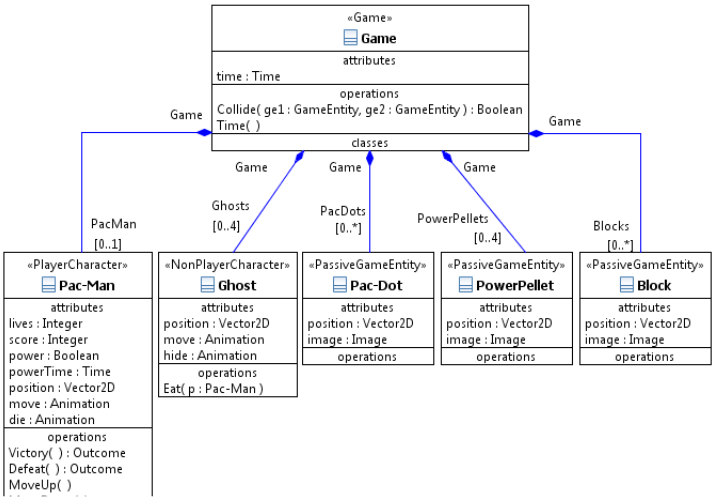

Fig. 4. Diagrama de estructura de la jugabilidad de *Pac-Man*.

El **diagrama de estructura de la jugabilidad** especifica qué entidades existen en el sistema de juego. La Figura 4 muestra el diagrama de estructura de la jugabilidad de *Pac-Man*, en la que cada clase se representa como un estereotipo UML. Las clases

de juego se pueden identificar como entidades de juego de varios tipos: personajes controlados por el jugador (como Pac-Man), personajes no jugadores (cuyo comportamiento es controlado por la inteligencia artificial del juego, como los fantasmas) y entidades pasivas (cuyo comportamiento puede considerarse pasivo, como los Pac-Dots y los Power-Pellets que forman parte del laberinto). Cada una de estas entidades de juego se especifica en mayor detalle de igual forma que un diagrama de estructura UML.

Habiendo definido de forma precisa la estructura de la jugabilidad, debemos modelar su comportamiento. Para ello, pueden utilizarse distintas herramientas de modelado UML: diagramas de transición entre estados, diagramas de secuencia, etc. Por su notación textual declarativa, en este trabajo se utilizará un **conjunto de reglas** para modelar el comportamiento de los videojuegos. Cada regla se define como una operación en el diagrama de estructura de la jugabilidad que será especificada utilizando un pequeño subconjunto de OCL. Todas las reglas se definen utilizando las cláusulas *pre* y *post* para definir pre-condiciones y post-condiciones. Según la función que cumpla la regla en el conjunto de reglas, pueden diferenciarse tres tipos de reglas: reglas de acción (que definen cómo controlan los jugadores a sus personajes en el sistema de juego), reglas de objetivo (que definen cómo los jugadores ganan o pierden el juego) y reglas internas (que definen cómo se comporta internamente el sistema de juego).

A modo de ejemplo se muestra la de regla de acción *MoveDown*, que verifica si Pac-Man tiene espacio para moverse hacia abajo, y lo mueve en consecuencia:

```
Context Pac-Man::MoveDown():void
Pre: Game.Blocks->Select(b:Block | b.position.X =
self.position.X and b.position.Y=self.position.Y+1)->notEmpty()
Post: self.position.Y = self.position.Y@pre+1
```

Para definir las reglas de objetivo se ha definido un tipo primitiva específica para el diseño de juegos: el resultado del juego (*game outcome*). El resultado de juego define si el juego termina en victoria, derrota o empate. Por ello, podemos diferenciar las reglas de objetivo en reglas de objetivo positivas (que definen cómo se gana el juego), negativas (que definen cómo se pierde el juego) o neutrales (que definen cómo se empata el juego). A modo de ejemplo se muestran las dos reglas de objetivo de *Pac-Man*, que definen cómo se pierde la partida (al quedarse Pac-Man sin vidas) y cómo se supera un nivel (al comerse Pac-Man todos los Pac-Dots y Power Pellets del nivel):

```
Context Pac-Man::Victory():Outcome
Pre: Game.Pac-Dots->size()=0 and Game.PowerPellets->size()=0
Post: result = Victory

Context Pac-Man::Defeat():Outcome
Pre: self.lives=0
Post: result = Defeat
```

3.1.2. Perspectiva de Control

El control define cómo se comunican los jugadores con el juego a través de dispositivos *hardware* controladores. Cada plataforma tecnológica ofrece diversos controladores que permiten a los jugadores distintas interacciones. Los PCs típicamente ofrecen teclado y ratón como controladores principales. Otras plataformas

tecnológicas, como las videoconsolas, ofrecen controladores específicos: mandos, instrumentos musicales, etc. A primera vista, estos controladores específicos ofrecen interacciones únicas. Sin embargo, todos los controladores ofrecen a los jugadores elementos de control que permiten enviar información al sistema de juego. Así, se pueden definir algunos conceptos de control independientes de la tecnología, comunes a todos los controladores y plataformas tecnológicas.

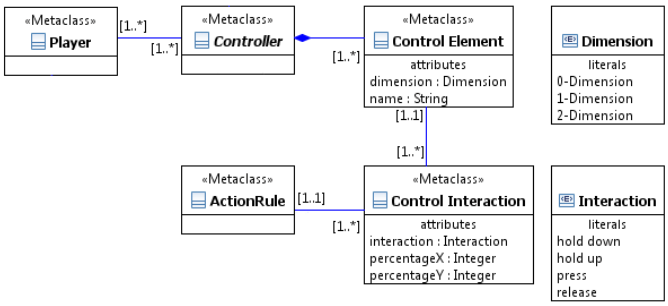

Fig. 5. Meta-modelo de control.

La Figura 5 muestra el meta-modelo de control, en el que las primitivas de control se muestran como meta-clases. Un controlador está formado por elementos de control (botones, gatillos, joysticks, etc). Los elementos de control envían información al sistema de juego (los botones envían información 0-dimensional, los gatillos 1-dimensional y los joysticks 2-dimensional). Los jugadores interactúan (pulsan, sueltan, mantienen, etc.) los elementos de control para ejecutar reglas de acción del conjunto de reglas (previamente definidas en la perspectiva de jugabilidad). Para facilitar el modelado visual a los diseñadores de juegos, se ofrece un DSL que permite especificar diagramas de control. Nótese que mediante este modelo PIM, el control del juego queda definido de forma genérica y puede implementarse en diversas plataformas tecnológicas. Por ejemplo en la Figura 6, el diagrama de control de *Pac-Man* puede implementarse mediante 4 teclas del teclado o bien mediante 4 botones de un mando de XBOX 360.

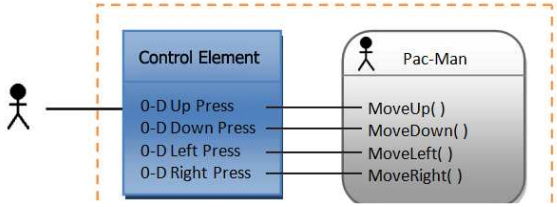

Fig. 6. Diagrama de control de *Pac-Man*.

3.1.3. Perspectiva de Interfaz Gráfica de Usuario

La perspectiva GUI define cómo se comunica información visual a los jugadores. Para ello se utilizan dos diagramas basados en UML: el diagrama de navegación entre pantallas y el diagrama de distribución de pantalla.

El **diagrama de navegación entre pantallas** muestra cómo se organiza la información del videojuego en diferentes pantallas. La Figura 7 muestra el meta-modelo de navegación entre pantallas, que representa las principales primitivas de navegación como meta-clases. Nótese que la navegación entre pantallas de un videojuego se define como un diagrama de transición de estados donde los nodos representan pantallas y las transiciones cambios de pantalla provocados por eventos, interacciones de control, o tiempo.

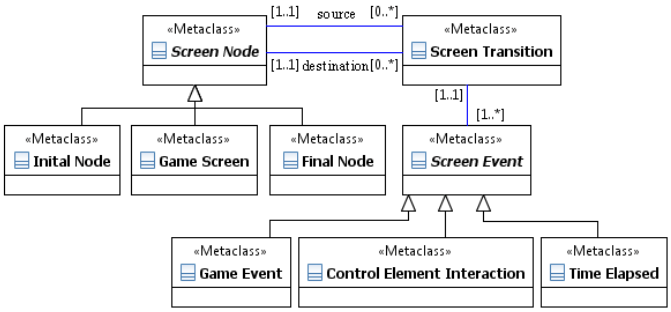

Fig. 7. Meta-modelo de navegación entre pantallas.

Para facilitar el modelado visual a los diseñadores de juegos, se ofrece un DSL para especificar diagramas de navegación entre pantallas. La Figura 8 muestra el diagrama de navegación entre pantallas de *Street Fighter IV*.

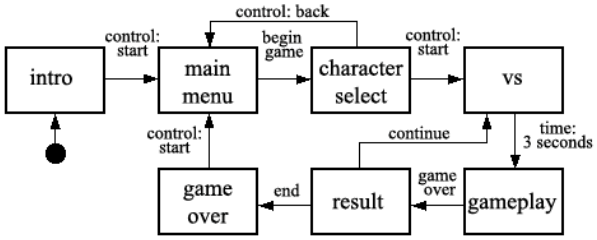

Fig. 8. Diagrama de navegación entre pantallas de *Street Fighter IV*.

Cuando el flujo de pantallas está claramente definido, cada pantalla se define en más detalle mediante un **diagrama de distribución de pantalla**. La Figura 9 muestra el meta-modelo del diagrama de distribución de pantalla. Una pantalla, previamente definida en un diagrama de navegación, puede especificarse en mayor detalle mediante primitivas de representación que muestran a los jugadores información de juego (atributos previamente especificados en la perspectiva de jugabilidad). Las primitivas de representación pueden posicionarse y escalarse en la pantalla, configurando el interfaz visual del videojuego.

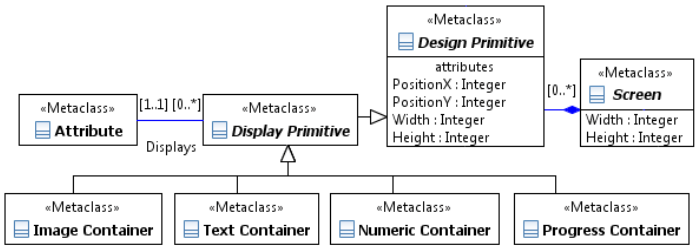

Fig. 9. Meta-modelo de distribución de pantalla.

Fig. 10. Captura de pantalla de *Street Fighter IV* (izquierda) y su correspondiente diagrama de representación de pantalla (derecha).

Para facilitar el modelado visual a los diseñadores de juegos, se ofrece un DSL para especificar diagramas de distribución de pantalla. La Figura 10 muestra el diagrama de distribución de pantalla de *Street Fighter IV*, donde se diferencian tres tipos de primitivas de representación:

Contenedores numéricos/textuales, que representan un atributo textualmente. En el DSL se representa mediante una caja con una letra N o T en su interior (dependiendo de si el atributo quiere mostrarse como un número o un texto). En *Street Fighter IV*, los nombres de los dos luchadores se muestran mediante un contenedor textual, mientras que el tiempo restante del combate se muestra mediante un contenedor numérico.

Contenedores de imagen, que representan una imagen o animación. En el DSL se representan mediante una caja con una letra i en su interior. En *Street Fighter IV*, los retratos de los dos luchadores se muestran mediante un contenedor de imagen.

Contenedores de progreso, que representan el valor relativo de un atributo respecto a su máximo y mínimo mediante una barra de progreso o una sucesión de imágenes. En el DSL se representa mediante una caja con una flecha en su interior (indicando la dirección y sentido del progreso) o bien una flecha con una letra i (indicando que el progreso se muestra mediante una sucesión de imágenes). En *Street Fighter IV*, la salud de los dos luchadores se muestra mediante una barra de progreso horizontal, mientras que el número de asaltos ganados en el combate se muestra mediante una sucesión de imágenes.

3.2 Modelo Específico de la Plataforma

En el desarrollo de videojuegos se utiliza una gran variedad de plataformas tecnológicas de juego como los PCs, videoconsolas, navegadores Web y dispositivos móviles. Del mismo modo, se utilizan diversas plataformas tecnológicas de desarrollo como lenguajes de programación, SDKs, *game engines* y *middleware* específico. Para lidiar con esta complejidad tecnológica, se utilizan modelos PSM para conceptualizar cada plataforma tecnológica de juego y desarrollo.

En este trabajo se ha utilizado *Microsoft XNA Game Studio* como plataforma de desarrollo, ya que ofrece un *framework* sencillo con primitivas específicas para el desarrollo de videojuegos para dos plataformas de juego distintas: PC y XBOX 360.

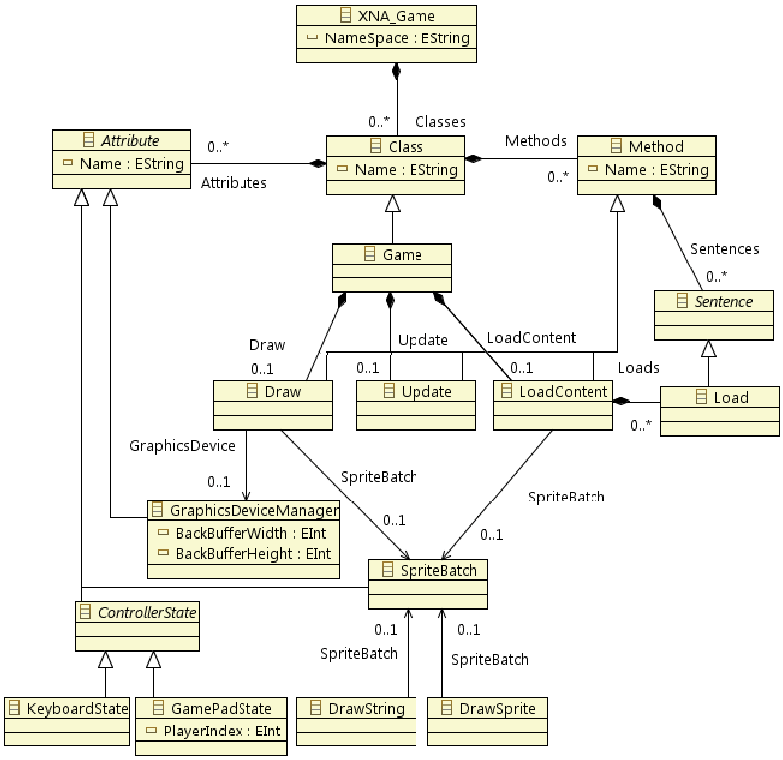

Fig. 11. Subconjunto del meta-modelo PSM para *Microsoft's XNA Game Studio*.

Siguiendo el meta-modelo PSM de *Microsoft XNA Game Studio* (ver Figura 11), un juego en XNA está formado por un grupo de clases en C# y una clase de juego *Game*. Las clases C# contienen atributos y declaraciones de métodos, como en cualquier lenguaje de programación de propósito general. La clase de juego contiene método especializados para gestionar las tareas más habituales del desarrollo de videojuegos:

El método **Initialize** se encarga de dar valores iniciales a los atributos de juego (y, por tanto, es llamado una única vez inmediatamente después de empezar el juego y declarar las variables en la clase de juego).

El método **LoadContent** carga en memoria el contenido artístico del juego: imágenes, animaciones, efectos de sonido, música y fuentes. La clase auxiliar *Content* ofrece un método *Content.Load* que permite cargar todo tipo de contenido artístico.

El método **Update** típicamente incluye la lógica y comportamiento del juego, ya que se ejecuta continuamente como parte del ciclo de juego. Para verificar y cambiar el estado de juego se utilizan sentencias condicionales de tipo if-then-else. Además, para facilitar el acceso a los controladores, XNA ofrece clases auxiliares *Keyboard* y *GamePad* para acceder al estado del teclado y del mando XBOX 360 respectivamente.

El método **Draw** dibuja en pantalla los gráficos del juego. XNA ofrece un controlador gráfico *GraphicDeviceManager* para este propósito. Así mismo, ofrece una clase *SpriteBatch* que permite dibujar primitivas 2D en pantalla.

3.3 Transformaciones para el Desarrollo de Videojuegos

La transformación manual del diseño de juego a la implementación es uno de los pasos de mayor importancia en el desarrollo de un videojuego. Esta transformación requiere un gran esfuerzo humano ya que los conceptos de diseño de alto nivel están semánticamente muy alejados de los conceptos de implementación y código fuente. Sin embargo, en una aproximación MDA al desarrollo de videojuegos, el diseño de alto nivel está precisamente especificado en un modelo PIM que puede transformarse automáticamente a modelos más concretos tecnológicamente (modelos PSM), siendo finalmente compilado a código.

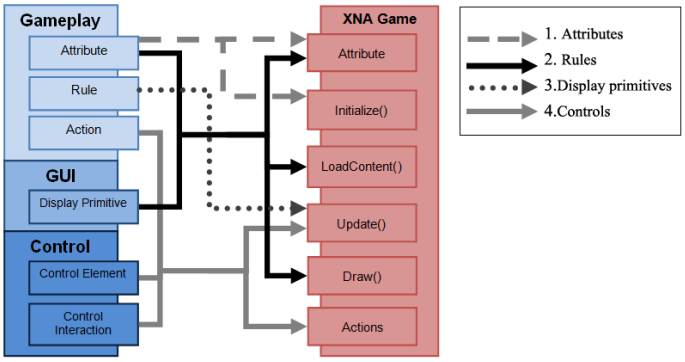

Fig. 12. Esquema general de las transformaciones entre modelos PIM y PSM.

La Figura 12 muestra un esquema de las transformaciones entre modelos PIM y PSM. Las primitivas del meta-modelo PIM, que representan conceptos del diseño de juegos en las perspectivas de jugabilidad, control y GUI son transformadas en primitivas del meta-modelo PSM, que representan conceptos de implementación en la plataforma tecnológica *Microsoft XNA Game Studio*. Por tanto, la transformación de diseño a implementación de videojuegos es responsable de las siguientes tareas de alto nivel (correspondientes a las distintas flechas en la Figura 12):

1. Crear e inicializar los atributos del diseño de juegos. Para cada entidad de juego del modelo PIM (personaje jugador, personaje no jugador o entidad

pasiva de juego), todos sus atributos se transforman en atributos de un tipo específico del modelo PSM (int, float, bool, string). Esta transformación declara los correspondientes atributos y los inicializa en el método *Initialize* de la clase de juego.

2. Crear el conjunto de reglas de la jugabilidad. Todas las reglas del modelo PIM se transforman en sentencias condicionales if-then-else en el método *Update* de la clase de juego. Las reglas de acción y de objetivo tienen prioridad respecto a las reglas internas, por lo que se comprobarán antes en el ciclo de juego.
3. Crear todas las primitivas de representación. Por cada par de atributo y primitiva de representación del modelo PIM, se declara un grupo de atributos en la clase de juego para almacenar la posición y tamaño de las primitivas de representación. Tras inicializar dichos atributos en el método *Initialize* con sus valores por defecto, las imágenes necesarias se cargan en el método *LoadContent* y, por último, las primitivas de representación se dibujan en el método *Draw*.
4. Crear las interacciones de control. Para cada par de acción de personaje jugador e interacción de elemento de control, se crea un método para definir el comportamiento asociado a la regla de acción. Los métodos de acción son llamados desde el método *Update*. Esta transformación debe ser complementada manualmente por el modelador PSM con los detalles específicos a cada elemento de control (como por ejemplo a qué tecla o botón se asocia una interacción con un elemento de control 0-dimensional).

Finalmente, tras aplicar la transformación de modelos PIM a PSM, se utiliza un sencillo compilador de modelos PSM para generar el código fuente C# correspondiente a la plataforma tecnológica *Microsoft XNA Game Studio*. Ya que los conceptos del meta-modelo PSM son muy próximos a los conceptos utilizados en el código fuente, esta transformación no merece explicación en mayor detalle.

Una vez integrado el código fuente en el SDK ofrecido por *Microsoft XNA Game Studio*, los programadores pueden aportar más detalles tecnológicos o simplemente compilar el proyecto obteniendo una versión ejecutable del juego.

4 Conclusiones

El objetivo de este artículo es ofrecer un modelo PIM para la especificación de videojuegos en tres perspectivas fundamentales: jugabilidad, GUI y control. En cada una de las perspectivas se utilizan modelos basados en UML que utilizan un DSL que ofrece una sintaxis concreta visual e intuitiva. Este multi-modelo PIM para el diseño de juegos permite, además, presentar una metodología MDA aplicada al desarrollo de videojuegos.

Nótese que la definición del meta-modelo PIM es totalmente general y permite especificar videojuegos de cualquier tipo, género de jugabilidad y plataforma tecnológica. Únicamente se ha restringido la expresividad del meta-modelo GUI, limitándolo por simplicidad a la especificación de videojuegos 2D. Como muestra de

ello, durante el artículo se han utilizado dos ejemplos (*Pac-Man* y *Street Fighter IV*) de distinto género, apariencia y tecnología, que sin embargo son especificados utilizando las mismas primitivas de alto nivel.

Aplicar MDA al desarrollo de videojuegos aporta además ventajas inherentes como la extensibilidad de los meta-modelos y la capacidad de adaptación tecnológica. Los desarrolladores de juegos pueden extender los meta-modelos para adaptarlos a nuevos requisitos del diseño de videojuegos, así como a nuevas plataformas tecnológicas. De forma similar, pueden definir nuevas transformaciones entre modelos PIM y PSM para dar soporte tecnológico a otras plataformas tecnológicas, una práctica muy habitual en el entorno actual de desarrollo multi-plataforma.

Como trabajos futuros, se incluye la definición de otras perspectivas de juego como el diseño de niveles, la historia o la inteligencia artificial. Estas perspectivas permitirán especificar juegos más complejos, cubriendo un mayor espectro de los videojuegos que se desarrollan en la actualidad. Otra posible línea de investigación sería estudiar métodos de evaluación temprana de la usabilidad y la jugabilidad de los modelos PIM, como factores clave en la calidad del videojuego.

Agradecimientos. Queremos agradecer la asistencia del Ministerio de Ciencia y Tecnología ya que la investigación se ha financiado como parte del proyecto MULTIPLE, con referencia TIN2009-13838. También queremos agradecer a la Universidad Politécnica de Valencia por la beca de Formación de Personal Investigador con referencia 199880998.

Referencias

1. Altunbay, D., Metin, M. G., Çetinkaya, M. E.: Model-driven Approach for Board Game Development. In: First Turkish Symposium of Model-Driven Software Development (TMODELS), Ankara, Turkey (2009).
2. Bast, W.: The Essence of Model Driven Architecture. In: Jax Magazine (2006). http://www.jaxmag.com/itr/online_artikel/psecom,id,548,nodeid,147.html
3. Blow, J.: Game Development: Harder Than You Think. In: ACM Queue, vol. 1, no. 10 (2004).
4. Dobbe, J.: A Domain-Specific Language for Computer Games, Master thesis (2007), TU Delft.
5. Crawford, C.: On Game Design, pp 76-78, New Riders Publishing (2003).
6. Fullerton, T., Swain, C., Hoffman, S.: Game Design Workshop: Designing, Prototyping, and Playtesting Games, CMP Books (2004).
7. Furtado, A. W. B., Santos, A. L. de M.: Using Domain-Specific Modeling towards Computer Game Development Industrialization (2006), DSM'06.
8. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley (2003).
9. OMG web site, How Systems Will Be Built, http://omg.org/mda
10. Pac-Man, developed by Namco (1980).
11. Rollings, A., Adams, E.: Andrew Rollings and Ernest Adams on Game Design, pp 13-17. New Riders Publishing (2003).
12. Schell, J.: The Art of Game Design: A Book of Lenses, pp 382-384, Morgan Kaufmann Publishers (2008).
13. Street Fighter IV, developed by Capcom (2009).