UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMATICA

Departamento de Ingeniería del Software e Inteligencia Artificial

TESIS DOCTORAL

Metodología ontológica para el desarrollo de videojuegos

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

David Llansó García

Directores

Pedro Antonio González Calero
Marco Antonio Gómez Martín

Madrid, 2014

© David Llansó García, 2014

Metodología ontológica para el desarrollo de videojuegos

es-una

tesis doctoral

realizada-por

David LLansó García

tutelada-por

director

fue

fue

Pedro Antonio González Calero

Marco Antonio Gómez Martín

Universidad Complutense de Madrid

Metodología ontológica para el desarrollo de videojuegos

TESIS DOCTORAL

David Llansó García

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Enero 2014

Documento maquetado con TEXiS v.1.0+.

Este documento está preparado para ser impreso a doble cara.

Metodología ontológica para el desarrollo de videojuegos

Memoria que presenta para optar al título de Doctor por la Universidad Complutense de Madrid en Ingeniería Informática

David Llansó García

Dirigida por

Pedro Antonio González Calero

Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial

Facultad de Informática

Universidad Complutense de Madrid

Enero 2014

Copyright © David Llansó García
Universidad Complutense de Madrid

Agradecimientos

La elaboración de esta tesis ha sido posible gracias a la concesión de una beca de Formación del Personal Investigador, concedida por la Universidad Complutense de Madrid.

Quiero agradecer, en primer lugar, a mis directores de tesis, Pedro y Marco, su dirección, consejo y apoyo durante todos estos años, tanto en los trabajos previos a esta tesis como en la investigación que se expone en este documento. Sé que no ha sido un camino fácil, ni para ellos ni para mi, pero también sé que sin ellos no hubiese sido posible. Quiero también agradecer especialmente a Pedro Pablo su implicación ya que, aunque no conste como director de esta tesis, ha estado altamente involucrado en toda la investigación que ha dado pie a este trabajo.

Por supuesto agradecer a todos los compañeros de despacho que he tenido durante estos años: Almu, Antonio, Fede, Gonzalo, Guille, Isma, Javi, Lara, Manu, Raquel, Samer, Sandra y Virginia. Gracias por haberme echado una mano cuando lo he necesitado y por esas necesarias palmeras de chocolate para desconectar. Este agradecimiento lo hago también extensible a todos aquellos compañeros de departamento y facultad con los que he compartido muchas comidas y charlas de pasillo.

En plano personal quiero empezar agradeciendo a mis padres, Enrique y Paloma, los cuales me han dado una educación y la oportunidad de formarme académicamente, apoyándome en todas mis decisiones y ayudándome en los malos momentos. A mis hermanos, Leticia y Numa, por echarme una mano siempre que han podido, mi abuela, madrina y resto de la familia que siempre habéis estado ahí.

Por último agradecer a todos mis amigos, por haberme permitido desconectar ofreciéndome un plan, pádel, tenis, fútbol o viaje siempre que hacía falta, haciendo mucho más llevaderos todos estos años. Kik, Rober, Cosc, Edu, Jaime, Al, Xolo, Santos, Sebas, Elena, Mora y todos aquellos de los que sin querer me olvido en estas líneas, gracias.

Resumen

El desarrollo software en general, y el desarrollo de videojuegos en particular, ha sufrido grandes transformaciones en los últimos años. Hoy en día suelen primar las metodologías ágiles, las cuales proponen un desarrollo software iterativo en el que se busca la generación de software funcional cada poco tiempo. En el desarrollo de videojuegos es especialmente importante probar desde el primer día las mecánicas de juego para validarlas, ya que la naturaleza multidisciplinar del equipo de desarrollo complica la comunicación y los requisitos de juego pueden variar durante el desarrollo. Por este motivo, en los desarrollos profesionales, la arquitectura software responsable de gestionar las partes más susceptibles a cambios es muy flexible, aunque el precio que paga es que es poco intuitiva y propensa a tener problemas de consistencia.

En esta tesis proponemos una nueva metodología de desarrollo de videojuegos que gire en torno a un dominio formal para, con ello, mitigar los problemas que existen a día de hoy en la industria. En esta aproximación, el dominio de juego, en vez de especificarse mediante lenguaje natural, se especifica mediante el uso de una herramienta visual, con la que se crea ese dominio formal de gran contenido semántico almacenado en una ontología. Con el uso de la herramienta se pueden tener diferentes vistas del dominio para los diferentes miembros del grupo de desarrollo, de manera que este dominio es usado como medio de comunicación entre los diferentes roles, ya que reduce ambigüedad y es mucho menos costoso de crear y modificar. Además, gracias a su riqueza semántica, es también usado en el resto de fases del desarrollo, detectando las inconsistencias durante la edición, proporcionando métodos de distribución automática de funcionalidad, guiando la generación procedimental de contenidos y permitiendo realizar grandes refactorizaciones de alto nivel que se trasladan automáticamente a la implementación de manera transparente al usuario.

Como prueba del correcto funcionamiento de la metodología y las técnicas se ha creado *Rosette*, una herramienta que las implementa, y se han realizado unos experimentos que validan nuestras premisas.

Abstract

The software development in general, and game development in particular has undergone big changes in recent years. Today agile methodologies are the design of choice, which propose an iterative software development that looks for generating functional software as fast as possible. In game development is especially important to test from the first day the game mechanics to validate them because the multidisciplinary nature of the development team complicates the communication and the gaming requirements may change during the development. For this reason, in professional development, software architecture responsible for managing the part most susceptible to changes is very flexible, but the price we pay is that it is not intuitive and prone to consistency problems.

In this thesis we propose a new methodology for developing games that is based on a formal domain thus mitigate the problems that exist today in the industry. In this approach, the game domain rather than being specified in natural language is specified by using a visual tool, which is used to create a formal domain with semantic content stored in an ontology. The tool offers different views of the domain for different members of the development team, so this domain is used for communicating the different roles, reducing ambiguity and being much less expensive to create and modify. Moreover, thanks to their semantic content, the ontology is also used in other phases of development, detecting inconsistencies during the edition, providing methods for automatic distribution of functionality, guiding procedural content generation and allowing to make big high-level refactorings that automatically trigger the changes to the implementation without the intervention of designers and programmers.

As a proof of the proper functioning of the methodology and techniques we created *Rosette*, a tool that implements them, and some experiments have been carried out to validate our assumptions.

Índice

Agradecimientos V

Resumen VII

Abstract IX

1. Introducción 1

1.1. Motivación . . . 1
1.2. Objetivos . . . 3
1.3. Resumen de las Contribuciones . . . 6

2. Metodología y arquitectura en el desarrollo de juegos 11

2.1. Diferentes roles del equipo de desarrollo . . . 11
2.1.1. Artista . . . 12
2.1.2. Diseñador . . . 12
2.1.3. Programador . . . 12
2.1.4. Otros roles . . . 13
2.2. Fases del desarrollo . . . 13
2.2.1. Pre-producción . . . 14
2.2.2. Producción . . . 15
2.2.3. Post-producción . . . 16
2.2.4. Mantenimiento . . . 17
2.3. Metodologías Ágiles: Scrum . . . 17
2.4. Arquitectura software de videojuegos . . . 19
2.4.1. Arquitectura general . . . 19
2.4.2. Arquitecturas software para las entidades de juego . . . 21

3. Estado del Arte 29

3.1. Documento de diseño como herramienta de comunicación entre diseñadores y programadores . . . 29
3.2. Representación formal del dominio de juego . . . 31

3.3. Generación de contenido de juego 36
3.4. Falta de herramientas en diseño 37

4. Metodología de desarrollo basada en ontologías 41
4.1. Introducción 41
4.2. Modelado del dominio como contrato entre programadores y diseñadores de videojuegos 43
4.2.1. Dominio formal como complemento al documento de diseño 43
4.2.2. Representaciones del dominio para los diferentes roles 46
4.3. Procedimientos efectuados en el desarrollo de la metodología iterativa 49
4.3.1. Puntos potencialmente conflictivos 57
4.3.2. ejemplo 59
4.4. Representación ontológica de los diferentes elementos . . . 64
4.4.1. Ejemplo 70

5. Técnicas de soporte a la metodología 75
5.1. Identificación automática de componentes software 75
5.1.1. Introducción 75
5.1.2. Análisis formal de conceptos 76
5.1.3. Metodología para la obtención del retículo 78
5.1.4. Infiriendo componentes del retículo 81
5.1.5. Refinamiento experto 85
5.1.6. Ejemplo de sugerencia de componentes 88
5.1.7. Desarrollo iterativo usando FCA 90
5.1.8. Ejemplo de desarrollo iterativo 91
5.1.9. Conclusiones 93
5.2. Generación Procedimental de Contenido 95
5.2.1. Introducción 95
5.2.2. Contenido generado a partir de la descripción semántica 96
5.2.3. Iteratividad 102
5.2.4. Conclusiones 105
5.3. Chequeo de inconsistencias 106
5.3.1. Introducción 106
5.3.2. Inconsistencias a nivel de entidad 108
5.3.3. Inconsistencias debidas a la configuración del nivel . . 118
5.3.4. Conclusiones 127

6. Rosette 129
6.1. Una herramienta de autoría para un desarrollo ontológico . 129
6.2. Identificación automática de componentes software 136
6.3. Generación procedimental de contenido 138
6.4. Detección de inconsistencias 145

7. Experimentos 151
7.1. Metodología para enseñanza de la arquitectura basada en componentes 151
7.1.1. Introducción 151
7.1.2. Metodología 152
7.1.3. Metodología en práctica 155
7.1.4. Evaluación 162
7.1.5. Conclusiones 167
7.2. Evaluación empírica de la identificación automática de componentes software 168
7.2.1. Introducción 168
7.2.2. Metodología 169
7.2.3. Evaluación 171
7.2.4. Conclusiones 181

8. Conclusiones y trabajo futuro 183
8.1. Contribuciones 183
8.2. Discusión de los resultados 185
8.3. Trabajo futuro 186

9. Ontological methodology for game development 189
9.1. Introduction 189
9.2. Goals 191
9.3. Development methodology based in ontologies 193
9.3.1. Methodology 194
9.3.2. Using Ontologies for Modelling Entities 199
9.4. Automatic identification of software components 203
9.4.1. Formal Concept Analysis 204
9.4.2. Generating components through Formal Concept Analysis 206
9.4.3. Expert Tuning 210
9.4.4. Iterative Software Development with FCA 213
9.5. Consistency checking 216
9.5.1. Entity inconsistencies at a component level 217

9.5.2. Inconsistencies due to attribute definitions 218
9.5.3. Entity inconsistencies due to a level configuration . . 218
9.5.4. Example 220
9.6. Rosette 226
9.7. Methodology to teach the component-based architecture . . 230
9.7.1. Methodology 230
9.7.2. Methodology in Practice 232
9.7.3. Evaluation 238
9.7.4. Conclusions and Future Work 242
9.8. Empirical evaluation of the automatic generation of component-based architecture 243
9.8.1. Methodology 243
9.8.2. Evaluation 245
9.8.3. Conclusions 253
9.9. Conclusions and future work 254
9.9.1. Discussion of the results 256
9.9.2. Future work 257

Bibliografía **259**

Índice de figuras

2.1. Vista de una arquitectura de videojuegos en su forma más simple 20
2.2. Árbol de herencia tradicional 22
2.3. Ejemplo de invocación usando RTTI 24
2.4. Ejemplo de la descripción de una entidad *Soldier* 27

4.1. Metodología de desarollo basada en un dominio formal . . . 49
4.2. Vista parcial de la jerarquía de entidades del *Half-Life* 60
4.3. Mensajes y atributos con los que se definiría la entidad *Zombie* del *Half-Life* 62
4.4. Código del método *Turn* ya implementado por los programadores 63
4.5. FSM simplificada del comportamiento de un *Zombie* 63

5.1. Descripción ontológica de las entidades de juego 79
5.2. Retículo de conceptos 81
5.3. Conjunto de componentes obtenidos de la aplicación de nuestra técnica 83
5.4. Conjunto de componentes creados desde cero por un programador experto sin el uso de ninguna técnica 88
5.5. El nuevo retículo de conceptos formales 92
5.6. Nuevo conjunto de componentes propuesto 94
5.7. Jerarquía de entidades de juego 110
5.8. Entidad NPC definida en función de sus componentes, mensajes (acciones y actuadores) y atributos 111
5.9. Ejemplo de un nivel con el dominio implementado en Unity . 122
5.10Simple FSM 123

6.1. Vista general de la arquitectura de *Rosette* 130
6.2. Vista de diseño de *Rosette* 132

6.3. Vista de programación de *Rosette* 133
6.4. Vista de un mensaje para los programadores 134
6.5. Vista de un atributo para los programadores 135
6.6. Vista de un componente para los programadores 136
6.7. Pantalla donde los programadores anotan similitudes entre componentes 138
6.8. Una plantilla para la cabecera de los componentes en C# de *Unity* 141
6.9. Una plantilla para la cabecera de los componentes en C++ . 142
6.10Una plantilla para la implementación de los componentes en C++ 143

7.1. Primer juego, creado usando Unity 156
7.2. Segundo juego, construido en C++ desde cero 159
7.3. Declaración: *Rosette* y la metodología, para enseñar (o aprender) qué es una arquitectura basada en componentes, es muy útil 161
7.4. Declaración: La generación iterativa de contenido de *Rosette* es muy útil cuando el dominio es modificado, ya que preserva todo el código implementado. 163
7.5. Número de estudiantes que finalizaron las diferentes iteraciones del primer juego 164
7.6. Número de estudiantes que finalizaron las diferentes iteraciones del segundo juego 165
7.7. Grupos de estudiantes con diferentes progresión y calidad en sus distribuciones de componentes 166
7.8. Falta de cohesión de las distribuciones por iteración 172
7.9. Acoplamiento de las distribuciones por iteración 175
7.10Funcionalidad duplicada en las diferentes iteraciones 178
7.11Número de operaciones hechas en cada iteración 180

9.1. development methodology based in a formal domain 195
9.2. Ontological description of game entities 206
9.3. Concept lattice 208
9.4. The candidate components proposed 210
9.5. New concept lattice 215
9.6. The new candidate components proposed by the technique . 216
9.7. Example distribution 222
9.8. Unity example 223
9.9. Simple FSM 223

9.10. OWL definitions for entities, components, messages and attributes . . . 224
9.11. General view of the *Rosette* architecture . . . 226
9.12. First game, created using Unity . . . 233
9.13. Second game, build in C++ from scratch . . . 235
9.14. Statement: *Rosette* and the methodology, to teach (or learn) what a component-based architecture is, is very useful . . . 236
9.15. Statement: The iterative code generation of *Rosette* is very useful when the domain is modified because it preserves all the code implementation . . . 238
9.16. Number of students that finished the different iterations of the first game . . . 240
9.17. Number of students that finished the different iterations of the second game . . . 240
9.18. Groups of students with different progression and quality of their component distributions . . . 241
9.19. Lack of cohesion of the distributions per iteration . . . 246
9.20. Coupling of the distributions per iteration . . . 249
9.21. Duplicated functionality in the different iterations . . . 251
9.22. Number of operations done per iteration . . . 252

Índice de Tablas

5.1. Parte del *contexto formal* del dominio de juego 80

9.1. Partial formal context of the game domain 207

Capítulo 1

Introducción

1.1. Motivación

Desde la aparición de los primeros videojuegos hasta el día de hoy los juegos y su proceso de desarrollo se han recorrido un largo camino. Los primeros videojuegos no mostraban más que unos puntos, líneas o figuras geométricas en la pantalla y se controlaban mediante el uso de simples potenciómetros o pulsadores. Sin embargo, hoy en día, muestran gráficos 3D extremadamente realistas y, los modos de control, son tan abundantes como diversos, yendo desde teclado o mandos hasta reconocimiento de movimiento mediante el uso de cámaras. El aumento de la popularidad de los videojuegos ha provocado un incremento en la exigencia y, por tanto, el número de personas involucradas en un desarrollo ha ido creciendo con los años. Antiguamente uno o dos programadores se encargaban de todo el proceso de desarrollo mientras que hoy en día, las grandes producciones, involucran a una gran cantidad de gente que cumplen diferentes roles dentro del desarrollo.

Debido a este fenómeno de especialización del equipo, ahora hay diferentes roles dentro de un desarrollo, donde ya no existen solamente programadores que se encarguen de todas las tareas (al menos en desarrollos medios y grandes), sino que ahora se tiene un equipo multidisciplinar de personas que provienen de áreas muy diferentes. Aparte de los programadores, que implementan la funcionalidad del juego, y los artistas, que generan los recursos gráficos, podemos destacar el rol de diseñador. Los diseñadores son los encargados del aspecto creativo del desarrollo y se dedican a idear y desarrollar la jugabilidad del juego, sus mecánicas, la historia de trasfondo con sus posibles acontecimientos y diálogos, los tipos de personajes, cómo éstos deben comportarse, etc. pero no se les presumen en principio nociones de programación (aunque

el buen diseñador es aquel que tiene conocimiento de todas las áreas, aunque sea un conocimiento global).

Al incremento de la exigencia y la dificultad de gestionar un grupo multidisciplinar se le suma el hecho de que el diseño de mecánicas de juego requiere de un desarrollo rápido de funcionalidad para probar si las ideas diseñadas “funcionan” o debe replantearse el diseño del juego. Todas estas cuestiones provocan que las metodologías clásicas de desarrollo software no sirvan y se deba optar por metodologías actuales, más flexibles y que asimilan relativamente bien cambios en la especificación, primando la velocidad de desarrollo frente a la sobreingeniería. El desarrollo de mecánicas o características del juego suele ser un proceso iterativo donde las ideas que van surgiendo se documentan, se implementan, se evalúan y, en función de esa evaluación, se descartan, se aceptan o se refinan, volviendo a pasar de nuevo por todas las fases anteriores.

Con el paso de los años la tecnología usada para soportar este tipo de metodologías ha ido evolucionando y, hoy en día, prácticamente todo los juegos del mercado usan una arquitectura basada en componentes (Sección 2.4.2) para implementar las diferentes entidades u objetos del juego. Este tipo de tecnología es muy flexible, permitiendo todo tipo de modificaciones en las especificaciones y premiando la reusabilidad de código. No obstante estas arquitecturas, basadas en composición en lugar de la herencia, complican la legibilidad y depuración del código ya que la funcionalidad de las entidades de juego se encuentra fraccionada y repartida por varios componentes software. La distribución de funcionalidad en componentes permite que la creación de las entidades de juego esté dirigida por datos, retrasando la formación de las entidades hasta la ejecución del juego. Esto aporta velocidad a la hora de formar entidades nuevas o modificar ya creadas y ahorra grandes tiempos de compilación, pero dificulta también la comprensión a alto nivel de qué es cada entidad, ya que se limita a un conjunto de componentes/habilidades perdiendo el significado ontológico que aporta una jerarquía y, además, traslada muchos de los errores que antes podían detectarse en tiempo de compilación a tiempo de ejecución, retrasando su detección y complicando la identificación del problema.

La especialización en roles y el aumento del número de integrantes ha introducido un problema concreto en el desarrollo que es la comunicación entre los diferentes integrantes. Históricamente esta comunicación se ha realizado mediante un *documento de diseño* escrito por los diseñadores en lenguaje natural. No obstante, si como hemos comentado la tecnología ha tenido que adaptarse para permitir un desarrollo

ágil y flexible, la comunicación existente debe también evolucionar para permitir optimizar el proceso de desarrollo. Al tratarse de un documento básicamente textual su naturaleza es estática por lo que los cambios y modificaciones que se deban hacer sobre la especificación inicial son bastante costosos de realizar (cuesta describir y actualizar pormenorizadamente una funcionalidad) y la comunicación es muy lenta (el lenguaje natural es muy expresivo pero es bastante más costoso de leer o interpretar que una representación mas esquemática y visual).

Los problemas que se han resaltado afectan a dos de los roles más importantes en el desarrollo de videojuegos y provocan importantes retrasos y cuellos de botella en la aplicación de técnicas de desarrollo ágil. Además, el flujo tal y como se ha explicado, donde los diseñadores plasman el conocimiento en un documento de diseño y los programadores replican dicho conocimiento en código, quedando el comportamiento de las entidades troceado y oculto en él, complica la reutilización de código y provoca una duplicidad de la información, donde es fácil que se pierda la equivalencia entre lo que está documentado en el documento de diseño y el resultado final de la implementación.

1.2. Objetivos

El objetivo de esta tesis es ofrecer una metodología de desarrollo que minimice la ambigüedad de las especificaciones, evite la duplicidad de información, mejore la velocidad de transmisión de ésta y flexibilice la descripción del dominio del juego, haciendo que sea más sencillo realizar cambios. En conclusión, una metodología que nos permita agilizar el desarrollo. Como veremos, la metodología que proponemos gira entorno a una ontología o dominio formal por lo que la pregunta de investigación que quiere responder la tesis sería:

> *¿Podría un dominio formal ayudar a paliar los hándicaps de la creación de videojuegos y a agilizar su desarrollo?*

Mediante una descripción ontológica podemos representar una gran cantidad de información semántica que antes se encontraba simplemente descrita en un documento e implementada y oculta en el código, donde los programadores tenían dificultades para ver la coherencia de la distribución final de funcionalidad, llevando a veces a la duplicación del código. Mediante este tipo de representación formal del conocimiento, los diseñadores pueden describir de forma sencilla (con algún tipo de editor visual) las entidades de juego de una manera rápida. No sólo eso

sino que pueden cambiar la descripción de una entidad, una funcionalidad, una restricción, etc. sin mayor esfuerzo. Los programadores reciben a su vez una representación del dominio más concreta donde la descripción de los elementos de juego es mucho menos ambigua y deja menos posibilidades a la interpretación. Éstos últimos a su vez usan la ontología no solo para entender qué es lo que los diseñadores quieren sino también para aprovechar todo ese conocimiento (completado por conceptos más orientados a la implementación y añadidos por ellos mismos a la ontología) durante el desarrollo. El conocimiento encapsulado en la ontología puede perfectamente usarse con un razonador para detectar inconsistencias difícilmente localizables en una arquitectura basada en componentes tradicional y también para, a partir de ese conocimiento, generar contenido de juego de manera procedimental, evitando a los programadores redefinir información en el código que ya había sido definida en la ontología. De esta manera se mantiene una relación entre los elementos definidos en la ontología y los elementos software de la arquitectura basada en componentes, sacando así a la luz todo ese conocimiento que antes se encontraba distribuido por los componentes software y que ahora se encuentra en la ontología.

Para poder llevar a cabo este objetivo se ha analizado en el Capítulo 2 cuál es la metodología que, de una manera más o menos fiel, usan a día de hoy en la industria los equipos de desarrollo para crear un videojuego. En el Capítulo 3 se ha hecho un repaso a diferentes técnicas utilizadas por otros autores para mitigar problemas similares a los que tratamos de dar aquí una respuesta. Dentro de esta investigación acerca del estado del arte se ha profundizado en corroborar que otros autores y diseñadores de la industria habían identificado de una u otra manera los problemas existentes en la transmisión de la información a través de un documento de diseño tradicional, como otros autores han usado diversas representaciones semánticas para definir ciertos aspectos de los videojuegos y cómo a partir de esas (u otras) descripciones semánticas generan contenido de código de manera procedimental. Por último se hace un repaso a alguna de las herramientas que se encuentran tanto en la industria como en el ámbito académico para realizar las representaciones del conocimiento de una manera más sencilla y rápida.

El Capítulo 4 describe la base de esta tesis, explicando cuál es nuestra propuesta metodológica para el desarrollo de videojuegos, cual sería el flujo del trabajo y las interacciones entre diseñadores y programadores, cómo representamos y relacionamos nosotros en la ontología los diferentes conceptos del juego y cómo dicha ontología sirve como contrato entre diseñadores y programadores. Por otro lado, en el Capítulo 5,

se explican ciertas técnicas específicas y complementarias que, gracias a tener un dominio formal como descripción del juego, permiten acelerar el desarrollo y mejorar la calidad del software finalmente obtenido. Una de estas técnicas nos permite la identificación automática de la mejor distribución en componentes software a partir de la funcionalidad especificada en las entidades. También se muestra cómo se puede ir modificando dicha distribución iterativamente mediante adición, modificación o supresión de funcionalidad, manteniendo siempre coherencia con fases pasadas del desarrollo. Por otro lado, a partir de la ontología se explica cómo generamos procedimentalmente contenido de juego. Este contenido sirve tanto a los programadores, produciendo gran cantidad del código de la lógica de juego, como para los diseñadores, generando contenido para ser usado en los editores de niveles, mapas y comportamiento. Estas dos técnicas están estrechamente relacionadas ya que, debido a la relación existente entre los elementos especificados en la ontología y el contenido generado, se puede modificar la ontología con la seguridad de que la generación de contenido trasladará esos cambios al contenido manteniendo todo el trabajo previo realizado por programadores y diseñadores. Para finalizar el Capítulo 5 se explica cómo usar el conocimiento semántico para, mediante el uso de un razonador, poder encontrar inconsistencias. Esas inconsistencias engloban tanto aquellas que se encuentran en la descripción del dominio del juego a nivel de entidad como aquellas detectadas en los diferentes mapas creados por los diseñadores. De esta manera se permite mantener la coherencia y detectar fallos a nivel de diseño que, de otra manera, sería imposible de detectar hasta la ejecución del juego. A su vez, mejora el uso de las técnicas anteriores ya que es posible detectar inconsistencias ante cambios en el dominio, distribución de los componentes o cambios en el contenido de juego generado (por ejemplo mapas de niveles creados por los diseñadores con instancias concretas de entidades).

Con la idea de comprobar la validez del trabajo realizado, hemos desarrollado un prototipo llamado *Rosette* que sirve como herramienta gráfica de autoría y que soporta la metodología y las técnicas comentadas previamente. En el Capítulo 6 se cuenta cómo ha sido desarrollado *Rosette* y cómo se han implementado las esas características previamente explicadas. *Rosette* ha sido usada por alumnos del Máster de videojuegos de la Universidad Complutense de Madrid[1] y, gracias a ello, hemos podido realizar dos experimentos con los que comprobar la validez de nuestra metodología. Ambos experimentos se encuentran explicados en el Capítulo 7. El primero de ellos, prueba cómo el uso de

[1] http://www.videojuegos-ucm.es/

nuestra metodología y, más concretamente, de *Rosette* ayuda a entender mejor la arquitectura basada en componentes y sirve como herramienta de aprendizaje ya que la abstracción semántica y su separación de los detalles concretos de implementación permite entender mejor los conceptos. Además, la generación procedimental de contenido permite poder realizar dos pequeños juegos en un tiempo asumible, que permite centrarnos en un nivel de comprensión más alto, evitando detalles relacionados con el código. En el segundo experimento, lo que fue hizo es valorar la técnica de sugerencia de componentes, analizando si las sugerencias ofrecidas por esta técnica acelera el desarrollo y si las distribuciones de funcionalidad en componentes software son de mayor o menor calidad que las generadas por un experto. A tenor de los resultados pudimos afirmar que la técnica de sugerencia de componentes mejora la calidad de la distribución final y que a la larga incrementa la velocidad de desarrollo.

Finalmente, en el Capítulo 8, se recogen las conclusiones obtenidas de la realización de esta tesis así como una discusión de los resultados y un pequeño resumen de las posibles líneas de trabajo futuro.

1.3. Resumen de las Contribuciones

El trabajo recogido en esta tesis ha sido publicado en una serie artículos que se enumeran a continuación:

- Gonzalo Flórez Puga, David Llansó, Marco Antonio Gómez-Martín, Belén Díaz Agudo, Pedro Pablo Gómez-Martín y Pedro Antonio González-Calero. *Empowering Designers with Libraries of Self-validated Query-enabled BehaviourTrees*. Springer. Marzo 2011. P. 55-82. ISBN 978-1-4419-8187-5 (edición impresa), 978-1-4419-8188-2 (edición electrónica).

- Antonio Sánchez-Ruiz, David Llansó, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Authoring Behaviours for Game Characters Reusing Automatically Generated Abstract Cases*. 8th International Conference on Case-Based Reasoning, ICCBR 2009. P. 129-137. Seattle, Washington, Estados Unidos. Julio 2009. ISBN 978-3-642-02997-4.

- Antonio Sánchez-Ruiz, David Llansó, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Authoring behaviour for characters in games reusing abstracted plan traces*. The 9th International

Conference Intelligent Virtual Agents (IVA 2009). P. 56-62. Amsterdam, Holanda. Septiembre 2009. ISBN 978-3-642-04379-6 (edición impresa), 978-3-642-04380-2 (edición electrónica). Categoría B según el ranking CORE 2013. Tasa de aceptación: 19 % (19 de 98).

- David Llansó, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Self-Validated Behaviour Trees through Reflective Components*. The Fifth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2009). P. 70-75. Palo Alto, California, Estados Unidos. Octubre 2009. ISBN 978-1-57735-431-4 (edición impresa), 978-3-642-04380-2 (edición electrónica).

- David Llansó, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín y Pedro Antonio González-Calero. *Explicit Domain Modelling in Video Games*. The 6th International Conference on the Foundations of Digital Games (FDG 2011). P. 99-106. Burdeos, Francia. Julio 2011. ISBN 978-1-4503-0804-5. Categoría C según el ranking CORE 2013. Tasa de aceptación: 29 % (31 de 107).

- David Llansó, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Knowledge Guided Development of Videogames*. Papers from the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process (IDP 2011). P. 8-13. Palo Alto, California, Estados Unidos. Octubre 2011. ISBN 978-1-57735-542-7.

- David Llansó, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín y Pedro Antonio González-Calero. *Iterative Software Design of Computer Games through FCA*. 8th International Conference on Concept Lattices and their Applications (CLA 2011). P. 143-158. Nancy, Francia. Octubre 2011. ISBN 978-2-905267-78-8. Tasa de aceptación: 57 % (27 de 47).

- David Llansó, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, Pedro Antonio González-Calero y Magy Seif El-Nas. *Tool-supported iterative learning of component-based software architecture for games*. The 10th International Conference on the Foundations of Digital Games (FDG 2013). P. 376-379. La Canea, Creta, Grecia. Mayo 2013. Categoría C según el ranking CORE 2013. Tasa de aceptación: 30 % (45 de 149).

- David Llansó, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Domain Modeling as a Contract between Game Designers and Programmers*. Actas del

Primer Simposio Español de Entretenimiento Digital (SEED 2013). P. 13-24. Madrid, España. Octubre 2013. ISBN 978-84-695-8350-0.

- David Llansó, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *A Declarative Domain Model can serve as Design Document*. Papers from the 2013 AIIDE Workshop on Artificial Intelligence in the Game Design Process (IDP 2013). P. 9-15. Boston, Massachusetts, Estados Unidos. Octubre 2013. ISBN 978-1-57735-635-6. Tasa de aceptación: 53 % (7 de 13).

- David Llansó, Pedro Pablo Gómez-Martín, Marco Antonio Gómez-Martín y Pedro Antonio González-Calero. *Empirical Evaluation of the Automatic Generation of a Component-Based Software Architecture for Games*. P. 37-43. Boston, Massachusetts, Estados Unidos. Octubre 2013. ISBN 978-1-57735-607-3. Tasa de aceptación: 27 %.

En las siguientes líneas se especifica por capítulos dónde puede encontrarse más información:

- *Capítulo 4:* La descripción de los elementos ontológicos que usamos y sus relaciones fueron publicadas en Llansó et al. (2011a) mientras que la metodología fue publicada en Llansó et al. (2013b,a).

- *Capítulo 5:* La primera versión de la técnica de sugerencia de componentes fue publicada en Llansó et al. (2011c) mientras que su versión iterativa se publicó en Llansó et al. (2011b). La parte de generación de contenido fue publicada en Llansó et al. (2013d,b). Por último, la detección de inconsistencias en el dominio y descripción de las entidades de juego fue publicada en Llansó et al. (2011a) mientras que la detección de inconsistencias en los niveles creados por los diseñadores fue presentado en Llansó et al. (2013a). En el caso de la detección de inconsistencias teníamos ya trabajo previo relacionado Llansó et al. (2009b); Sánchez-Ruiz et al. (2009a,b); Flórez-Puga et al. (2011) que dio lugar a mi tesis de máster (Llansó et al., 2009a), la cual fue fundamental para el desarrollo de estas técnicas y sirvió de semilla para la realización de esta tesis.

- *Capítulo 6:* La herramienta se ha ido presentando poco a poco en los siguientes trabajos: Llansó et al. (2011a,c,b, 2013d,b,a,c).

- *Capítulo 7:* El experimento que valora la utilidad de *Rosette* como herramienta para asimilar y comprender mejor los conceptos

básicos de la arquitectura basada en componentes fue presentado en Llansó et al. (2013d) mientras que el experimento que prueba la validez de la técnica de sugerencia de componentes fue publicada en Llansó et al. (2013c).

Capítulo 2

Metodología y arquitectura en el desarrollo de juegos

En este capítulo se explica cómo es la metodología que se usa habitualmente en el desarrollo de videojuegos: cuales son sus fases, cuales son los miembros del equipo y los métodos de gestión del desarrollo que se suelen adoptar hoy en día. Esta metodología usada, de una manera más o menos fiel en la mayoría de los grandes y pequeños estudios, es fruto de la experiencia de mucha gente tras el desarrollo de muchos proyectos a lo largo de los años. A día de hoy sigue evolucionando mediante la adición de nuevos métodos y variación de las diferentes fases para potenciar la velocidad y la calidad de los desarrollos. Se introduce también qué tipo de arquitectura se suele usar en este tipo de desarrollos, haciendo especial hincapié en las arquitecturas que gestionan la lógica de juego y sus elementos (entidades).

2.1. Diferentes roles del equipo de desarrollo

El desarrollo de videojuegos es una tarea llevada a cabo por un equipo multidisciplinar donde confluyen perfiles de campos muy diferentes. Los roles que se necesitan varían de proyecto a proyecto ya que no es lo mismo desarrollar un juego para móvil, donde 4 personas pueden afrontar todos los roles del proyecto, que un desarrollo de un juego AAA, donde puede haber más de 80 personas, cada una con un rol asignado (Chandler, 2009). De todas maneras, podemos considerar que hay tres roles básicos que aparecen siempre en el proceso de producción: artista, programador y diseñador.

2.1.1. Artista

Los artistas suelen ser licenciados en Bellas Artes o diseño gráfico y son los responsables de la creación de todos los recursos gráficos del juego. Trabajo suyo será crear y texturizar los modelos de los personajes con sus correspondientes animaciones, elementos del mundo, niveles y todo tipo de contenido 3D así como realizar las imágenes usadas en el juego para los menús, la interfaz de usuario, la maquetación y marketing.

2.1.2. Diseñador

Los diseñadores son la cabeza pensante del proyecto, son los que deciden el rumbo que éste debe llevar y cual es el resultado que se desea obtener. Están involucrados desde el principio al fin del proyecto, siendo los encargados de desarrollar la idea inicial del juego y pensar en hasta el último detalle del producto que se quiere obtener, controlando que realmente el rumbo sea el adecuado. Entre sus responsabilidades, aparte de decidir en qué va a consistir el juego, está establecer cuáles van a ser las mecánicas principales de éste, las reglas, cómo va a interaccionar el jugador, quá respuesta visual y sonora va a obtener, cómo van a ser los personajes, cuál va a ser la historia del juego, guión, diálogos o diseñar los diferentes niveles que se tendrán.

Una de las tareas más importantes del diseñador es saber transmitir exactamente qué es lo que quiere de cada miembro del equipo y de plasmar perfectamente cuáles son los requisitos ya que en caso contrario, si deja detalles sin especificar, lo más probable es que el resultado no sea el que él esperaba y por tanto haya que rehacer cierta parte del trabajo, con el hándicap que eso supone.

2.1.3. Programador

Responsabilidad del programador es crear todo el código necesario para hacer que todas las ideas descritas por los diseñadores se hagan realidad y funcionen usando los recursos que han generado los artistas. Los programadores están involucrados en todos los aspectos del juego: gráficos, animaciones, lógica de juego, gameplay, interfaz de usuario, física, sonido, scriptado, etc. Además son también los encargados de realizar herramientas que ayudan y simplifican el desarrollo a otros integrantes del equipo: editores de niveles, editores de comportamiento de los personajes, herramientas de exportación de recursos gráficos, etc.

Los programadores suelen ser ingenieros informáticos y, debido a que están implicados en todos los aspectos del juego y son los que saben acerca de las limitaciones técnicas existentes, deben evaluar los riesgos técnicos del diseño del juego y acordar con los diseñadores qué cosas se pueden hacer y qué cosas no son viables y deben ser rediseñadas. De la misma manera, son los encargados de poner restricciones técnicas a los recursos que generan los artistas y que deben ser incluidos en el juego.

2.1.4. Otros roles

Aunque las anteriores figuras son las que destacan en el desarrollo, hay otras figuras como el músico, encargado de crear los sonidos, melodías y diálogos del juego, o roles que, aunque no participen directamente en el desarrollo son indispensables para la creación del videojuego. A veces hay un rol de director que se encarga de planificar el desarrollo y hacer de intermediario entre los diferentes roles del desarrollo, liberando de estas tareas al diseñador. El productor que se encarga de hacer todo lo demás que no tenga que ver con el juego en sí como creación de manuales, montaje de contenidos y materiales, etc. Marketing se debe encargar de vender el producto por lo que deben hacer publicidad en medios especializados, concertar entrevistas o planificar eventos y por último, el rol de testeador y control de calidad es el encargado de comprobar que el juego funciona como es debido, no hay fallos ni problemas de compatibilidad, tiene una jugabilidad adecuada y hay una buena localización.

2.2. Fases del desarrollo

El proceso de producción de un videojuego es todo el camino que recorre un equipo para transformar una idea en un producto software cerrado listo para ser distribuido. El desarrollo de videojuegos cubre un amplio espectro de desarrollos ya que dentro de este campo caben grandes proyectos de juegos AAA para PC o consolas, donde trabajan mas de 80 personas durante 2 o tres años, y pequeños desarrollos para móviles, donde grupos pequeños de 4 o 5 personas desarrollan un producto en apenas un par de meses. Este proceso obviamente varía en función del tipo de proyecto que se lleve a cabo, del grupo que lo desarrolle, de los recursos que se tengan, de la plataforma objetivo para la que se desarrolla, de la tecnología elegida y de un largo etcétera de elementos. Todos ellos pasan por una serie de fases distinguidas (Chandler, 2009), aunque no siempre perfectamente delimitadas, que son: pre-producción,

producción, post-producción y mantenimiento.

2.2.1. Pre-producción

El proceso de pre-producción es la primera fase en la construcción de un videojuego. Aunque englobe solo entre un 10 % y un 25 % del proceso completo probablemente sea la fase más importante ya que es donde se desarrolla y se da forma a la idea de la que parte el desarrollo, se crean los primeros prototipos para asegurarse de que la idea funciona, se estima el dinero y el número de personas que serán necesarias en la creación del producto en el tiempo propuesto y se documenta de manera abundante cómo debe ser el juego, cuáles son las reglas de éste, cómo interacciona el jugador, etc.

Previamente a la fase de preproducción se debe tener por escrito cuál es el concepto del juego, donde un buen concepto debe poder ser definido mediante una sola frase. Los primeros pasos de esta fase consisten en desarrollar esa idea original del juego y describir minuciosamente cómo será cada pequeño detalle. Generalmente este proceso descriptivo se plasma en un documento conocido como *documento de diseño* (Adams, 2009; Rogers, 2010). En dicho documento se describe el concepto ampliado del juego, que debe estar explicado de manera que cualquier miembro del equipo sea capaz de entender cuáles son los elementos del juego, los objetivos de éste, las reglas y el gameplay. El *documento de diseño* del juego suele extenderse a cientos de páginas de longitud (Katharine, 2012) donde, aunque hay algún elemento como flujos de datos (*data flows*), diagramas o imágenes, el grueso del documento está confeccionado en lenguaje natural. De todas formas hay otra corriente que prefiere tener documentos de diseño más ligeros y manejables aunque eso implique un menor grado de detalle. Un ejemplo extremo de esto es el caso del director creativo de Electronic Arts que experimentó con una técnica que llama *one-page design* (Librande, 2010) que presenta el diseño del juego en una sola página de manera mucho más visual.

Con la finalidad de reducir el riesgo de desarrollar un juego caro pero con una idea no satisfactoria desde el punto de vista de la jugabilidad, el equipo de producción (programadores y artistas) genera uno o varios prototipos siguiendo el *documento de diseño* y bajo la dirección de los diseñadores. Estos prototipos centran la atención en la implementación de las partes más significativas del gameplay del juego, sin tener tan en cuenta acabados finales relacionados con el arte o apartados más específicos de programación (Salen y Zimmerman, 2003). Éste es un momento crítico del desarrollo ya que los prototipos son la única manera

de evaluar la calidad de las ideas plasmadas en el *documento de diseño* (Katharine, 2012). Preguntas como "¿el juego cumple los objetivos de diseño?" o "¿los jugadores se lo pasan bien?" sólo pueden contestarse jugando (Salen y Zimmerman, 2003). Por tanto, tan pronto como se encuentre el primer prototipo disponible se debe comenzar el proceso de testeo de la jugabilidad (Swift et al., 2008).

Por último, en esta fase también se suelen generar herramientas que serán usadas durante la fase de producción del juego. Estas herramientas, tales como editores de niveles o exportadores de recursos gráficos, servirán para simplificar las tareas de producción de los diferentes miembros del grupo.

2.2.2. Producción

Una vez se ha documentado la idea del juego, se ha comprobado que ésta funciona mediante prototipos y se ha estudiado cuál va a ser el plan de desarrollo se empieza la etapa de desarrollo propiamente dicha. Puede suceder, no obstante, que la línea que separa la pre-producción de la producción a veces no esté tan claramente marcada. De esta manera se puede dar el caso de que parte del equipo se encuentre ya en producción (por ejemplo artistas realizando recursos para la versión final) mientras otra parte del equipo se encuentra aun en pre-producción (realizando los últimos prototipos o terminando las herramientas que se usarán en producción) (Chandler, 2009).

En esta fase es donde se generan todos los contenidos finales del juego y donde el equipo de desarrollo alcanza su tamaño máximo incorporando nuevos trabajadores a los que se encontraban en la fase de pre-producción. Durante la producción se debe generar no solo toda la jugabilidad y funcionalidad del juego sino también todos los recursos como modelos, texturas, sonidos, menús, etc. o todos los niveles que vaya a tener el juego final. Si todo fuese según lo previsto, en esta fase se debería seguir el plan de desarrollo ideado en la fase de producción. Sin embargo siempre suelen surgir imprevistos que conllevan a la adición, supresión o modificación de alguna de las características o recursos del juego. Durante esta fase, por lo tanto, es imprescindible realizar un plan de implementación que seguramente vaya evolucionando según se modifiquen los requisitos del diseño. Debido a los numerosos cambios que se pueden llegar a producir hay estudios que adoptan otras técnicas alejadas de desarrollos más convencionales (Sección 2.3). Aun así siempre se debe controlar cómo progresa el desarrollo, ya que si no hay un plan establecido y unas metas a ir alcanzando es muy fácil que el desarrollo pase a estar rápidamente fuera de control.

Igual de importante que tener un plan y controlar su progreso es saber cuando ha terminado el desarrollo, ya sea de una parte del proyecto o del juego en sí. Es muy difícil determinar cuando ciertas partes de un juego están terminadas ya que siempre se puede seguir depurando el resultado, mejorando o solucionando pequeños detalles. Es por tanto importante marcar unos claros criterios de terminación que sean fácilmente entendidos por las personas involucradas.

2.2.3. Post-producción

Una vez que el equipo de desarrollo considere que el juego está terminado y cerrado se pasa a la fase de post-producción, donde se lleva a cabo el control de calidad. En esta fase del desarrollo el equipo debe asegurarse que el producto quede libre de errores (*bugs*) y de conseguir que el juego sea robusto sin fallos. También será necesario comprobar temas de localización si el juego está previsto para varios idiomas y, si por ejemplo, el juego es para PC o dispositivos móviles, se deberán hacer pruebas con la mayor cantidad de dispositivos posibles para asegurarse que no hay fallos debidos a los diferentes elementos software y hardware que puedan existir.

El testeo sin embargo no solo trata de solucionar temas de funcionalidad sino también de jugabilidad. Se debe comprobar si la progresión del juego es adecuada (si la dificultad crece adecuadamente), si la usabilidad está clara y es intuitiva y si el juego está equilibrado en cuanto a las estrategias que puede usar el jugador.

El trabajo de detección de fallos debería ser idealmente realizado por gente externa a la producción y deberían reportar los fallos a dicho equipo de producción para que los solventase. Una vez que este equipo de control de calidad considera que los problemas más importantes han sido solventados y que el producto es comercializable se realiza el proceso de lanzamiento del juego. Dicho proceso varía mucho en función de la plataforma ya que si el proyecto se realiza para consola o ciertos dispositivos móviles suele tener que ser previamente aprobado por el fabricante del dispositivo, mientras que en otras plataformas no es necesario este tipo de aprobación.

Finalmente es altamente recomendable que una vez que el juego esté lanzado el equipo dedique cierto tiempo a analizar las cosas que fueron bien y mal en el desarrollo, para poder aplicar dicha experiencia en proyectos futuros. Del mismo modo se debe archivar una copia que contenga no solo el producto final lanzado sino también todo el material producido durante el desarrollo: código, recursos gráficos, sonoros,

documentación, herramientas auxiliares, etc.

2.2.4. Mantenimiento

Cada vez es más habitual que el desarrollo añada una fase extra de mantenimiento que da soporte al juego tras el desarrollo. Por mucho esfuerzo que se ponga en la post-producción del juego testeando en juego y arreglando *bugs*, en el momento que el juego se comercializa es muy normal que se descubran nuevos errores que haya que solucionar o que los desarrolladores quieran introducir algún tipo de contenido extra o mejora. Durante esta fase por tanto un pequeño equipo de desarrollo se dedica a publicar parches con correcciones o mejoras del juego.

Además, actualmente cada vez son más los juegos que necesitan soporte de red, ya sea porque el juego es directamente multijugador a través de internet o porque tiene algún tipo de soporte donde los jugadores publican sus logros o los comparten con sus amigos. Toda esta gestión de la información transmitida por la red necesita de un buen soporte tras el lanzamiento ya que hasta ese momento no se tiene una prueba real con un número grande de usuarios accediendo a dichos recursos.

2.3. Metodologías Ágiles: Scrum

Uno de los grandes problemas de la creación de videojuegos es que, debido a que el equipo de desarrollo es multidisciplinar y a la necesidad de generar funcionalidad jugable lo antes posible, es muy difícil planificar correctamente. Otra corriente defiende también que la creación de videojuegos es un tipo de arte y, como tal, es prácticamente imposible predecir cómo de buena va a ser una obra hasta que está materializada (Salen y Zimmerman, 2003). Es similar a juzgar un cuadro antes de que el pintor lo pinte, solo a partir de una descripción textual de qué es lo que el artista quiere pintar o judgar una película a partir del storyboard realizado por el director o guionista. Por tanto, debido al comportamiento emergente del desarrollo de videojuegos (Dormans, 2012), cada vez menos estudios siguen un desarrollo clásico en cascada, donde el equipo de desarrollo crea varios componentes de juego independientes que se unen al final del desarrollo confiando en que todo irá según lo esperado.

Hoy en día, el desarrollo software en general está evolucionando, de manera que el proceso de creación de un producto es cada vez más iterativa sobre todo en la etapa de pre-producción, aunque también durante la producción. El este proceso cíclico se conciben nuevas ideas o mecánicas, se documentan, se implementan, se evalúan a ver si cum-

plen los propósitos esperados y en función de eso se descartan, aceptan o se refinan dichas ideas volviendo a empezar el ciclo. Es por eso que, desde hace tiempo, cada vez más desarrolladores de software en general, y de videojuegos en particular, están adoptando metodologías ágiles como *scrum* (Schwaber y Beedle, 2001; Keith, 2010) o *extreme programming* (Beck, 1999; Beck y Andres, 2004) que están enfocadas en realizar un producto software mediante iteraciones.

Las metodologías ágiles pretenden que se genere nuevo software funcional iterativamente cada poco tiempo, donde las iteraciones pueden ir desde una semana a un par de meses, de manera que se trata de maximizar la simplicidad del software realizado, implementando solo lo que se necesita. Esto incentiva a que se mantenga un ritmo de desarrollo constante, evitando picos con grandes cargas de trabajo, y que sea más sencillo medir cuál es el progreso del desarrollo, ya que el director va viendo nuevo software funcional constantemente. Este tipo de metodologías son muy flexibles a los cambios en el diseño, que lamentablemente son necesarios en un desarrollo, y permiten un diseño incremental de mejora constante gracias a que se debe usar tecnología flexible como la arquitectura basada en componentes (Seción 2.4.2). También promueve el contacto de los miembros del grupo en reuniones donde se decide cuál debe ser el rumbo a seguir y animando a la programación en parejas ya que evita distracciones, facilita el brainstorming y clarifica ideas.

La planificación se hace en base a *historias* que definen una necesidad funcional para el juego. Estas historias se reparten entre los integrantes del equipo de desarrollo y, para saber qué carga de trabajo conllevará realizar una de dichas historias, el equipo completo llega a un acuerdo estimando cuál será el coste en horas. Una de las bases de las metodologías ágiles es que todo el mundo puede mejorar cualquier parte del desarrollo cuando lo necesite, y para esto se debe obligar a que haya integración continua de contenidos, donde el trabajo de uno se debe integrar varias veces al día, evitando así grandes conflictos que pueden surgir cuando se trata de juntar dos versiones desarrolladas por separado. Todo aquel que integre su trabajo debe asegurarse de que no hay errores para no entorpecer o frenar el trabajo del resto del grupo.

En el caso concreto del *Scrum*, en la planificación del proyecto, las historias se agrupan en iteraciones (sprints) que a su vez se agrupan en hitos más grandes (*releases*). De esta manera se hacen planificaciones continuas, siempre en base a funcionalidad que tenga valor para el usuario, a más corto y más largo plazo, donde las historias más prioritarias están definidas con más detalle. Las historias se encuentran almacenadas por prioridad en una cola de historias pendientes y, en la planifica-

ción del sprint actual, se hace una planificación más a corto plazo aun, dividiendo las historias que se vayan a realizar en ese sprint en tareas más concretas. Esas tareas se asignan luego a alguno de los miembros del equipo y se estima el coste de realizarlas en función de las horas que se prevé que se van a invertir. En la metodología *Scrum* se valora mucho el trato cara a cara en detrimento de la extensiva documentación. Se pauta que el equipo se reúna al final de cada sprint para valorar el trabajo realizado en el sprint actual y para decidir entre todos cuales son los objetivos que se deben cumplir al final de siguiente sprint. Además, se promueve que todos los días haya una pequeña reunión donde se comentan los problemas y necesidades más inmediatas del trabajo de cada uno. De esta manera todo el mundo conoce cuál es el estado del proyecto y en qué área está trabajando cada miembro por si hay necesidades especiales. Así mismo, esa comunicación promueve la retroalimentación del resto del equipo por lo que es bastante complicado que alguien haga algo que no se hubiese pautado (salvo que se replanifique entre todos), ya que todo el mundo está involucrado en el trabajo del otro y, en estas reuniones, saldría a la luz.

2.4. Arquitectura software de videojuegos

2.4.1. Arquitectura general

Los videojuegos son aplicaciones de tiempo real grandes y complejas que usan una gran cantidad de recursos multimedia, deben proporcionar una experiencia inmersiva y, lo más importante, deben de ser divertidos. Como se ha comentado previamente, esa búsqueda de la diversión es un proceso iterativo de prueba y error que va desde las primeras etapas de prototipado a las últimas en las que se balancean las mecánicas de juego. Desde el punto de vista de la ingeniería del software esta situación es muy poco deseable ya que las especificaciones van evolucionando hasta el día en que se entrega el producto final (Swift et al., 2008).

Aunque hay muchos tipos y complejas descripciones de arquitecturas de videojuego, en su forma más simple podemos considerar dos grandes bloques:

- *Front-end*, responsable de retroalimentar la experiencia del jugador procesando los comandos de entrada del usuario (teclado, ratón, etc.) y presentando el juego (mediante motores gráfico, de sonido, etc.)

- *Back-end*, encargado de la lógica de juego que especifica las re-

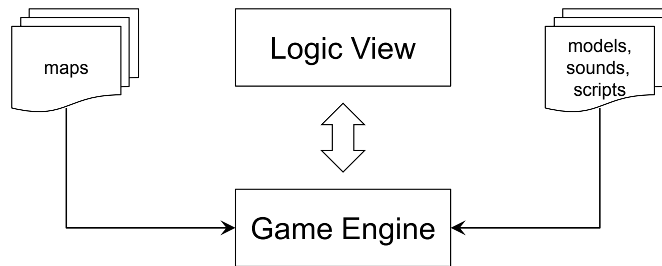

Figura 2.1: Vista de una arquitectura de videojuegos en su forma más simple

glas, determina la dinámica de interacción con los objetos, proporciona comportamientos a los personajes no jugadores (NPC), orquesta las interacciones entre ellos, etc.

La Figura 2.1 muestra un esquema simplificado de la arquitectura de videojuegos. El módulo que hace referencia al *motor de juego* viene a ser el *front-end* y, desde el punto de vista arquitectónico, es el encargado de cargar los recursos externos, procesar la entrada de usuario y presentarle los resultados. Por otro lado, la *vista lógica* del juego, el *back-end*, es el módulo encargado de *personalizar* el juego.

El funcionamiento de un videojuego puede reducirse a un bucle principal donde en cada iteración del bucle se hacen tres acciones:

1. El motor de juego recoge la entrada del usuario y le comunica a la vista lógica los comandos que ha decidido hacer el jugador.

2. La lógica de juego, en función de los comandos de entrada del usuario, del estado actual del juego y de las acciones realizadas por los NPCs, toma decisiones acerca de qué es lo que se debe presentar al usuario, comunicándoselo al motor del juego.

3. El motor de juego debe transmitir el nuevo estado del juego al usuario haciendo uso de recursos gráficos y sonoros.

Mismamente en el *Pong*, el cual podríamos clasificar como el primer videojuego comercial, ya tenía este bucle donde el usuario se comunicaba con un potenciómetro, el sistema iba actualizándose calculando la nueva posición de la pelota y de las dos palas y, finalmente, presentaba por el televisor los resultados.

Los componentes que conforman el *front-end* suelen estar especificados desde las primeras fases del desarrollo incluso, en la mayoría de los casos, se hace uso de middleware externo. La parte del *back-end*,

la lógica de juego, es la que controla el flujo del juego y la que toma las decisiones de qué es lo que se deba hacer en cada instante, comunicándose con el motor de juego. La lógica de juego mantiene siempre un estado del juego y, prácticamente siempre, este estado suele gestionarse mediante el uso de de un conjunto de entidades u objetos de juego, que representan todo tipo de elementos del juego.

2.4.2. Arquitecturas software para las entidades de juego

Las entidades de juego son piezas de lógica autocontenidas (Bilas, 2002), o piezas de contenido interactivo (Rabin, 2010), que pueden llevar a cabo diferentes tareas como renderizarse a si mismas, encontrar o seguir caminos, tomar decisiones, etc. Ejemplos obvios son los NPCs, jugadores objetos interactivos o mobiliario pero también son entidades elementos no tan obvios como secuencias de cámara, puntos de ruta o disparadores que controlan la historia del juego.

Además de las entidades en si es necesario un sistema de gestión de éstas para saber cuáles están activas, cuáles se deben crear o destruir, etc. Este módulo forma parte del núcleo de un juego y, debido a su tamaño y complejidad, requiere de esfuerzo en el desarrollo. Por poner algún ejemplo, el conocido juego Half-Life® que data de 1999 tiene más de 65,000 líneas de código en este módulo (sin contabilizar líneas vacías ni comentarios), mientras que el Far Cry® de 2004 excede las 95,000 líneas de C/C++[1] incluso teniendo en cuenta que la mayoría del módulo estaba escrito en LUA. Dos años más tarde, en 2006, Gears of War® tenía 250,000 líneas de C++ y scripts (Sweeney, 2006).

Debido a el tamaño, la complejidad y a su naturaleza de especificación cambiante, se hace necesaria una arquitectura altamente flexible y extensible. Al principio de los años 2000, cuando los lenguajes orientados a objetos se fueron imponiendo en el desarrollo de videojuegos, Las entidades de juego se organizaban en jerarquías basadas en herencia donde, por ejemplo, *jugadores* y *enemigos* son *personajes* y tanto los *personajes* como los *objetos* son *entidades*. Este tipo de estructura se demostró rápidamente que era rígida y difícil de mantener como se comenta en la Sección 2.4.2.1.

Aún dentro del paradigma de orientación a objetos pero usando composición dinámica en lugar de estáticas jerarquías de clases, la arquitectura basada en componentes es la tecnología que se usa hoy en día para la implementación del módulo encargado de las entidades de juego. Un *componente* en esta arquitectura representa una pequeña porción

[1] ambas cuentas fueron llevadas a cabo por David A. Wheeler usando SLOCCount

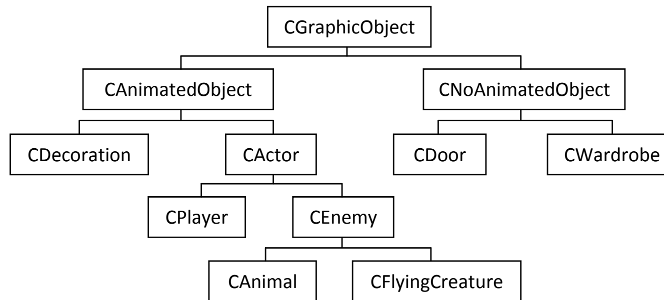

Figura 2.2: Árbol de herencia tradicional

de funcionalidad idealmente autocontenida que puede ser agregada o eliminada de una entidad de juego, siendo de alguna manera una combinación de los patrones de diseño Command y Decorator. Por tanto, un nuevo tipo de entidad es simplemente una combinación de componentes que pueden ser creados en tiempo de ejecución, lo que aporta flexibilidad y potencia la reutilización (Sección 2.4.2.2)

2.4.2.1. Jerarquías de clases

Durante el diseño de un juego es habitual categorizar las entidades mediante el uso de una ontología que modela la relación *es-un* entre todas ellas. Este conocimiento puede ser usado en la fase de desarrollo mediante herencia en el código del juego por lo que no sorprende que la gestión de entidades tradicional se basase en herencia dode las entidades se implementan como subclases que derivan de la misma clase base. En función de la complejidad del juego, las jerarquías pueden ser bastante profundas, con clases abstractas intermedias que agregan funcionalidad común para incrementar la reusabilidad del código. Como ejemplo, la Figura 2.2 muestra una jerarquía parcial hipotética, donde los nodos hoja representan clases concretas mientras que los nodos internos son clases que pueden no haber sido concebidas durante la fase de diseño pero creadas para la reutilización de código.

Esta jerarquía trae varias decisiones implícitas debido a la manera en que se parten las clases cuando descendemos por la jerarquía. Estas decisiones pueden volverse problemáticas a la larga, sobre todo en un entorno tan cambiante como el desarrollo de videojuegos. Supongamos que a la jerarquía de la Figura 2.2 quisiésemos añadir una clase *CHumanEnemy*, una entidad con las mismas habilidades que el jugador pero controlada por una inteligencia artificial. La nueva entidad tendría por tanto características de *CEnemy* y de *CPlayer* por lo que la solución que parece obvia sería que heredase de ambas. Sin embargo, en

primer lugar, la herencia múltiple no es soportada por todos los los lenguajes orientados a objetos y, en segundo, aunque fuese soportada provocaría *herencia en diamante*, donde *CHumanEnemy* hereda dos veces de *CActor*, lo cual genera ambigüedad, confusión y una complejidad no deseable.

La solución por tanto sería usar herencia simple, de manera que *CHumanEntity* sería subclase de, por ejemplo, *CEnemy*. Para evitar la duplicación de características como *conducir vehículos* en *CPlayer* y *CHumanEntity*, éstas pasan a la superclase común (*CActor*), lo que implica que *todas* las subclases hereden dichas características (incluyendo *CAnimal* y *CFlyingCreature*).

El efecto que esto provoca es superclases repletas de funcionalidad y subclases que se encargan de activar y desactivar funcionalidades heredadas, lo que constituye un diseño orientado a objetos bastante pobre. Por mencionar 2 ejemplos de arquitecturas clásicas reales, la clase base de Half-Life 1 tiene 87 métodos y 20 variables públicas, mientras que la clase base de Sims 1 tiene más de 100 métodos.

Algunas de las consecuencias de las superclases gigantes son el incremento del tiempo de compilación (Lakos, 1996), código difícil de entender y el conocido *fragile base class problem* (Szyperski, 1997), el cual provoca que cambios aparentemente pequeños en la clase base tiene un impacto negativo en las subclases.

Todos estos problemas no solo surgen en el desarrollo de videojuegos, pero en este campo se acentúan debido a los constantes cambios en la especificación, lo que fuerza a los programadores a revisar y extender una arquitectura bastante estática como es la jerarquía de clases.

2.4.2.2. Arquitectura basada en componentes

El mayor problema de la arquitectura anterior es su naturaleza estática, ya que los continuos cambios de diseño provocan que la herencia no sea la mejor solución. Por esta razón los desarrolladores hoy en día tienden a usar sistemas basados en componentes (West, 2006; Rene, 2005; Buchanan, 2005; Garcés, 2006). En lugar de tener entidades de una clase concreta que defina su comportamiento, ahora cada entidad es simplemente un contenedor donde cada funcionalidad o habilidad que tenga la entidad es implementada por un componente. Desde el punto de vista del desarrollador, cada componente hereda de una clase o interfaz *IComponent*, mientras que la entidad es simplemente una lista de *IComponent*s.

Como los componentes son objetos genéricos con una interfaz común

independiente de su funcionalidad, no es sencillo invocar un método de otro componente para activar su funcionalidad. Por ejemplo un componente *AI* no es capaz de hacer una llamada directa a un método como *MoveTo* cuando se necesita. Una solución es tener un sistema de inferencia de tipos en tiempo de ejecución (RTTI de sus siglas en inglés run-time type information) para, sabiendo el tipo, poder invocar directamente a un método *MoveTo* del componente *Movement*. Este ejemplo se puede ver en la Figura 2.3.

```
void AI::Update() {
    ...
    IComponent *c;
    c = m_entity->getComponent(Movement);
    assert(c);

    MoveTo *mc = static_cast<Movement*>(c);
    mc->MoveTo(position);
    ...
}
```

Figura 2.3: Ejemplo de invocación usando RTTI

Sin embargo, aunque esta solución funciona, produce una dependencia entre componentes: el componente *AI* debe conocer en tiempo de compilación qué componente es responsable del movimiento de la entidad para poder acceder a la funcionalidad del componente.

Otra solución es llevar a cabo la comunicación entre componentes de una manera genérica, usando paso de mensajes. La información acerca de la invocación se encapsula en un pequeño objeto o estructura y es enviado a los componentes hermanos con la esperanza de que uno de ellos (o varios) acepte el mensaje y lleve a cabo la funcionalidad requerida. Esta estrategia combina el patrón Command (Gamma et al., 1995) combinado con el patrón Decorator ya que el mensaje se envía a través de todos los componentes.

la interfaz *IComponent* es por tanto vista como un puerto de comunicación que es capaz de recibir y procesar mensajes, donde un mensaje es una porción de información con un identificador y algunos parámetros opcionales. Cuando un componente (o un módulo externo) necesita comunicarse con otro envía un mensaje (mediante la entidad a la que pertenece) a sus componentes hermanos. La entidad propaga el mensaje entre sus componentes llamando a un método del componente (por ejemplo *handleMessage*); dependiendo de su funcionalidad, el mensaje se ignorará o procesará. En este escenario, las entidades juegan el rol

de difusor de mensajes. En nuestro ejemplo anterior, es el componente *AI* el que enviaría un mensaje a la entidad y el componente que implementase la habilidad de moverse interceptaría el mensaje, calcularía la ruta y cambiaría periódicamente la posición.

Otra cuestión que surge con esta arquitectura es dónde se deben almacenar los atributos de la entidad (como por ejemplo la posición o la vida). Una aproximación usual es situar todos los atributos que virtualmente tienen todos las entidades del juego (nombre, posición, orientación) en la misma entidad mientras que el resto se colocarían en los componentes. De esta manera el componente gráfico sería el encargado de almacenar el modelo a renderizar, el componente de salud en encargado de tener un atributo vida, etc. De esta manera cada componente puede acceder tanto a sus atributos como a los atributos comunes que se encuentran en la entidad pero, si se quiere acceder a atributos de otro componente, habría que usar nuevamente RTTI o enviar mensajes con petición de valores de atributos y esperar a recibir mensajes de respuesta.

Una posible solución, cuando se quiere que los diferentes componentes puedan compartir atributos, es extender la funcionalidad de la entidad con una lista de pares de clave-valor. Durante la inicialización de la entidad los componentes registran los atributos en la entidad, de manera que se hacen públicos también para otros componentes. El acceso se realiza invocando un método de la entidad que, opcionalmente, puede informar a otros componentes cuando se modifica el valor almacenado.

Mediante el uso de estas técnicas cambiamos una jerarquía de clases estática creada por mecanismos de herencia por un diseño versátil que nos permite crear nuevas entidades de manera sencilla por combinación de componentes. Como ahora las entidades son meras listas de componentes, los componentes concretos que conforman una entidad no tienen por qué estar descritos en el código fuente sino que pueden especificarse en un fichero externo que se procesa en tiempo de ejecución, haciendo que la creación de las entidades esté dirigida por datos. Este fichero, habitualmente conocido como *blueprints*, se interpreta al inicio de la ejecución del juego y, cuando el juego carga un nuevo nivel de un fichero de mapa, se usan esas descripciones de entidades para crear dinámicamente todas las entidades descritas en nivel. Bien como parte del fichero *blueprints* o bien en otro fichero conocido como *archetypes*, es habitual acompañar la descripción por componentes de las entidades por una serie de valores por defecto que se le dan a los diferentes atributos.

La creación de las entidades se realiza en dos pasos. En primer lu-

gar se añaden los componentes descritos en el *blueprints* a la entidad y posteriormente se inicializa la entidad con el conjunto de atributos. De alguna manera esta aproximación simplifica la creación de nuevos tipos de entidades, ya que no se requiere ninguna tarea de programación sino simplemente seleccionar las diferentes habilidades requeridas para la entidad de un conjunto de componentes. Por ejemplo, la Figura 2.4 describe una entidad *Soldier* mediante sus componentes en el fichero *blueprint* y unos pares atributo-valores por defecto (*archetypes*) y luego se instancia en un mapa mediante la adición de más valores de atributos que pueden sobrescribir esos valores por defecto.

Motores de juego más sofisticados suelen ser más versátiles aún, permitiendo el uso de lenguajes de scripts para la creación de componentes. Por poner un par de ejemplos, CryEngine®[2] permite el uso de LUA para la creación de scripts mientras que Unity®[3] permite el uso de C#, UnityScript y Boo. Centrándonos ya en juegos comerciales, Dungeon Siege® tenía 168 componentes pero solo 21 de ellos estaban escritos en C++. El uso de este tipo de motores permite a los desarrolladores crear componentes de una manera más rápida y sin necesidad de usar el compilador (Bilas, 2002).

Finalmente cabe resaltar que prácticamente todos los juegos modernos se crean con una arquitectura basada en componentes (Chady, 2009), la cual ha pasado a ser la principal solución frente a los sistemas basados en herencia. Un juego como [Prototype]® tenía 145 componentes diferentes y ninguno de ellos aparecía en todas las entidades. En una charla dada por Marcin Chady (Chady, 2009) comentó que el componente más usado fue aquél responsable de renderizar la entidad, seguido por el componente físico. También reconstruyó una jerarquía de clases, usando herencia múltiple, que implementaba el mismo comportamiento que el conjunto de componentes. El resultado fue una jerarquía estática difícil de mantener con más de 160 clases frente a la arquitectura de componentes, la cual era de un mantenimiento más sencillo y más fácilmente extensible. De acuerdo a la experiencia relatada por Mick West (West, 2006), la mejor decisión que puede tomar un equipo de desarrollo, cuando se tiene un escenario como este, es refactorizar todo su código y basarlo en componentes, tal y como él hizo con la saga Tony Hawks®.

[2]`http://mycryengine.com/`
[3]`http://unity3d.com/`

```
<blueprints>
  <entity type="Soldier">
    <component type="AnimatedGraphics"/>
    <component type="Physics"/>
    <component type="AI"/>
    <component type="Move-To"/>
    <component type="Shoot-To"/>
    <component type="Health"/>
  </entity>
   ...
</blueprints>

 ...

<archetypes>
  <entity type="Soldier">
    <attribute key="model" value="soldier.mesh"/>
    <attribute key="script" value="randomMovement.lua"/>
    <attribute key="speed" value="0.5f"/>
    <attribute key="shoot" value="0.8f"/>
    <attribute key="health" value="200"/>
  </entity>
   ...
</archetypes>

 ...

<map name="Island">
  <entity type="Soldier">
    <attribute key="name" value="my_turtle"/>
    <attribute key="position" value="32,0,8"/>
  </entity>
   ...
</map>
```

Figura 2.4: Ejemplo de la descripción de una entidad *Soldier*

Capítulo 3

Estado del Arte

El desarrollo de videojuegos es un proceso llevado a cabo por un equipo multidisciplinar donde personas de índole muy diversa deben ponerse de acuerdo. Con los años los productos son cada vez más ambiciosos lo que provoca que su desarrollo sea una tarea complicada. Debido a esto, la tecnología, técnicas y metodologías adoptadas han ido y van evolucionando con el tiempo, tratando de minimizar los problemas, agilizar el desarrollo y conseguir el mejor producto posible en el menor tiempo. En este capítulo se recogen algunos de los problemas y cuellos de botella que han sido detectados por literatura especializada en desarrollo de videojuegos, exponiendo también soluciones adoptadas por otros autores a ciertos problemas.

3.1. Documento de diseño como herramienta de comunicación entre diseñadores y programadores

El propósito del *documento de diseño* es expresar la visión del juego, describir sus contenidos y presentar un plan para la implementación. Este documento es la “biblia” donde el productor sugiere los objetivos, los diseñadores luchan por sus ideas y de donde los programadores y artistas obtienen las instrucciones para expresar su pericia (Ryan, 1999). Para la escritura de este *documento de diseño*, los diseñadores tienen que traducir a palabras las ideas que tienen acerca de la interacción del jugador con el entorno, cómo esas interacciones están restringidas por reglas y cómo las mecánicas de juego pueden ayudar a superar los desafíos propuestos por el juego de una manera interesante (Sicart, 2008).

Aunque prácticamente la totalidad de los desarrollos de videoju-

gos están guiados por un *documento de diseño* no hay ningún modelo estandarizado y cada compañía sigue sus propios modelos basados en la experiencia. Como se ha comentado en la Sección 2.2.1, Katharine (2012) comenta que los *documentos de diseño* suelen amplias y abundantes descripciones pormenorizadas de las diferentes facetas del juego en lenguaje natural. Debido al formato y la extensión, el *documento de diseño* es raramente usado de manera íntegra por los desarrolladores en futuras etapas del desarrollo, sirviendo sólo como una especie de contrato o acuerdo, entre programadores y diseñadores, del enfoque que se le va a dar al desarrollo (Dormans, 2012; Keith, 2010).

Debido a que no es posible anticipar totalmente la jugabilidad y predecir la experiencia que trasmitirá un juego (Salen y Zimmerman, 2003), es habitual que haya cambios en el diseño según avanza el desarrollo y mantener actualizado un documento de diseño clásico se vuele poco productivo (Dormans, 2012). Por ello, ciertos estudios (Long, 2003) aseveran que realizar un *documento de diseño* extremadamente detallado y de gran tamaño durante la pre-producción es una pérdida de tiempo y recomiendan que solo se haga un diseño suficiente para poder realizar una planificación general. Como ya se hizo referencia en la Sección 2.2.1, otros incluso van más allá y prefieren documentos de diseño minimalistas con presentaciones más visuales en una sola página o póster (Librande, 2010). Esto se debe a que un *documento de diseño* clásico es de naturaleza estática, mientras que la naturaleza del desarrollo de videojuegos es cambiante, como ya hemos comentado. Esto suele llevar a que se produzcan cambios drásticos en el diseño del juego incluso en fases avanzadas del desarrollo (Salen y Zimmerman, 2003).

la idoneidad de las mecánicas de juego no puede ser inferida a partir de las descripciones de las reglas de juego o del *documento de diseño* lo que provoca que la jugabilidad del juego no pueda ser predecida ni siquiera por diseñadores expertos. Esto implica que haya que construir prototipos para probar las ideas (Sección 2.2.1), por lo que los diseñadores pierden en cierta manera el control sobre su construcción y deben trabajar codo con codo con los programadores y artistas. Que el medio por el cual los diseñadores trasmiten sus ideas sea básicamente textual ralentiza la trasmisión de la información, facilita una errónea interpretación y complica la modificación y revisión de ideas ya probadas. Ciertos autores hacen comparaciones con otros campos y dicen que construir un prototipo o videojuego a partir del documento de diseño es como si un arquitecto diseñase un edificio usando lenguaje natural (Katharine, 2012) o como si se tratase de crear una película a partir de los comentarios del director (Sheffield, 2007).

Hay una gran disyuntiva acerca de hasta qué punto se deben definir las características del videojuego que se quieren implementar ya que es posible que una idea o mecánica de juego sea descartada al ser prototipada y que esto provoque un replanteamiento de otros aspectos del diseño, provocando pérdida de trabajo realizado. A su vez, aunque una descripción extensamente detallada puede ser contraproducente debido a la reticencia de los programadores a leer documentos extensos, una descripción extremadamente simplista puede llevar a que los programadores hagan libre interpretación de detalles poco definidos. Esto resulta un gran problema ya que es bastante normal que el equipo de producción implemente alguna característica de manera que no coincida con lo que el diseñador tenía en mente, lo que provoca una brecha entre la concepción de las ideas de juego y la implementación final de éstas (Katharine, 2012).Cuando esto sucede provoca que ciertas características deban pasar nuevamente por el equipo de producción, generando grandes retrasos y roces entre los diferentes equipos. Por tanto, parece necesario que los diseños deban ser realizados por los diseñadores en conjunción con el equipo de programación ya que serán los encargados de implementar el sistema (Long, 2003).

El lenguaje natural es fácil de leer, pero no es sencillo revisar una especificación de gran tamaño en lenguaje natural. Por esa razón en muchas disciplinas se buscan representaciones alternativas, como gráficas o diagramas, que transmita visualmente la misma información pero de una manera más sencilla (Araújo y Roque, 2009). En el campo del diseño de videojuegos es necesario tener algún tipo de elemento que sirva para transmitir las ideas de los diseñadores a los programadores pero, sin embargo, un *documento de diseño* tradicional es generalmente considerado como difícil de manejar e ineficiente (Dormans, 2012). Por eso hay bastante esfuerzo realizado en conseguir representaciones alternativas que sustituyan o complementen al *documento de diseño* tratando de representar formalmente el dominio del juego (Sección 3.2) y presentando gráficamente esos dominios a los diseñadores y programadores (Sección 3.4).

3.2. Representación formal del dominio de juego

Los problemas de comunicación existentes entre diseñadores y programadores, y la dificultad que supone transmitir de manera sencilla las ideas que tienen los primeros a los segundos, provoca que haya ciertos esfuerzos tanto en la industria como en el ámbito académico para aliviar este proceso. Varios diseñadores reclaman al menos un lenguaje o mé-

todo unificado de diseño (Zagal et al., 2005; Zagal y Bruckman, 2008; Silvano-Orita-Almeida y Soares-Correa-da Silva, 2013) que permita describir los juegos existentes y pensar en el diseño de otros nuevos, de manera que sea más sencillo describir un juego sin dar lugar a dobles interpretaciones, para que así sea más fácilmente implementable.

Una de las aproximaciones en la que creen varios diseñadores es que hace falta una gramática o notación gráfica para modelar el diseño del juego (Katharine, 2012). Para este propósito el problema radica en identificar las piezas fundamentales de dicha gramática, para lo cual se deberían trocear los juegos de manera analítica en sus unidades fundamentales. En este sentido varios diseñadores señalan como elementos centrales todo lo relacionado con las interacciones del jugador, que estarían en el corazón de la interactividad. Como ejemplo, Salen y Zimmerman (2003) describe esas unidades fundamentales como las posibles decisiones que puede tomar el jugador en cada momento y las consecuencias que se obtienen de éstas. Por otro lado, el diseñador Dan Cook presenta su idea de los *skill atoms* que describen, mediante el uso de los ingredientes básicos de los videojuegos, cómo un jugador adquiere o percibe una habilidad del juego. Más concretamente, cada uno de estos *skill atoms* es un bucle de retroalimentación compuesto por cuatro elementos centrales que son acción, simulación, *feedback* y modelado que permiten describir una habilidad que el jugador adquiere mediante la repetición de dicho proceso (Cook, 2007).

Hay otros proyectos más ambiciosos que van más allá de definir sólo las interacciones y experimentan las ventajas de substituir o complementar el *documento de diseño* del videojuego con un diseño formal o declarativo del domino. Se usa para mejorar la transmisión de ideas o para automatizar ciertos procesos, aunque desgraciadamente su uso en la industria es bastante limitado por no decir prácticamente nulo. Usando una definición formal de los elementos del juego y sus relaciones, los diseñadores traducirían sus ideas de una manera más sencilla a la tradicional. Sicart (2008) propone usar el paradigma de la programación orientada a objetos para la representación de las mecánicas, que se representan como métodos que son invocados por los agentes para interactuar con el estado de juego.

Las reglas de juego son al final un conjunto de condiciones que sirven para transitar de estado, afectando a la aplicabilidad de las mecánicas de juego. En el marco de orientación a objetos, Sicart (2008) representa las mecánicas como propiedades particulares o generales del sistema y de los diferentes agentes. Dentro de este paradigma los desarrolladores deben describir los desafíos y los objetivos del juego, además de

describir las diferentes mecánicas de juego que son necesarias para superar un desafío o alcanzar un objetivo. Por otro lado, Araújo y Roque (2009) propone el uso de *redes de Petri* como lenguaje de modelado. Las *redes de Petri* pueden ser descritas como un conjunto de ecuaciones algebraicas por lo que debido a su expresividad han sido usadas en diferentes áreas. En este caso concreto de modelado de videojuegos, el uso de *redes de Petri* permite la descripción de interacciones, procesos o secuencias de acciones, que determinan una acción de más alto nivel, y toma de decisiones de los personajes. Otra alternativa propuesta, que no se aleja de la anterior, es realizar la descripción del modelo formal de juego mediante *Answer Set Programming* (Lifschitz, 2008), de manera que con él se definen una serie de reglas mediante el uso de programación lógica. Una de las ventajas es que dicho conocimiento puede luego usarse para generación procedimental de contenido (Smith y Mateas, 2011; Smith et al., 2012).

Las ideas anteriores dan una representación visual (o no visual) a la jugabilidad representando gráfica o lógicamente las interacciones entre los diferentes elementos, las reglas de juego, secuencias de acciones o comportamientos de los personajes. Sin embargo ponen poco o ningún esfuerzo en la descripción de los personajes y de los diferentes elementos de juego. Centrándonos en ese otro aspecto, la descripción de los elementos de juego, hay diferentes aproximaciones donde una de ellas es el uso de objetos inteligentes o *smart objects* (Peters et al., 2003) para describir la jugabilidad del juego mediante los objetos de éste. Esta técnica funciona bastante bien en juegos donde haya mucha interacción del usuario o de los personajes con los diferentes objetos del juego o donde la jugabilidad esté basada en el movimiento o la interacción directa del usuario como por ejemplo los juegos que usan el *Wiimote* de Nintendo, el *Kinect* de Microsoft o el *Move* de Sony (Tutenel, 2012). Generalmente este tipo de modelado se hace enfocado a los objetos de juego, que son los que definen las reglas y cómo deben actuar los diferentes personajes no jugadores (NPCs) cuando los usan. El objeto inteligente toma el control del NPC indicando dónde debe colocarse, dónde debe mirar, que acciones debe realizar, etc. En la saga *The Sims* se hace uso de este tipo de técnica de manera que los objetos contienen información del modelo, definiciones de interacciones, listas de animaciones, meta-datos con información, scripts ejecutables, etc. Tener la información centralizada es muy útil tanto durante el diseño del juego como durante la ejecución del mismo ya que hace que el juego sea mucho más modular, lo que explica la facilidad de *Electronic Arts* para sacar expansiones de su juego.

Enfatizando en la idea de la centralización de la información, una téc-

nica que destaca por su integración es el uso de información semántica asociada a los modelos tridimensionales (Tutenel et al., 2011; Zhang et al., 2012; Youngblood et al., 2011; Gaildrat, 2009). De esta manera los propios modelos creados para el juego vienen con una descripción adjunta que les autodescribe en función de su forma, funcionalidad y características. Mediante este tipo de anotaciones adjuntas al modelo 3D, la creación del dominio de juego podría realizarse de una manera declarativa a través de las entidades que lo forman. Con estas descripciones el trabajo de los diseñadores es más sencillo ya que toda esa información facilita un correcto diseño manual de niveles de juego. Además se puede aprovechar esa información de los modelos para detectar inconsistencias o, incluso, para generar niveles de manera procedimental. Una de las ventajas de este tipo de anotaciones es que si los recursos se anotan con un lenguaje común se pueden reutilizar en diferentes tipos de juego. Sin embargo, a la hora de la verdad, serían también necesarias anotaciones específicas, que tengan que ver con características concretas del juego que se desea crear. Por otro lado también habría que añadir descripciones de entidades que no tienen representación gráfica.

Siguiendo con la idea de poder describir el juego de manera que la información no se encuentre escondida dentro del código, y tratando de agruparla en un modelo que permita obtener valor añadido, en la investigación dentro del ámbito académico se han hecho algunas aproximaciones para el uso de ontologías. El uso de descripciones ontológicas como medio para el desarrollo de software es usado en otras áreas de la informática (Tetlow et al., 2006). Dentro del campo de desarrollo de videojuegos, el uso de ontologías puede usarse para definir juegos ya existentes y de esta manera tener un lenguaje base que nos permita definir nuevos juegos a partir de los elementos ontológicos que simplifiquen su comprensión por otros miembros del equipo (Zagal et al., 2005; Zagal y Bruckman, 2008). Sin embargo, el uso de ontologías puede usarse no solo para definir y categorizar juegos ya existentes en base a ciertos conceptos, sino para el desarrollo de un juego concreto. Como ejemplo se pueden definir los elementos que se van a usar dentro de un videojuego y sus relaciones en una ontología y, posteriormente, usar directamente el contenido de la ontología en el videojuego propiamente dicho (Machado et al., 2009).

Sin embargo, un ejemplo más completo del uso que se puede dar a las ontologías dentro del proceso de desarrollo de videojuego viene dado por el trabajo de Gao et al. (2005). En dicho trabajo se introduce el uso de ontologías para la definición de agentes (o NPCs) que describan comportamientos y simplifiquen la interacción entre agentes o NPCs y seres

los humanos. También diferencian los diferentes roles en la creación del dominio, considerando por un lado a los diseñadores de los agentes y por otro los diseñadores de la aplicación, que diseñan el escenario. Los dominios que proponen para la descripción de interacciones se limitan al uso de 5 elementos que son: 1) *señales*, necesarias para que los agentes perciban el entorno, 2) *acciones*, que pueden realizar los agentes, 3) *escenarios*, que son descripciones de acciones que se desencadenan ante la detección de un conjunto de señales, 4) *agentes*, que se definen por diferentes escenarios y 5) *avatars* que son controlados por humanos. Por otro lado, las definiciones de las diferentes *entidades* de la escena (edificios, puentes, puertas, armas, puntos de cobertura, etc.) se definen mediante las relaciones *es-un* y *tiene-un*, el uso de *propiedades* como nombre, grado de cobertura, visibilidad, etc. e *interacciones* que son efectos generados cuando los agentes u otras entidades interaccionan con ellos (sonido, efectos de partículas, etc.). Además, las acciones también se encuentran descritas en la ontología de manera jerárquica de modo que los agentes pueden ejecutar acciones abstractas de alto nivel en lugar de simplemente poder listar acciones primitivas.

El uso de ontologías en el desarrollo de videojuegos ya no solo se encuentra en el ámbito académico sino que en un futuro cercano podremos encontrarnos propuestas como las de *XaitKnow*[1] que permitirá el modelado del conocimiento global del juego mediante el uso de ontologías.

Una representación formal tiene otras ventajas más allá de la centralización de la información y de la posibilidad de sacar un valor añadido de ella durante el desarrollo o la ejecución del juego. Una definición semántica rica en conocimiento puede proporcionarnos la posibilidad de mostrar la información en diferentes niveles de detalle, de manera que todo el conocimiento esté representado en una única estructura pero que se pueda consultar a diferentes niveles. Esto permite que diferentes roles puedan trabajar con el mismo conocimiento, unos desde un prisma más técnico y específico, con detalles de implementación, mientras otras trabajan a un nivel más conceptual y con una visión más global. Un ejemplo de como se pueden hacer anotaciones de más alto o bajo nivel (incluso alguna automática) está en el trabajo de Ibáñez y Delgado-Mata (2006); Ibáñez-Martínez y Mata (2006).

Al final, si toda la información acaba descrita en lenguaje natural en el enorme *documento de diseño*, un montón de trabajo habrá sido en cierta manera desperdiciado ya que no puede reaprovecharse y deberá replicarse manualmente todo el código necesario para realizar las

[1] `http://www.xaitment.com/english/products/xaitknow/xaitknow.html`

funcionalidades descritas en dicho documento. Por otro lado, un buen diseño formal debería ayudar a evitar diferentes interpretaciones, cosa que con lenguaje natural puede resultar más complicado. No obstante, el uso de un diseño formal no debería porque implicar la desaparición del documento de diseño ya que hay cosas que siguen siendo necesarias describir textualmente, pero una declaración semántica del dominio puede complementar en gran medida la documentación.

3.3. Generación de contenido de juego

Una de las grandes ventajas de tener un dominio formal rico en conocimiento es que se puede usar para diferentes etapas del desarrollo de un videojuego. Por tanto no solo sirve como herramienta de comunicación entre los diseñadores y los programadores sino que la descripción semántica que tienen dichos dominios pueden ser usados para inferir información y para generar contenido de juego.

Youngblood et al. (2011) propone describir los modelos 3D en el propio fichero del modelo, etiquetando cada uno de ellos con anotaciones que definan *qué es*, *qué propiedades tiene* y *qué acciones se pueden realizar sobre él*. En este trabajo muestran cómo dichas anotaciones ayudan a la creación mundos interactivos ricos en conocimiento, valiéndose de las descripciones semánticas para dar soporte a herramientas de autoría donde los diseñadores trabajan con los elementos del entorno para mejorar las interacciones de los NPCs con otros personajes y con el propio entorno, abriendo incluso posibilidades a la reutilización de contenido y aprendizaje de los NPCs durante el juego.

Tutenel (2012) propone también generar descripciones ontológicas de diversos elementos de juego (básicamente muebles) que son usadas para generar escenarios de manera procedimental. Las descripciones se añaden directamente a uno o varios modelos 3D, de manera que a la hora de generar escenarios estas descripciones se tienen en cuenta para, en base a unas descripciones semánticas acerca de qué objetos son necesarios o plausibles en la escena, unas restricciones y unos filtros, poder generar distribuciones de interiores que sean consistentes semánticamente. La mayor diferencia respecto al trabajo de Youngblood et al. (2011) es que las escenas pueden ser generadas en un proceso totalmente automático o en un proceso asistido por el diseñador, que va colocando elementos en el escenario mediante el *feedback* recibido por el sistema. Además, las escenas siempre pueden ser modificadas en función de las necesidades del desarrollo, manteniendo siempre la consistencia después de los cambios.

Smelik (2011) parte y se apoya en ideas previas y conjuntas con Tim Tutenel (Tutenel et al., 2008, 2009b,a, 2010) para extender la generación de distribuciones de interiores, concretada finalmente en el trabajo de tesis Tutenel (2012), a generación procedimental de paisajes y ciudades. Para la descripción de los escenarios propone una descripción por capas en diferentes niveles de abstracción, donde uno de esos niveles es la definición semántica propuesta para la generación de distribuciones interiores. Según Smelik (2011), los métodos tradicionales de generación procedimental de contenido son complejos y poco intuitivos, ofreciendo poco control al usuario. Lo que propone en su trabajo es un modelado declarativo de mundos virtuales donde el diseñador se concentra en *qué es lo que quiere crear* en vez de en *cómo debe modelarlo*, permitiendo al diseñador un proceso interactivo que le ofrece control a varios niveles de granularidad, pero manteniendo siempre la consistencia de manera automática.

Otro tipo de generación procedimental es aquella que permite tanto la generación de contenido de juego como de código a partir de un modelo de juego bien definido. En este sentido hay mucho trabajo acerca de las arquitecturas dirigidas por modelo, que generalmente suelen basarse en modelos definidos en UML, pero durante los últimos años también se han introducido los desarrollos software dirigidos por ontologías (Tetlow et al., 2006; Deline, 2006). Esta técnica, aunque menos madura, permite inferir mucho más conocimiento y, por tanto, automatizar mayor cantidad de contenido, debido a la mayor riqueza del lenguaje en el que se describe el modelo (un lenguaje ontológico).

3.4. Falta de herramientas en diseño

El mayor cuello de botella existente en el desarrollo de videojuegos se produce en la fase de pre-producción, donde los diseñadores tienen muchas mecánicas que deben probar para así valorar cuales de esas mecánicas deben ser aceptadas, rechazadas o refinadas y, por lo general, los diseñadores necesitan varias pruebas para conseguir un resultado que les resulte satisfactorio. Diseñadores e investigadores buscan métodos alternativos que hagan accesible la creación de prototipos directamente a los diseñadores sin tener que valerse de los programadores (Katharine, 2012), de manera que que se permita crear prototipos a partir de la definición formal de una serie de características del juego (Silvano-Orita-Almeida y Soares-Correa-da Silva, 2013).

A día de hoy hay herramientas de producción simples que permiten a los diseñadores crear videojuegos si requerir conocimientos de progra-

mación, sin embargo, se puede observar que, cuanto más simples son de implementar mecánicas de juego con estas herramientas, más limitada está la creatividad del diseñador (Katharine, 2012) y no son por tanto herramientas válidas para prototipar (Nelson y Mateas, 2009) al no ser herramientas suficientemente flexibles.

El modelado visual ha demostrado, según ciertos diseñadores, que es un gran aliado para comunicar la visión que se tiene del juego o de una mecánica de juego a otros miembros del equipo (Silvano-Orita-Almeida y Soares-Correa-da Silva, 2013). Incluso, unos pocos investigadores han dejado constancia de este hecho implementando sistemas de modelado visuales que ayudan en el diseño de mecánicas de juego usando paradigmas como UML, redes de Petri o árboles de decisión (un tipo de flowcharts). En el caso, descrito en Sicart (2008), al usar el paradigma de la programación orientada a objetos (Sección 3.2), automáticamente puede modelar visualmente mediante el uso de UML todo su modelo de mecánicas de juego, reglas, objetivos, etc. Por otro lado, las redes de Petri no son solo una representación matemática sino que también se pueden representar gráficamente como un diagrama con entrada y salida de datos, condiciones, eventos, tareas y claúsulas y arcos que las conectan entre sí. De hecho, Araújo y Roque (2009) propone el uso de esta representación visual en su trabajo con redes de Petri como lenguaje de modelado. Otro ejemplo es el uso de árboles de decisión para representar todas las posibles situaciones donde el jugador puede tomar una decisión y las consecuencias de dichas decisiones (Salen y Zimmerman, 2003).

Desde el punto de vista de la industria hay varias soluciones que se han ido incorporando paulatinamente y que permiten definir gráficamente comportamientos de personajes, coreografías entre éstos e incluso el *gameplay* de una partida. Buenos ejemplos de ésto son *Unreal Kismet*[2], *Flowgraphs*[3] de *Crytek*, *XaitControl*[4] o *PlayMaker*[5] para *Unity*[6], los cuales nos permiten definir visualmente reacciones ante eventos del juego o definir rutinas básicas que deban realizarse por los personajes o elementos del entorno.

Las herramientas presentadas anteriormente demuestran los esfuerzos que se han realizado en los últimos años tanto en el mundo académico como en la industria para introducir herramientas para el diseño

[2] http://www.unrealengine.com/features/kismet/
[3] http://freesdk.crydev.net/display/SDKDOC2/Flow+Graph+Editor
[4] http://www.xaitment.com/english/products/xaitcontrol/xaitcontrol.html
[5] http://www.hutonggames.com/
[6] http://unity3d.com/

de comportamientos y *gameplay* en videojuegos. Sin embargo dichas herramientas, que permiten describir reacciones a eventos, toma de decisiones, secuencias de acciones, etc., no permiten describir los elementos que componen el juego de una manera gráfica e intuitiva. Machado et al. (2009) propone describir los elementos del juego mediante el uso de ontologías y para ello usan *Protégé*[7], que les permite definirlas gráficamente, de manera que (aquí si) se pueden modelar formalmente los diferentes elementos del juego y las relaciones que existen entre ellos. De la misma manera, se encuentra en desarrollo un producto comercial más extenso y ambicioso llamado *XaitKnow*[8] y que permitirá modelar el conocimiento global del videojuego mediante el uso de ontologías y relaciones entre los diferentes objetos de las ontologías. Extraer el conocimiento a una ontología permite no solo poder visualizar el conocimiento de manera más sencilla sino que también agiliza el desarrollo de videojuegos permitiendo generar nuevo contenido de juego sin necesidad de recompilar, aumenta la reusabilidad y extensibilidad de los componentes de juego y abre la puerta a mecanismos de aprendizaje ya que los diferentes NPCs tiene conocimiento parcial del estado del mundo.

Al final, lo que estas herramientas ofrecen al diseñador es una forma de definir de manera visual un modelo semántico del videojuego y de generar contenido de juego procedimentalmente. Puede que no sea posible suprimir por completo la figura del programador en la fase de prototipado, pero si se debería minimizar todo lo posible y se debería tratar de mejorar la comunicación entre los miembros del equipo.

[7]`http://protege.stanford.edu/`
[8]`http://www.xaitment.com/english/products/xaitknow/xaitknow.html`

Capítulo 4

Metodología de desarrollo basada en ontologías

En este capítulo se presenta una una metodología, para el desarrollo de videojuegos, que gira en torno a un dominio formal. La ontología que se modela guía todo el desarrollo ya que sirve como especificación de los requisitos, contrato entre programadores y diseñadores y luego es usada durante el proceso de implementación y generación de contenido ya que hay una estrecha relación entre la especificación formal y la implementación final. En este capítulo se explica en profundidad cómo se puede usar una representación ontológica como medio de comunicación entre programadores y diseñadores, explicando el proceso, cómo puede este dominio guiar un desarrollo iterativo y cuáles son los puntos potencialmente conflictivos. En el Capítulo5 se explicarán técnicas específicas que usan el dominio y completan la metodología.

4.1. Introducción

El desarrollo de videojuegos es una disciplina especial dentro del mundo del desarrollo software. La naturaleza multidisciplinar del equipo de desarrollo y el componente artístico que se introduce en esta especialidad de la ingeniería del software provoca que las técnicas propuestas en las metodologías clásicas de desarrollo no funcionen. Aquí las ideas y mecánicas propuestas deben ser probadas para saber si son válidas y si aportan lo que se esperaba de ellas. Esto lleva a que el proceso de desarrollo deba ser iterativo ya que, aunque de partida hay un plan de trabajo, una descripción de qué es lo que se quiere realizar, las ideas y mecánicas del juego siguen surgiendo durante todo el desarrollo, se van modificando y algunas se van descartando.

A lo largo de los años la tecnología para soportar este tipo de desarrollos ha ido mejorando, y ahora contamos con arquitecturas que son altamente flexibles, permitiendo cambios en el dominio y potenciando la reutilización de código. De la misma manera han ido surgiendo y evolucionando metodologías que soportan este tipo de desarrollos, donde se premia la creación de funcionalidad inmediata en lugar de creación de software potencialmente escalable que no se sabe si se acabará usando.

Además, el componente artístico provoca que personas provenientes de disciplinas muy diferentes trabajen codo con codo, dando lugar a complicaciones en la comunicación ya que ésta se realiza entre personas que tienen métodos de trabajo muy diferentes. Dos de los roles que más interactúan durante el desarrollo son el diseñador y el programador ya que el diseñador es el que idea las mecánicas de juego, personajes, historia y comportamientos que el programador debe implementar para que, nuevamente el diseñador, evalúe el resultado y decida si la funcionalidad obtenida es la deseada. Al tratarse de un desarrollo iterativo, esa comunicación copa todo el desarrollo y debe agilizarse lo más posible.

Tradicionalmente la comunicación entre programadores y diseñadores se ha llevado a cabo mediante la creación de un documento de diseño textual y, de manera opuesta a lo que sucede con el software y con las metodologías, a día de hoy gran parte de la comunicación sigue realizándose de esta manera.

El lenguaje natural escrito es un medio de comunicación bastante lento y potencialmente ambiguo por lo que nuestra finalidad es proponer una metodología de desarrollo donde:

1. La base de la comunicación se agilice, tanto para el emisor como para el receptor del mensaje.
2. Esta comunicación sea lo menos ambigua posible.
3. Toda esa información usada para comunicarse pueda ser reaprovechada en otros contextos del desarrollo.

Nuestra propuesta metodológica gira en torno a una representación formal del dominio de juego, donde las diferentes entidades (cualquier elemento que participa en la jugabilidad) son descritas de manera semántica usando ontologías (Web Ontology Language OWL[1] en nuestra implementación). Para simplificar la edición y visualización de ese dominio formal proponemos el uso de herramientas que permitan mostrar y editar dicha información de una manera visual. A su vez, y aprovechando la versatilidad que nos ofrece un lenguaje como OWL, adaptamos la

[1] http://www.w3.org/TR/owl-features/

interfaz o la forma en la que se muestra la información en función del rol que desee consultarla ya que un diseñador querrá visualizar y editar el dominio en una vista más simplificada y de más alto nivel mientras que un programador necesitará ver y editar detalles más relacionados con elementos tecnológicos al dominio del juego y la implementación del software.

Una representación formal ayuda a limitar la ambigüedad, hacer validaciones de dominio o chequeos de consistencia y mejorar la calidad y automatización del desarrollo completo de sistemas complejos (Tetlow et al., 2006). Como no podía ser de otra manera, nuestra propuesta se basa en el uso de este tipo de representación formal, en el uso de metodologías ágiles como scrum y también de una tecnología flexible como la arquitectura basada en componentes. En esta tesis se propone una metodología iterativa en la que participen de manera conjunta diseñadores y programadores y que permita un prototipado rápido de iteración constante para la adición y modificación de nueva funcionalidad. Al basarse en una metodología tipo scrum, la parte de producción no se diferencia demasiado de la de prototipado, por lo que durante la producción también es flexible a cambios y agiliza las iteraciones y la velocidad de desarrollo.

En la Sección 4.2 se comentan las ventajas que se obtienen de usar un dominio formal como vía de comunicación entre programadores y diseñadores. Éste complementa el *documento de diseño* y cada rol tiene una visualización diferente del mismo. En el Capítulo 4.3 se entra en la explicación concreta de la metodología basada en ese dominio formal, explicando sus pasos, los papeles que juegan diseñadores y programadores, resaltando los puntos potencialmente conflictivos que se resolverán en el Capítulo 5 y poniendo un ejemplo de desarrollo.

4.2. Modelado del dominio como contrato entre programadores y diseñadores de videojuegos

4.2.1. Dominio formal como complemento al documento de diseño

Como ya hemos anticipado, la metodología y tecnología usadas en los desarrollos de videojuegos se han adaptado para mejorar la flexibilidad y permitir desarrollos más ágiles y rápidos que permitan probar lo más rápido posible las ideas modeladas. Sin embargo la comunicación sigue

siendo, mediante la documentación, básicamente textual y por tanto lenta y estática.

Nuestra propuesta para solventar los problemas previamente descritos es usar un dominio formal (una ontología) como base de la comunicación entre diseñadores de juego y programadores. Para que la comunicación cumpla su función, dicha ontología debe ser fácilmente editable y debe ser de lectura sencilla, presentando información en diferentes capas dependiendo de la granularidad con la que ésta se quiera consultar. El desarrollo software dirigido por ontologías (ontology-driven software construction) es una técnica usada y documentada en otros ámbitos de la ingeniería del software (Deline, 2006), donde el uso de ontologías es bien recibido por los desarrolladores como método de comunicación y punto de partida para un desarrollo software.

El dominio formal usado para la comunicación y que se establece como contrato entre los diseñadores y los programadores se propone como un complemento al documento de diseño tradicional modelando el conjunto de entidades de juego. Por entidades de juego nos referimos a todos aquellos elementos autocontenidos de lógica que pueden realizar diferentes tareas () como renderizarse a si mismos, buscar y seguir caminos o tomar decisiones (Bilas, 2002). Ejemplos comunes de estas entidades de juego son los avatares de los jugadores, elementos interactivos o incluso elementos del decorado como puertas, árboles o edificios, pero también se consideran entidades elementos menos obvios tales como secuencias de cámara, puntos de ruta o elementos detonantes que controlan puntos de la trama del juego.

Se debe que recalcar que este dominio formal no viene a sustituir por completo al documento de diseño, sino a una parte concreta de éste, aquella que se refiere a las entidades de juego o a lo que nosotros denominamos el dominio de juego. El documento de diseño es una herramienta en la que se describen muchos más factores del juego como pueden ser las mecánicas de éste, la historia, la estética, los comportamientos de los enemigos, la ambientación y un largo etcétera. Todos esos detalles no estarían abarcados por nuestra descripción formal del dominio de juego, y deberían seguir describiéndose por las vías tradicionales. Considerar la inclusión únicamente del dominio de juego se debe a que es una de las partes más sensibles a los cambios y está relacionada con prácticamente la totalidad de los otros elementos de la descripción del juego, siendo la base o cimientos donde este se sustenta.

Debido a los argumentos presentados en la Sección 2.3, sabemos que encontrar una mecánica divertida en el desarrollo de videojuegos es un proceso difícil que requiere de varias iteraciones y refinamientos

hasta obtener el resultado deseado. Por esta razón, la industria necesita de técnicas de software más flexibles y extensibles que permitan que el desarrollo de funcionalidad sea tan rápido como sea posible y una de las consecuencias en el desarrollo profesional de desarrollo de videojuegos es el uso de la arquitectura basada en componentes (Sección 2.4.2) para la representación de las entidades de juego.

Como ya se vio, este tipo de arquitectura potencia la reusabilidad y la extensibilidad de código y, por tanto, es una buena solución desde el punto de vista de la ingeniería del software. Sin embargo tiene problemas frente a arquitecturas más tradicionales y estáticas como pueden ser aquellas basadas en herencia. Está claro que, aunque la herencia de clases presenta sus problemas en desarrollos tan cambiantes, ésta puede ser vista como una ontología conceptual de entidades. En cierta manera, dicho árbol de herencia es una definición no formal del modelo de juego. La mayor ventaja de las jerarquías es que encajan muy bien con el pensamiento humano. Cuando se *diseña* un juego, los diferentes elementos (enemigos, objetos, decoración...) son habitualmente categorizados usando una ontología (Gruber, 1993) que modela la relación *es-un* entre todos ellos. Esta ontología de entidades es fácilmente traducida en el código con herencia, de manera que ambos, diseñadores y programadores, manejan un modelo conceptual similar.

Usando un sistema basado en componentes se pierde esta ontología conceptual, ya que las entidades se crean en tiempo de ejecución por composición, de manera que no hay ninguna relación entre las diferentes entidades. Además una gran cantidad de información se queda escondida detrás del código, empezando por que las entidades, al construirse en tiempo de ejecución, son mucho más complicadas de interpretar y depurar, siendo a su vez una posible fuente de inconsistencias. Estos motivos provocan que la arquitectura basada en componentes esté lejos de ser obvia en relación con el pensamiento humano y esa falta general de estructura produce el rechazo de algunos programadores (West, 2006; Chady, 2009).

Modelando el dominio con una ontología nuestra ambición es paliar los problemas que se presentan en la comunicación de dos roles que hablan diferentes “idiomas”, adaptando la representación del conocimiento semántico de manera que programadores y diseñadores tengan diferentes visualizaciones de éste, más cercanas a su manera de pensar. En resumen, los beneficios que se consiguen con esta alternativa son los siguientes:

- Reducir la ambigüedad que introduce el lenguaje natural

- Mejorar la velocidad de edición y de lectura del dominio
- Ofrecer vistas personalizadas del dominio en función del rol
- Generación procedimental de contenido de juego
- Distribución semiautomática de funcionalidad en componentes software
- Detección automática de inconsistencias en el dominio del juego o en la edición de niveles

4.2.2. Representaciones del dominio para los diferentes roles

Si queremos adaptar nuestra metodología al desarrollo real que se realiza en la industria es obvio que el dominio que se pretende modelar debe ser compatible con la arquitectura basada en componentes. Hemos de tener en cuenta que este tipo de arquitectura es de por si complicada para los programadores por lo que no podemos plantearnos que los diseñadores se encarguen de representar el dominio en dichos términos. Al usar una ontología como base de la comunicación lo que si es factible es que los diseñadores representen el dominio de juego en un modelo simplificado y de más alto nivel que pueda luego ser extendido y completado por los programadores. Por lo tanto, la primera cuestión a resolver es hasta qué punto, o nivel de abstracción, deben definir los diseñadores el dominio del juego y qué detalles de bajo nivel serán los que tengan que implementar los programadores para tener una descripción completa.

La visualización de la ontología que manejen los diseñadores debe ser una representación de muy alto nivel, sin entrar en detalles técnicos de implementación y, según nuestra propuesta metodológica, se debe basar en una descripción de funcionalidad y estado de las entidades. Los diseñadores serán los encargados por tanto de modelar todas aquellas entidades que tengan un valor semántico para describir la lógica de juego. Para su descripción deberán primero organizarlas de manera jerárquica, de forma que se hace explícita esa distribución ontológica conceptual que se tenía con arquitecturas basadas en herencia. En estas distribuciones de tipo *es-un* los diseñadores representan no solo las entidades finales del juego sino que, igual que sucede en la herencia de clases, pueden definir entidades abstractas intermedias que engloban características comunes que describirán a las entidades que hereden de éstas.

Las entidades, además de estar relacionadas entre ellas de manera jerárquica, deben ser descritas. Nuestra propuesta determina que se debe definir las entidades en función de su *funcionalidad* y su *estado*. Desde el punto de vista de la funcionalidad, los diseñadores definen qué *acciones* pueden realizar las entidades (generalmente denotadas por un verbo como andar, atacar, abrirse...) y qué eventos o *percepciones* pueden percibir las entidades (representados en muchas ocasiones por participios como enemigo visto, objetivo perdido, tocado...). El estado de una entidad se modelaría en función de atributos (que se suelen denotar por nombres propios como vida, fuerza, velocidad...). Para definir y diferenciar mejor a las entidades, a estos atributos se les pueden asignar valores iniciales por defecto que, aunque luego las instancias concretas del juego se podrán personalizar, dan una idea de las características que tendrán. Por último se permite añadir una pequeña descripción textual de las entidades de manera que, además del apoyo visual de la distribución de las entidades y su descripción de funcionalidad y estado, se pueda definir algún detalle de grano fino que pueda ser interesante para los programadores.

Aunque en la Sección 4.3.2 veremos un ejemplo más completo y elaborado, utilicemos aquí el conocido juego de *Super Mario Bros* para un primer ejemplo. Pongamos que un diseñador quiere definir al personaje protagonista del juego. En primer lugar situaría la entidad de *Mario* en la ontología, donde esta tendría el concepto *Personaje*, que puede ser de tipo *Enemigo* o *Jugador*. *Goomba* o *Koopa Troopa* serían *Enemigo*s en dicha descripción mientras que *Mario* o *Luigi* serían *Jugador*es. Los conceptos de *Personaje*, *Jugador* y *Enemigo* no serían más que conceptos abstractos que ayudan a entender las relaciones entre las entidades finales del juego, pero que en la implementación final del juego no existirán. A la hora de añadir funcionalidad, *Mario* es capaz de *andar*, *correr*, *saltar*, *entrar* en las tuberías, etc. las cuales serían las acciones que describirían al jugador. El estado de *Mario* podría ser un simple atributo que podría tomar valores de *pequeño*, *grande* y *disparo*. Una descripción de este estilo es lo que se le exigiría a los diseñadores.

La experiencia muestra que el diseño de videojuegos no debe solo ser llevado a cabo por los diseñadores sino que también se debe tener en cuenta al equipo de programación. Después de todo, los programadores son los responsables de llevar a cabo los retos tecnológicos que el juego impone y conocen donde está el límite de la plataforma sobre la que se desarrolla (Long, 2003), por lo que deben restringir a los diseñadores cuando estos son excesivamente ambiciosos. A nivel de diseño las cosas se definen a grano grueso pero aun así se pueden detectar

elementos o descripciones que podrían ser un riesgo para el desarrollo futuro del juego. En este sentido los programadores y diseñadores deben llegar a un acuerdo de viabilidad, que finalmente será interpretado como un contrato de mínimos que los programadores se comprometen a implementar.

Los programadores, partiendo del dominio de los diseñadores, deben generar el prototipo o producto que implemente la funcionalidad requerida por lo que su visión de dicho dominio debe ser mucho más cercana a la implementación real. El nivel de detalle debe ser mayor y, obviamente, para que esto sea posible se les debe ofrecer una visión del dominio mucho más específica y más orientada a una arquitectura basada en componentes, que es como finalmente se implementará el juego. Los programadores deben añadir (o modificar) entidades, mensajes y atributos que aporten al dominio información más específica de la implementación, aunque quizás esa información no tenga tanta relevancia semántica, pero además, en el dominio se debe añadir el concepto de *componente* software. Los programadores por tanto distribuirán la funcionalidad en componentes software, donde lo que antes se representaba como *acciones* y *percepciones* ahora será representado por *mensajes* que intercambian dichos componentes.

Siguiendo el ejemplo anterior los programadores podrían añadir entidades lógicas con las que los diseñadores no contaban, como puede ser la representación de la cámara, y extender la descripción de Mario desde el punto de vista de implementación. Mario debería poder *renderizarse*, *reproducir animaciones* o *colisionar* con ciertos elementos, por lo que se incrementarían los mensajes que la entidad recibe, y también deberá tener nuevos atributos que contengan el *modelo gráfico* de Mario y su radio de colisión para simular la física. Finalmente, los diferentes elementos de la descripción deberán estar en diferentes componentes como podría ser el componente de *control* que tendría los mensajes de *andar*, *correr*, *saltar* y *entrar* o el componente *gráfico* que podría *renderizar* y *reproducir animaciones* y contendría el *modelo gráfico* de la entidad.

En resumen, los diseñadores y programadores trabajan sobre el mismo dominio aunque la vista que tienen los dos roles de este dominio son diferentes. Los diseñadores tienen una vista semántica de más alto nivel basada en distribuciones ontológicas, entidades, acciones, percepciones y atributos mientras que los programadores tienen una vista más técnica del dominio, basada en componentes software que conforman entidades, mensajes que se intercambian los componentes y atributos. Este dominio subyacente sirve sin embargo como base del acuerdo que

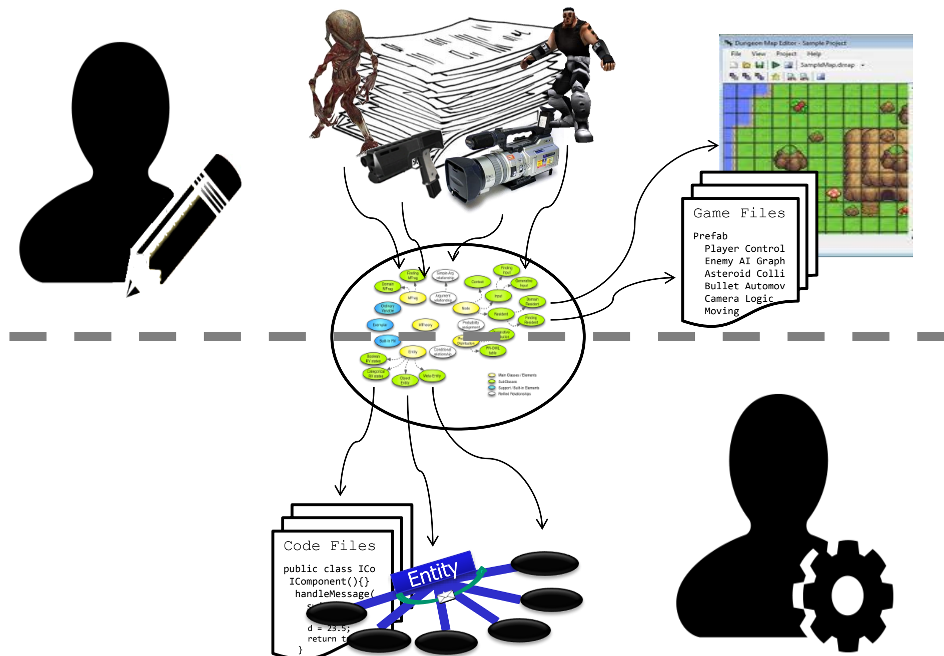

Figura 4.1: Metodología de desarollo basada en un dominio formal

se debe alcanzar entre los dos roles aportando dinamismo y flexibilidad a la comunicación. Cada cambio introducido en una nueva iteración es más fácilmente reconocible y analizable que modificar el documento de diseño. En la siguiente sección se explicará con más detalle como sería el proceso de creación de un videojuego y como este dominio sirve, además de como contrato entre diseñadores y programadores, para otras tareas del desarrollo.

4.3. Procedimientos efectuados en el desarrollo de la metodología iterativa

En la sección anterior se han comentado cuales serían las bases de la comunicación entre diseñadores y programadores, que se agilizará mediante el uso de un dominio formal. En esta sección se explica como se podrían aplicar estas ideas en un desarrollo completo de un videojuego, donde se premie la generación rápida de funcionalidad. Esta metodología es un ejemplo concreto del uso de ontologías en ingeniería del software (Seedorf et al., 2006), donde el dominio formal que se creará durante el desarrollo no puede estar escrito en piedra, sino que debe poder ser cambiado y adaptado durante la evolución del desarrollo. Esto

también debería ser cierto para los documentos de diseño pero, como se explica en la sección anterior, son difíciles de actualizar una vez que el desarrollo ha empezado. Afortunadamente, tener un dominio formal aporta suficientes ventajas para convencer tanto a programadores como a diseñadores de tenerlo actualizado. La Figura 4.1 muestra una vista conceptual de cuáles son las responsabilidades de diseñadores y cuáles la de los programadores, denotando que el nexo de unión entre los dos roles es, precisamente, la ontología. La metodología que se extrae de esta representación promueve y facilita desarrollos ágiles de manera incremental e iterativa donde las fases del desarrollo de una de estas iteraciones se podrían dividir en los siguientes procedimientos:

1. *Modelado de un dominio formal:* Diseñadores y programadores colaboran en la descripción del dominio del juego.

    a) *Definición ontológica desde el punto de vista de diseño:* Junto con el documento de diseño clásico los diseñadores definen formalmente (o cambian) las entidades de juego y sus habilidades usando una interfaz gráfica que simplifica la edición y visualización de los conceptos.

    b) *Aceptación del modelo:* Los programadores analizan, negocian y validan el nuevo dominio de juego, que se convierte en un contrato entre diseñadores y programadores.

    c) *Extensión desde el punto de vista de la implementación:* Los programadores enriquecen el dominio inicial añadiendo detalles técnicos. Además identifican los componentes software de manera que toda las acciones y atributos que describen las entidades acabe encapsulada en piezas de de funcionalidad autocontenida.

2. *Generación de contenido:* Una vez que el dominio de una iteración concreta ha sido modelado los programadores y los diseñadores deben generar el contenido del juego representado en la ontología.

    a) Los programadores deben implementar la funcionalidad descrita en la plataforma y lenguaje de programación previstos y deben crear los recursos necesarios para que los diseñadores puedan generar el contenido que les es encomendado.

    b) Los diseñadores son los encargados de, usando los recursos creados por los programadores, crear los diferentes niveles del juego y de definir la lógica de juego y comportamientos de las entidades.

Antes de entrar más en detalle de cada una de estas fases debemos mencionar que en los diferentes pasos de la metodología se hace referencia a dos roles: diseñador y programador. Sin embargo, en grandes desarrollos, podemos ser más específicos detallando los roles existentes en el equipo de desarrollo. El tipo de diseñador que define la ontología de entidades mediante la herramienta visual, debe ser un diseñador con conocimiento técnico. Esta *diseñador técnico* es el que negocia con los programadores cuáles serán los requerimientos del juego en términos de funcionalidad y, por lo tanto, deberá conocer cuál es la *manera de hacer las cosas* de los programadores. Del mismo modo, el tipo de programador que completa la ontología del juego debe tener una visión global acerca del desarrollo del juego, especialmente de su lógica. Por tanto, este *programador jefe* de la parte de la lógica es el que diseña la arquitectura software y se comunica con el *diseñador técnico* para llegar a un acuerdo. Finalmente, hay *programadores estándar* que implementan la funcionalidad en un lenguaje de programación concreto y *diseñadores de niveles* que crean mapas y comportamientos para los personajes. En desarrollos más pequeños sin embargo los roles se encuentran más mezclados. En el resto de la tesis se hablará de diseñadores y programadores por simplificar, pero teniendo en cuenta que puede haber roles más concretos.

Cabe recordar nuevamente que, aunque este proceso se presenta en cascada, la metodología soporta iteraciones por lo que las diferentes fases pueden ser realizadas las veces que sea necesario. Los desarrollos dirigidos por ontologías como este son ya de por sí procesos iterativos por lo que pueden influenciar el uso de arquitecturas dirigidas por modelos (Model Driven Arquitecture) (Pastor y Molina, 2007) que encajan muy bien con los desarrollos ágiles, que es lo que estamos buscando (Deline, 2006).

El primer punto de la metodología ha sido avanzada en la sección anterior, de manera que la creación comienza por tanto por los diseñadores, los cuales definen por completo la funcionalidad que se quiere añadir en una iteración concreta. La jugabilidad se proporciona por medio de entidades clasificadas ontológicamente en una jerarquía, donde estas tienen un estado (en forma de *atributos*) y un conjunto de funcionalidades (representadas por *acciones* y *percepciones*).

Los atributos usados para describir las entidades pueden ser *simples* o *complejos* y se pueden estructurar ontológicamente. Los atributos *simples* son tipos “básicos” como enteros, reales, cadenas, vectores, etc. a los cuales se les pueden establecer restricciones de rango. Las restricciones pueden establecer un rango de valores, por ejemplo, la *vida* de

un personaje puede ser un atributo real donde $0 \leq vida \leq 100$ o un simple entero donde $vida \geq 0$, etc. También se pueden establecer una serie de valores válidos para una variable, de manera que el tipo físico de una entidad puede ser una cadena de caracteres que pueda tomar los valores “dinámico”, “estático” o “cinemático”. Además de los atributos *simples* se pueden tener atributos de tipos *complejos*, que simplemente se refieren a otros elementos del dominio, donde por ejemplo un atributo que define el *objetivo* de un personaje no jugador (NPC) puede ser de tipo *jugador*. El *jugador* sería un tipo de entidad descrita en el dominio que incluso podría ser abstracta y de la cual pueden heredar otras entidades más específicas que definan los diferentes jugadores. Estos serían los valores que podrían tomar ese atributo *objetivo*. Otro ejemplo podría ser que un componente de *inteligencia artificial* necesitase consultar el estado de otros componentes como por ejemplo el de *vida*. En ese caso, el componente de *inteligencia artificial* tendría un atributo cuyo tipo fuese en componente de *vida* y así poder acceder a él directamente.

Las acciones y percepciones que se describen se pueden jerarquizar del mismo modo que las entidades, de manera que sea más sencillo encontrar o inferir qué hacen las acciones en función de su colocación en la ontología. Estas acciones y percepciones pueden a su vez ser parametrizadas de manera que se incremente su funcionalidad y se pueda especificar con más detalle. Por ejemplo, una acción de tipo *andar* podría tener asociada un parámetro que indique la dirección en la que se debe andar y otro para especificar la velocidad. Del mismo modo, una percepción de un enemigo podría ser *jugador visto* y, si es un juego con varios jugadores tener asociado un parámetro que indique cual es la entidad concreta que ha sido detectada.

Esta información proporcionada por los diseñadores constituye la especificación semántica de la jugabilidad y, cuando los programadores aceptan toda la información descrita, podríamos decir que ese modelo semántico se convierte en un contrato acordado entre programadores y diseñadores acerca del juego que se va a crear. Si hubiese alguna decisión que comprometiese la implementación del juego los programadores deberían reportarlo antes de aceptar el nuevo dominio.

Una vez se llega a un acuerdo, los programadores son los encargados de dar vida a todas esas entidades que los diseñadores han concebido trasladando todo ese conocimiento a una implementación en un lenguaje de programación. Sin embargo, en vez hacer todo el trabajo desde cero y pelearse directamente con el código, que oculta mucha información semántica en diferentes clases, se aprovecha la representación ontológica del dominio y el esfuerzo que han realizado los diseñadores. Los pro-

gramadores deben por tanto añadir descripción semántica al dominio que, aunque pasa desapercibida desde el punto de vista del diseñador, es importante desde la perspectiva de la implementación. Esa modificación o ampliación semántica se realiza mediante la adición de nuevas entidades, atributos y mensajes (acciones y percepciones) o modificando alguno de los elementos ya que hay entidades a las que les puede faltar descripción, mensajes a los que le pueden faltar parámetros o atributos que no tengan un rango bien definido. Además se añade el concepto de componente a esta definición semántica del modelo del juego, por lo que las entidades son troceadas en componentes que contienen subconjuntos de funcionalidad y estado.

Todo este esfuerzo declarativo que realizan diseñadores y programadores hasta este punto tiene utilidad por sí mismo más allá de que una descripción formal agiliza la comunicación, ayuda a que se llegue a un acuerdo de una manera más rápida y dicho acuerdo deja menos lugar a la interpretación y posibles problemas futuros. Toda esa descripción semántica que almacenamos en OWL se puede aprovechar en esta y otras etapas del desarrollo para poder inferir información mediante el uso de razonadores. Por ejemplo, la tarea de crear una distribución de componentes puede ser en parte automatizada como se detalla en la Sección 5.1. Ante un conjunto de entidades descritas por una serie de atributos y de los mensajes que son capaces de llevar a cabo, se puede realizar una división semi-automática de responsabilidades, dando lugar a una colección de componentes con los que se podría implementar la jugabilidad descrita para dichas entidades. El proceso es semi-automático debido a que el programador podría realizar pequeñas modificaciones en la distribución sugerida para terminar de perfeccionarla previendo futuras necesidades.

Los componentes se pueden definir usando un grano más fino aún y, en lugar de simplemente describir que un componente puede interpretar ciertos mensajes, se puede describir que un mensaje será realizado si se cumplen una serie de requisitos. Esto es debido a que puede haber componentes de alto nivel que necesiten la ejecución de otras acciones para poder llevar a cabo una acción concreta, dividiendo o delegando parte de la ejecución de la acción. Esto además permite la aparición de polimorfismo, de manera que componentes de alto nivel pueden requerir que otro componente ejecute una subacción sin importar como ésta es llevada a cabo. Por ejemplo, un componente que lleva a cabo el mensaje o acción de *ataque* podría ejecutarla sin más, realizando un ataque a distancia, mientras que otro componente podría dividir la acción, requiriendo que otro componente ejecute una acción *ir-a* para acercarse

al enemigo y luego ya realizar un ataque cuerpo a cuerpo. Este segundo componente de alguna manera *depende* de que haya otro componente en la misma entidad que sea capaz de llevar a cabo acciones de tipo *ir-a*, sin importarle como se realiza dicha acción, solo le interesa llegar al punto de destino. Por tanto, podría haber un componente de *movimiento* que implementase la acción *ir-a* para entidades terrestres mientras que otro componente de *vuelo* llevaría a cabo *ir-a* mediante un desplazamiento aéreo.

En este punto de la metodología, el conocimiento semántico permite detectar posibles inconsistencias durante el diseño de entidades, anticipando problemas que, de otra manera, en este tipo de arquitectura solo serían detectadas durante la ejecución del juego. Problemas como que falten componentes que realicen acciones requeridas por otros componentes o que se establezcan valores inválidos para los atributos de una entidad son ejemplos de inconsistencias que no son fáciles de detectar durante la edición. En la Sección 5.3 veremos éstos y otros tipos de problemas que podemos detectar para aumentar el *feedback* que reciben programadores y diseñadores en este proceso y que evita muchos dolores de cabeza durante la generación final de contenido.

Toda la descripción semántica modelada es independiente de la plataforma de desarrollo y del lenguaje de programación a usar, lo que ha permitido realizar una descripción a alto nivel de una manera más sencilla no solo a los diseñadores, a los que se les ocultan todos los detalles de implementación, sino también a los programadores, que pueden pararse a pensar en una buena distribución de entidades y componentes sin tener que entrar en detalles engorrosos de programación. Esta aproximación ayuda a los programadores a generar mejores distribuciones al tener una visión más global pero además, esta independencia respecto a la plataforma y lenguaje a usar, permite que el mismo modelo pueda ser usado en más de una implementación por lo que si se desea portar un juego ya hecho a una nueva plataforma o se quiere realizar un juego para dos plataformas simultáneamente ambas implementaciones pueden compartir la misma definición semántica.

Una vez programadores y diseñadores han modelado el conjunto de entidades y componentes que describen la funcionalidad que se quiere añadir al juego, es el momento de generar el contenido real que implemente todas las características descritas. Para ello ya si que el equipo de desarrollo deberá atarse a una plataforma concreta, un motor de juego, un lenguaje de programación, etc. Como se explica en la Sección 5.2, el dominio formal puede ser usado para guiar la creación del contenido de manera que, gran parte de dicho contenido, puede generarse de manera

procedimental.

Podemos decir por tanto que la metodología que estamos introduciendo en el ciclo de desarrollo de videojuegos es una metodología Model-Driven Architecture (MDA) o, más concretamente, una metodología Ontology Driven Architecture (Tetlow et al., 2006; Deline, 2006).

La parte de generación de contenido, de la que se responsabilizan los programadores, consta de tres partes:

- Generación procedimental de código.
- Programación de la funcionalidad.
- Generación procedimental de los ficheros de descripción de las entidades.

En la generación procedimental de código se traduce todo el conocimiento almacenado en la ontología a código para el lenguaje y plataforma elegida. Lo que se genera es la lógica de juego que incluye la gestión de entidades, la entidad genérica, la representación de los mensajes que se envían y los ficheros que implementan los componentes con su inicialización, donde se establece el valor por defecto a sus atributos, y toda la gestión del envío y aceptación de mensajes. Por ejemplo, por cada tipo de mensaje se generará una estructura, que dependerá de la plataforma elegida y podrán estar representados desde por una clase hasta por un simple enumerado. Cada componente se traducirá en una clase que tendrá automatizada la recepción y ejecución de los mensajes que se le asignaron durante la edición del dominio e inicializará los valores de los atributos que tuviese asignados.

El trabajo que no puede ser generado de manera procedimental, y que es responsabilidad del programador, es implementar el comportamiento que se dispara cuando se recibe un mensaje. Es decir, si un componente recibe un mensaje de tipo *walk* con un valor concreto para el parámetro *speed*, el programador tendrá que hacer que la entidad se desplace por el entorno (modificando su posición) a la velocidad indicada y, posiblemente, deberá enviar un mensaje de tipo *setAnimation* que será procesado por otro componente para animar la entidad. Básicamente, por cada mensaje recibido por un componente el programador tendrá que completar un método que ha sido automáticamente generado vacío. Esta implementación de funcionalidad sería el único trabajo de programación que debería hacer el programador encargado de la lógica de juego, ya que toda la *fontanería* de gestión de entidades, comunicación de éstas, su construcción dinámica, ciclo de vida, inicializaciones, etc. se crea de manera automática.

El último paso en la creación procedimental de contenido de juego es la creación de los ficheros que describen las entidades y que serán usados tanto por el motor del juego como por el editor de niveles. Este tipo de ficheros, comúnmente llamados *bueprints + arquetypes* o *prefabs* contienen las definiciones de entidades, describiéndolas en función de sus componentes y proporcionando valores por defecto a sus atributos. Estos ficheros, externos al código, son básicos en la edición de niveles. Al fin y al cabo, las diferentes entidades del dominio son las piezas fundamentales con las que cuenta el diseñador para montar los mapas donde transcurre la acción. Sin ellas no se pueden describir los niveles.

Como no podía ser de otra forma, la generación de contenido de juego no es solo responsabilidad de los programadores sino que los diseñadores tendrán también gran carga de trabajo. El trabajo de éstos en la generación de contenidos se concentrará en la creación de niveles, donde están a cargo de tres tareas principales:

- Crear el entorno del mapa usando modelos estáticos como paredes, pasillos, terrenos, árboles, edificios o lo que se necesite en el nivel.
- Añadir interactividad por medio de entidades, usando los prototipos de entidades definidos en el dominio formal, que contienen los componentes que los programadores crearon.
- Definir los comportamientos de los personajes usando alguna estructura de alto nivel y incorporar esos comportamientos a las entidades.

Con ayuda de un editor de mapas o niveles los diseñadores deben distribuir las entidades que definieron en la primera fase de la metodología. Los diseñadores podrían estar tentados a crear sus propias entidades, mediante la unión de componentes creados por los programadores, para hacer pruebas rápidas de ideas sin tener que hacer todo el ciclo de desarrollo. Esto no es necesariamente malo ya que se pueden descartar malas ideas rápidamente o encontrar mecánicas que funcionen y que luego mejorar al definirlas formalmente, sin embargo ha de hacerse con cuidado ya que puede acarrear problemas de consistencia.

Es importante darse cuenta que las entidades han sido creadas implementando las acciones y percepciones que fueron descritas, pero en *ningún momento* se define en el dominio formal ni el código *cuándo* se usan, ya que eso depende del mapa concreto y su nivel de dificultad. Es decir, una entidad *NPC* ha sido definida de manera que es capaz de *per-*

cibir enemigos, de *andar* o de *disparar* pero, hasta ahora, en ningún sitio se especifica cuándo se deben realizar acciones o si las percepciones desencadenan reacciones. Todo esto es ahora también responsabilidad de los diseñadores, que deberán definir los comportamientos de los personajes mediante algún modelo de comportamiento como las máquinas de estado (FSM por sus siglas en inglés) o los árboles de comportamiento (BT).

El dominio formal puede tener también un papel importante en la edición de niveles, no solo por los ficheros generados con las descripciones de las entidades, sino que puede ser usado para proporcionar *feedback* a los diseñadores. Se puede indicar cuando se están creando mapas, entidades o comportamientos inconsistentes o si alguno de estos elementos se vuelven inconsistentes ante una nueva modificación del dominio (Sección 5.3.3) e incluso proponer alternativas o dar sugerencias de edición. De esta manera los diseñadores pierden el miedo a realizar cambios y pueden dejar volar su imaginación y encontrar mecánicas o situaciones interesantes que en otro caso igual hubiese obviado.

4.3.1. Puntos potencialmente conflictivos

Una de las cosas que debe garantizar la implementación de esta metodología es la consistencia entre iteraciones, de manera que el esfuerzo realizado por diseñadores y programadores en la creación de contenido para una iteración no se pierda en una nueva iteración. Hay tres partes de la metodología que son especialmente críticas cuando se itera:

- Nuevas sugerencias de distribución de componentes.
- Generación procedimental del código.
- Adaptación de niveles y comportamientos de personajes ante cambios en el dominio.

Si los programadores han decidido distribuir el comportamiento de las entidades en componentes ayudados por la distribución automática de componentes (Sección 5.1), este proceso puede requerir de pequeñas modificaciones por parte del programador. Ante una distribución sugerida, el programador puede decidir renombrar algún componente, partir un componente muy grande en dos más pequeños o agrupar funcionalidad de varios componentes en uno solo más grande. En una nueva iteración, donde al introducirse nuevos elementos de juego las descripciones de las entidades varían, si el programador usase nuevamente la distribución automática de componentes el resultado sería bastante similar

al que obtuvo en la iteración anterior, aunque con pequeños cambios fruto de la modificación del dominio. Si en cada nueva iteración el programador tuviese que realizar las mismas pequeñas modificaciones que hizo en las iteraciones pasadas para adecuar la distribución a sus necesidades, la técnica dejaría de ser productiva e incluso podría causar inconsistencias relacionadas con los mapas y comportamientos editados en iteraciones pasadas. Por tanto, habrá que poner especial cuidado en almacenar esas alteraciones realizadas en la distribución por los programadores y aplicarlas automáticamente a las distribuciones sugeridas en futuras iteraciones.

Una vez se han diseñado todos los componentes y se usa la generación procedimental de código (Sección 5.2) ésta crea una gran cantidad de código automáticamente en función de los elementos definidos en el dominio. Sin embargo, los programadores acaban describiendo la funcionalidad esperada en los huecos (métodos vacíos) preparados para ello en los componentes generados. Si en cada iteración se vuelve a generar el código sin tener en cuenta esas implementaciones de funcionalidad la generación automática de código dejaría de tener sentido. Por tanto, todos los componentes definidos en el dominio formal deben estar directamente relacionados con el código que los implementa y todos los mensajes que se especifican para un componente del dominio formal deben a su vez estar directamente relacionados con los métodos que han implementado los programadores. De esta manera, aunque se realicen cambios de alto nivel en la arquitectura (cambios en el dominio formal), renombrando mensajes o componentes, fusionando o dividiendo componentes, etc. las implementaciones de las funcionalidades deben respetarse y, ante una nueva generación, el código añadido debe mantenerse y colocarse donde corresponda. Esto además permite que arquitecturas o distribuciones potencialmente poco escalables puedan modificarse a alto nivel, modularizando componentes, redistribuyendo funcionalidad entre componentes (moviéndola de unos a otros), etc. y que el código automáticamente se adapte sin necesidad de la intervención del programador, facilitando así la creación de funcionalidad rápida ya que durante el diseño no es necesario dedicar horas y horas para diseñar la mejor arquitectura posible.

Durante las diferentes iteraciones los diseñadores van creando niveles de juego basados en los dominios que acaban de ser o están siendo implementados pero, según avanza el desarrollo y esos dominios evolucionan, los niveles creados previamente podrían perder consistencia y dejar de funcionar, por ejemplo, cuando se eliminan tipos de entidades o cuando se modifica la funcionalidad de alguno de estos tipos (aña-

diendo, quitando o modificando componentes). Si el objetivo es acelerar la generación de nuevo software funcional y permitir el desarrollo en paralelo de programadores y diseñadores, se deberán suministrar herramientas de gestión que permitan garantizar que ninguna parte del proyecto se quede obsoleta. El uso del dominio formal en OWL, además de permitirnos detectar inconsistencias durante la edición de niveles o de comportamientos de personajes, nos permite comprobar si, niveles y comportamientos editados en iteraciones pasadas (y que por tanto se basaron en una versión anterior del dominio) siguen siendo consistentes con las nuevas especificaciones del dominio, o sino detectar que elementos deben ser modificados para que mantengan esa consistencia (Sección5.3). Lo que se consigue con esto es que desde el principio del desarrollo se puedan estar haciendo niveles de juego que no sean solo pruebas aisladas sino que terminen en el juego final, garantizando su correcto funcionamiento desde el punto de vista semántico.

Uno de los puntos importantes de esta metodología es que permite el trabajo en paralelo en cualquiera de las fases por lo que, al mismo tiempo, puede haber programadores implementando funcionalidad en código, diseñadores de niveles montando mapas de juego y diseñando comportamientos para los personajes mientras que otros diseñadores y programadores trabajan en la distribución formal de las entidades en una iteración posterior. Todo esto es posible obviamente gracias al uso del dominio formal que nos permite mantener la coherencia y a haber detectado y diseñado mecanismos para poder evitar esas partes críticas que potencialmente generan problemas durante el desarrollo.

4.3.2. ejemplo

Para ejemplificar como sería el proceso metodológico imaginemos cómo podría ser una pequeña iteración del desarrollo de un juego inspirado en la conocida saga *Half-Life*. Estando en un proceso intermedio del desarrollo, podríamos tener una descripción de los personajes del juego como la que se muestra en la Figura 4.2. Esta figura muestra una vista parcial de lo que sería el conjunto completo de las entidades de juego, donde se los diferentes *personajes* del juego se encuentran categorizados en función de si están controlados por un *jugador* humano o no lo están y, en este segundo caso si son *aliados* o *enemigos*.

Dentro de los *enemigos*, hay un conjunto de entidades catalogadas y, en este momento, los diseñadores deciden añadir una nueva clase de enemigo, el *Zombie*, que a su vez podría tener diferentes características y comportamientos en función de su tipo específico. El primer lugar, los diseñadores situarían las entidades en la ontología (Figura 4.2) donde

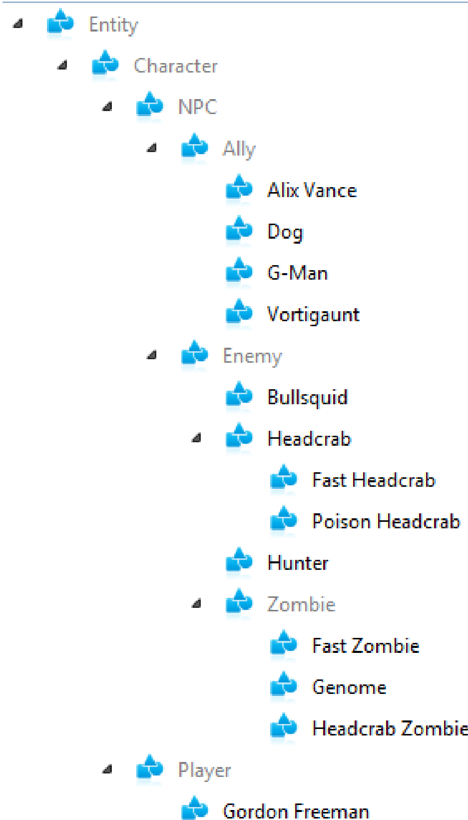

Figura 4.2: Vista parcial de la jerarquía de entidades del *Half-Life*

Zombie tendría sus especializaciones, tal y como deseaban los diseñadores. Los conceptos de *Character*, *Player*, *NPC*, *Ally*, *Enemy* o incluso *Zombie* no serían más que conceptos abstractos que ayudan a entender las relaciones entre las entidades finales del juego, pero que en la implementación final del juego no existirán.

A la hora de añadir funcionalidad, los diseñadores describirían un *zombie* como una entidad que es capaz de *andar*, *correr*, *saltar*, *atacar*, etc. por lo tanto estas serían las acciones obvias con las que el diseñador describiría al *zombie*. Además es capaz de reaccionar a ciertos eventos o percepciones como *enemigo visto*, *objeto visto*, *enemigo perdido de vista* o *disparo recibido*, los cuales servirán para desencadenar diferentes comportamientos y reacciones de los enemigos. El estado de un *zombie* desde el punto de vista de un diseñador sería modelado por atributos tales como la *salud*, la *velocidad* con la que se mueve o por la *fuerza*.

Los programadores recibirían un dominio donde los *zombie*s estarían descritos por las acciones y atributos mencionados anteriormente y, a

esta descripción, le añadirían funcionalidad más técnica y de bajo nivel. Las acciones y las percepciones estarían representados por mensajes en la vista del programador, y al ejecutar la mayoría de esas acciones se debería ofrecer *feedback* al jugador. Uno de los mensajes que debería poder llevar a cabo un *zombie* sería el de *poner una animación* o el de *reproducir un sonido* y usarlos para ofrecer al jugador una respuesta visual o sonora que le hagan intuir qué es lo que está haciendo el *zombie*. Otro ejemplo de *mensajes* de bajo nivel a los que respondería un *zombie* sería aquel que denota que se ha *producido una colisión*, de manera que el comportamiento del enemigo varíe en función de cuando colisione, con qué elemento, etc. Igual que los programadores tienen que añadir mensajes técnicos a la descripción del *zombie*, también deben añadir atributos que por ejemplo doten a la entidad de una representación gráfica mediante un atributo de *modelo gráfico*, que establezcan un *identificador* único u otro que defina el *radio* de la entidad para poder calcular en el motor de física cuando colisionará la entidad, evitando que atraviese elementos del juego como paredes y activando los eventos de *colisión recibida*.

Cuando los programadores terminan de definir las entidades, donde la entidad *Zombie* acabaría con los mensajes y atributos de la Figura 4.3, deberían agrupar los mensajes y atributos que estén directamente relacionados en componentes. Algunos puede que existiesen de iteraciones anteriores, que fuesen usados por otras entidades, otros deben tener que crearse nuevos y en ocasiones se debe modificar algún componente antiguo o refactorizar la distribución anterior. Un ejemplo de componente sería el *componente gráfico* que reuniría el atributo con el *modelo* 3D que representa la entidad junto con las acciones que *reproducen* o *paran* una animación. En otro ejemplo, el componente de *control* del la entidad, estarían los mensajes de *andar*, *torcer*, *correr*, *saltar* y los atributos con la *velocidad de andar* y la *altura de salto*.

Una vez que está todo el dominio modelado, se procede a la generación procedimental de los elementos de juego. Con la la plataforma de desarrollo ya decidida se genera el código donde, como se comentó previamente, la responsabilidad de los programadores tras la generación sería la implementación de las diferentes funcionalidades. Pongamos que como plataforma decidimos usar un pequeño motor de juego hecho en C++. En ese caso, además de la lógica de juego se generan ficheros .h y .cpp para cada componente, donde en el .cpp habría un método vacío por cada mensaje que ese componente fuese capaz de ejecutar. En el caso del componente *Controller*, estarían los métodos vacíos *Turn*, *WalkTo*, *RunTo* y *Jump*, la gestión de la invocación cuando el

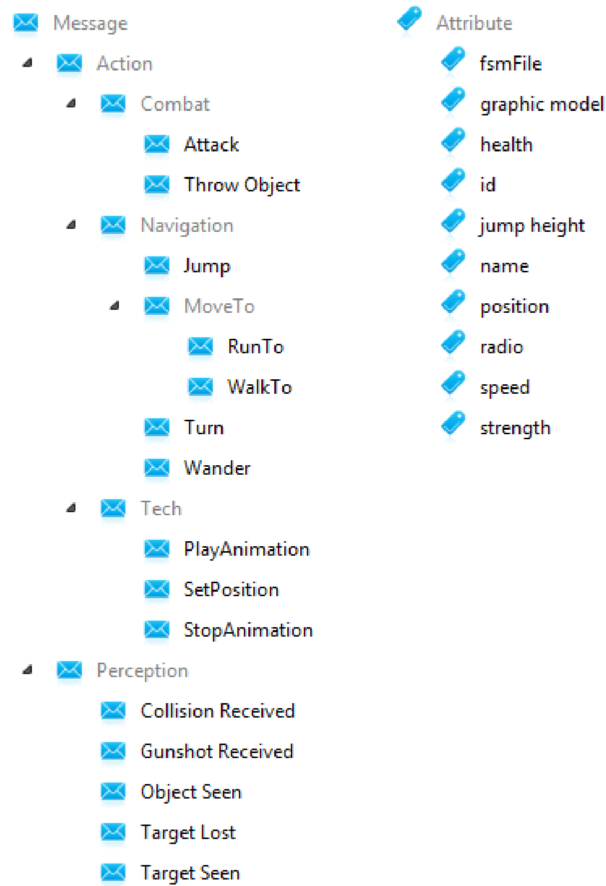

Figura 4.3: Mensajes y atributos con los que se definiría la entidad *Zombie* del *Half-Life*

componente recibe un mensaje está automáticamente implementada y lo que tendría que hacer el programador es dar las implementaciones de esos métodos. Por ejemplo, para el método *Turn* implementaría algo parecido a los que se muestra en la Figura 4.4.

El mismo dominio con el que se generaron los recursos de código para programadores genera también contenido para los diseñadores. Éstos reciben ficheros con las descripciones de las entidades del juego, los cuales se usan en el editor de mapas. Al crear instancias de las entidades el diseñador las posiciona y, opcionalmente, cambia los valores por defecto de los atributos de manera que se pueda dar variedad al juego poniendo en ocasiones *zombies* más rápidos, con más vida, con diferentes niveles fuerzas o con diferentes comportamientos. Las descripciones de las entidades también se usan para crear esos diferentes

```
class Controller : Component {
  ...
  void Turn(float delta){
     _entity.yaw += delta;
  }
  ...
};
```

Figura 4.4: Código del método *Turn* ya implementado por los programadores

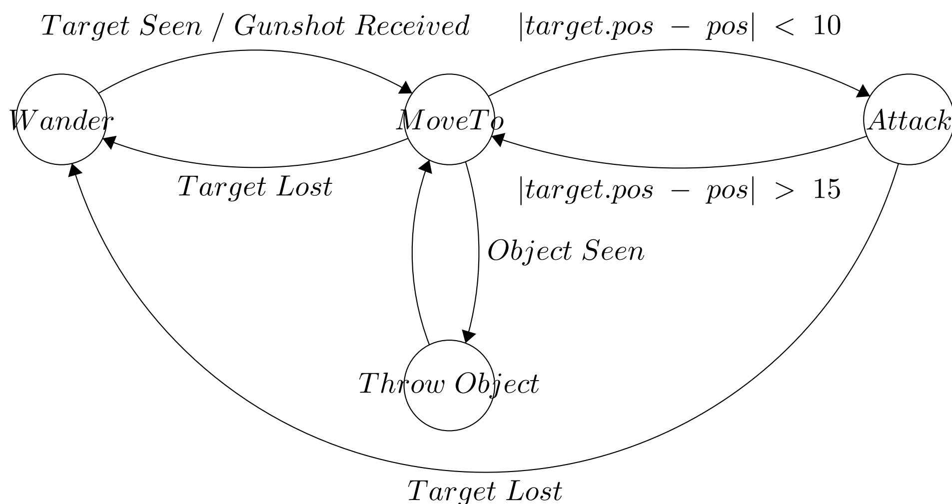

Figura 4.5: FSM simplificada del comportamiento de un *Zombie*

comportamientos de las entidades, donde podemos tener un comportamiento básico por defecto pero también comportamientos especiales o en forma de guión. Un ejemplo de comportamiento básico para un *zombie* sería la que muestra la Figura 4.5, donde el *zombie* está vagando pero si percibe al personaje va hacía él, si se encuentra un objeto se lo lanza y si está suficientemente cerca le ataca. Para modelar ese comportamiento los diseñadores deben usar las acciones y atributos que usaron para describir la entidad, ya que sino la ejecución del comportamiento fallaría en tiempo de ejecución. En el caso de esta máquina de estados (FSM), para las transiciones se usan las percepciones del *Zombie* o se consultan atributos mientras que en los estados en donde se describen las acciones que debe realizar. Si alguna de las acciones, percepciones o atributos no fuesen parte de la descripción de una entidad de juego a

la que se le asigna dicho comportamiento se debería avisar al diseñador.

Esto sería un ejemplo de una pequeña iteración de la metodología, pero mientras ésta termina podría haber otros diseñadores y programadores modificando el dominio para añadir características nuevas al juego. Características que pueden afectar tanto al código que escribieron los programadores cómo a los niveles y comportamientos que definieron los diseñadores. La generación automática debería proteger a los programadores de los cambios en el dominio y su influencia en el código generado, de manera que si el componente *Controller* que tiene el mensaje *Turn* se dividiese en dos componentes el código del método *Turn* debería ser movido al componente correspondiente, de la misma manera que si se cambia el nombre del componente o el nombre del mensaje *Turn*, se debería adaptar el código sin que afectase a la funcionalidad. En el lado de diseño habría que contemplar si los mapas antiguos son consistentes con el nuevo dominio, viendo si los atributos de una entidad y sus valores por defecto son correctos y no violan ninguna restricción y, además, ver si los comportamientos de las entidades siguen pudiéndose realizar ya que podría darse el caso que durante el diseño se decida por ejemplo que los *Zombie*s no pueden detectar ni lanzar objetos, por lo que el comportamiento desarrollado (Figura 4.5) dejaría de ser consistente. Acerca de estos posibles problemas habría que avisar a los diseñadores para que adecuen sus niveles.

4.4. Representación ontológica de los diferentes elementos

En las secciones anteriores hemos visto como los programadores y diseñadores manejan diferentes vistas de la misma información. También hemos anticipado que, por debajo de esas representaciones, la información está guardada usando Web Ontology Language (OWL) (W3C, 2012). En esta sección veremos como representamos y almacenamos todo ese conocimiento.

Las sintaxis en las que se representa habitualmente OWL no están diseñadas para simplificar su uso por humanos cuando construyen o analizan ontologías sino que para esos propósitos se suponen herramientas con una interfaz gráfica como Protégé (for Biomedical Informatics Research, 2013). En nuestra metodología se especifica que para la edición y visualización de los dominios en OWL necesitamos también una herramienta que lo represente gráficamente. En esta memoria, sin embargo, en vez de usar los ejemplos visuales, en ciertas ocasiones se usa la sin-

taxis Manchester (Horridge y Patel-schneider, 2008), una sintaxis amigable que nos permite expresar qué tipo de ediciones hacemos. De esta forma, lectores interesados en el uso concreto de OWL pueden ver qué tipo de modelo concreto hemos implementado.

Como hemos visto, tenemos cuatro grupos de elementos básicos en el modelo que son *entidades*, *componentes*, *mensajes* (que representan las acciones y percepciones) y los *atributos*. Todos estos elementos estarán representados en varias ontologías que, mediante la relación *es-un*, definen las jerarquías de los diferentes elementos. Las hojas de esas jerarquías (y alguno de los conceptos intermedios) se corresponderán en el futuro a clases o estructuras reales en el código del juego. La misión del resto de conceptos intermedios es estructurar el modelo en diferentes niveles de abstracción, por lo que sólo se necesitan en la representación formal:

- **Ontología de entidades:** Las entidades son colecciones de componentes que, para una aproximación de grano más fino, se pueblan de atributos. A estos atributos se les debe poder asignar valores por defecto, que pueden ser sobrescritos en las entidades hijas. Mantener las entidades en una ontología conceptual ayuda a programadores y diseñadores a reconocer rápidamente el propósito principal de éstas. Sin embargo, esta ontología es sólo conceptual y, aunque sería sencillo inferir de aquí una jerarquía de clases, no debe confundirse con ella. En esta representación cada entidad del juego será genérica y la diferencia entre los diferentes tipos de entidades se determina por la funcionalidad que les aportan los componentes. De hecho, las entidades no tienen por qué heredar todas las propiedades de sus padres; para mantener la flexibilidad de la arquitectura basada en componentes se pueden descartar componentes heredados en entidades hijas. Todas las entidades heredan de la entidad principal llamada *Entity* aunque, como se ha dicho, conceptualmente una entidad puede ser una especialización de otra entidad.

- **Ontología de componentes:** Cada componente de la ontología debe estar descrito por los mensajes que es capaz de llevar a cabo. A veces, hay dependencias entre el objetivo que un componente tiene (mensaje que procesa) y los subobjetivos (mensajes que genera y envía) que deben ser llevados a cabo por otros componentes para que se cumpla dicho objetivo. Además, un componente puede necesitar que la entidad a la que pertenece tenga atributos. Estos atributos también deben anotarse en la descripción del componen-

te. En este caso, la ontología podrá típicamente coincidir con la implementación de éstos en el código ya que un comportamiento de un componente puede especializar a otro. En esta ontología todos los componentes heredan del componente principal llamado *Component*.

- **Ontología de mensajes:** Un mensaje representa un requerimiento de funcionalidad o un evento que se ha producido y del que se espera reacción. Son enviados a las entidades y capturados por sus componentes y, para potenciar su descripción pueden ser parametrizados por medio de atributos. Como en las entidades, esta ontología suele ser conceptual y sólo los conceptos marcados como tal deberían ser implementados. El mensaje principal se llama *Message* y de él heredan *Action* y *Sense*, de donde los diseñadores (y luego los programadores) hacen que hereden sus mensajes.

- **Ontología de atributos:** Los atributos se describen por un nombre y el tipo de valores que puede tomar (enteros, reales, cadenas, vectores, etc.) a los que además se les pueden añadir restricciones que definan un conjunto finito de valores que puede tomar o un intervalo de estos. Además, puede haber atributos complejos como tipos de entidad, de componente o mensaje. El atributo principal se llama *Attribute* y toda esta definición es muy útil para chequear la consistencia del dominio.

La entidad de entidades se organizan como una *jerarquía de clases* de OWL (del inglés *class hierarchy*), donde todas las entidades son subclases de una entidad común y la jerarquía será conceptual. Las ontologías de componentes y mensajes se representan también en OWL como *jerarquías de clases* mientras que los atributos son un caso especial. Recordamos que en los atributos se podía definir su tipo como un tipos básico (enteros, reales, cadenas...), a los cuales llamábamos tipos simples, o como otro elemento del dominio (mensajes, componentes o entidades), llamados atributos complejos. Los atributos complejos también se representan como *jerarquía de clases* mientras que los atributos de tipos simples se representan como una jerarquía de *propiedades de tipos de datos* (del inglés *data properties*).

Para definir las relaciones entre elementos de las jerarquías usamos *propiedades de objeto* (del inglés *object properties*) como las que se definen a continuación:

- **isA(?e, ?p)**: Las entidades se pueden relacionar en una jerarquía conceptual. *isA* define que la entidad *?e* es hija de *?p* en esa je-

rarquía conceptual. No usamos subclases en este caso ya que las entidades hijas no tienen por qué heredar todas las propiedades de las entidades padre.

- **hasComponent(?e, ?c)**: Las entidades están compuestas por componentes por lo que esta propiedad indica que la entidad *?e* tiene como parte de su descripción el componente *?c*.
- **isComponentOf(?c, ?e)**: Es la propiedad inversa de *hasComponent* por lo que isComponentOf(?c, ?e) si y solo si hasComponent(?e, ?c).
- **interpretMessage(?x, ?m)**: Los componentes son los encargados de procesar los mensajes que recibe la entidad en la implementación del juego pero, conceptualmente, las entidades también son capaces de interpretar mensajes (que luego delegarán a los componentes). Por tanto esta propiedad indica que el componente o entidad *?x* es capaz de procesar el mensaje *?m*.
- **isMessageInterpretedBy(?m, ?x)**: es la propiedad inversa de *interpretMessage*.
- **hasAttribute(?x, ?a)**: Las entidades están descritas en función de sus atributos, pero también los componentes. Esta propiedad indica que el componente o entidad *?x* está descrita por el atributo *?a*.
- **isAttributeOf(?a, ?x)**: Es la propiedad inversa de *hasAttribute*.
- **isInstantiable(?x)**: Indica que *?x* es instanciable y qué por tanto estará representada en el juego final y los diseñadores podrán usarlos en la edición de niveles.
- **isNotInstantiable(?x)**: Es la propiedad opuesta de *isInstantiable* y describe a aquellos conceptos que sólo tienen sentido desde el punto de vista conceptual para estructurar el modelo.

Teniendo en cuenta estas propiedades podemos describir los elementos del juego. Una entidad se define de la siguiente manera:

```
Class: <EntityName>
  SubclassOf: Entity
    [and isA <ParentEntityName>]+
    [and (hasComponent some <ComponentName>)]*
    [and (interpretMessage some <MessageName>)]*
    [and (hasAttribute some <ComplexAttributeName>)]*
    [and (<SimpleAttributeName> some <RDFDatatype>)]*
    and (isInstantiable) | (isNotInstantiable)

(+) implica que debe haber una o más apariciones de dicha
    propiedad
(*) implica que puede haber cero o más apariciones de dicha
    propiedad
```

Una entidad es por tanto una clase que hereda de *Entity* (*es un Entity*) pero a su vez, conceptualmente tiene como padres una o más entidades de la jerarquía. Puede estar descrita por cero o varios componentes, interpretar mensajes y estará descrita también (aunque no necesariamente) por atributos complejos, o por atributos de tipos básicos (RDF Datatypes). Por último las entidades podrán ser instanciables, si tendrán representación en el juego, o no instanciables, si se usan simplemente para dar mayor coherencia al dominio.

Por otro lado, los componentes se definen como:

```
Class: <ComponentName>
  SubclassOf: <ParentComponentName>
    [and (interpretMessage some <MessageName>)]*
    [and (isComponentOf only
           (hasComponent some
             (interpretMessage some <MessageName2>)))]*
    [and (hasAttribute some <ComplexAttributeName>)]*
    [and (<SimpleAttributeName> some <RDFDatatype>)]*
    and (isInstantiable) | (isNotInstantiable)
```

Como se ve, los componentes también están representados por clases que son subclases de la clase *Component*. Están descritos por los mensajes que pueden interpretar y éstos pueden opcionalmente requerir de ayuda externa para poder llevar a cabo esos mensajes. Para poder procesar un mensaje, un componente puede necesitar delegar parte

del procesado en otros componentes que sepan interpretar mensajes adicionales. Esta delegación es modelada con las propiedades *isComponentOf*, *hasComponent* y *interpretMessage*. Por otro lado, igual que las entidades, los componentes están descritos por los atributos simples y complejos que los describen y se indica si son o no son instanciables.

Los mensajes son descritos como se detalla a continuación:

```
Class: <MessageName>
  SubclassOf: <ParentMessageName>
    [and (hasAttribute some <ComplexAttributeName>)]*
    [and (<SimpleAttributeName> some <RDFDatatype>)]*
    and (isInstantiable) | (isNotInstantiable)
```

Estos mensajes, representados en la ontología mediante clases, heredan de *Message* y pueden transportar parámetros que también son modelizados en la ontología. Esos parámetros pueden ser igual que en el caso de los atributos de las entidades y componentes, simples o complejos. Los mensajes pueden ser instanciables o no instanciables en función de si serán representados en la implementación del juego o si sólo tienen un valor semántico.

Los tipos de atributos complejos se describen como:

```
Class: <ComplexAttributeName>
  SubclassOf: <ParentComplexAttributeName>
  EquivalentTo: <EntityComponentOrMessageName>
```

Los cuales heredan de la clase *Attribute* y son clases equivalentes a los tipos de datos que representan. Por último, los atributos de tipos básicos se pueden definir normalmente sin ningún tipo de restricción:

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype
```

Con restricciones de rango para tipos contables:

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype[>= 0f, <1f]
```

O con restricciones que determinen que conjunto de valores puede tomar el atributo:

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype and {<value,>*}
```

Este tipo de atributos son *propiedades de tipos de datos* y heredan de *BasicAttribute*, que es el elemento superior de la jerarquía.

Por último, en la ontología guardamos entidades prototípicas de cada entidad, dando valores por defecto a sus atributos. Para eso, se crea un individuo por cada clase del dominio y, por convenio, cada individuo se llama igual que la clase del tipo del individuo pero precedido de una “i”. La representación del individuo de una entidad es la siguiente:

```
Individual: <iEntityName>
  Types: <EntityName>
  Facts: [isA <iEntityParentName>,]+
         [hasComponent <iComponentName>,]*
         [interpretMessage <iMessageName>,]*
         [hasAttribute <iComplexAttributeName>,]*
         [<SimpleAttributeName> <AttrivuteValue>,]*
```

Como es lógico, para que el dominio sea consistente, el individuo de una entidad tiene que estar definido por individuos de los mismos tipos que se especifica en la descripción de la entidad tipo (debe tener los mismos componentes, atributos, etc.) y asignar valores a los atributos descritos. Para poder describir los individuos de las entidades, se deben definir individuos de componentes, mensajes y atributos complejos ya que las *propiedades de objetos* usadas en la definición de las entidades deben asociarse a individuos de componentes, mensajes y atributos complejos.

4.4.1. Ejemplo

Basándonos en el ejemplo de la Sección 4.3.2, vamos a ver como serían las descripciones de algunos de los elementos de ese juego descritos en las Figuras 4.2 y 4.3. Empezando por el *Zombie*, que sería la entidad principal que se añadiría en esta iteración. La representación es auto-contenida y no necesita mayor explicación:

```
Class: Zombie
  SubclassOf: Entity
    and isA Enemy
    and (hasComponent some Graphic)
    and (hasComponent some Physics)
    and (hasComponent some Controller)
    and (hasComponent some Health)
    and (hasComponent some Combat)
    and (hasComponent some Wanderer)
    and (hasComponent some Perception)
    and (hasComponent some AI)
    and (interpretMessage some Attack)
    and (interpretMessage some ThrowObject)
    and (interpretMessage some Jump)
    and (interpretMessage some RunTo)
    and (interpretMessage some WalkTo)
    and (interpretMessage some Turn)
    and (interpretMessage some Wander)
    and (interpretMessage some PlayAnimation)
    and (interpretMessage some StopAnimation)
    and (interpretMessage some SetPosition)
    and (interpretMessage some CollisionReceived)
    and (interpretMessage some GunshotReceived)
    and (interpretMessage some ObjectSeen)
    and (interpretMessage some TargetSeen)
    and (interpretMessage some TargetLost)
    and (hasAttribute some target)
    and (fsmFile some string)
    and (graphicModel some string)
    and (physicsType some string and
                        {"static","dynamic","kinematic"})
    and (health some unsignedint)
    and (id some unsignedint)
    and (jumpHeight some float[>= 0f, <3f])
    and (name some string)
    and (radio some float[>= 0f])
    and (strenght some float[>= 0f, <100f])
    and (speed some float[>= 0f, <5f])
    and (isNotInstantiable)
```

Uno de los componentes a los que se hicieron mención en el ejemplo sería el componente *Controller*:

```
Class: Controller
  SubclassOf: Component
    and (interpretMessage some Jump)
    and (interpretMessage some RunTo)
    and (interpretMessage some MoveTo)
    and (interpretMessage some Turn)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some PlayAnimation)))
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some StopAnimation)))
    and (speed some float[>= 0f, <5f])
    and (isInstantiable)
```

Donde podemos ver que para poder realizar las acciones de navegación indicadas por mensajes deberá haber otro componente en la entidad capaz de ejecutar los mensajes de reproducción y parada de animaciones. La definición de un mensaje sencillo podría ser el de giro, que tiene como parámetro cuánto se debe girar:

```
Class: Turn
  SubclassOf: Navigation
    and (delta some float)
    and (isInstantiable)
```

Para poder asociar un objetivo al que perseguir tendremos un atributo *target* de tipo *Player* y que se representa:

```
Class: target
  SubclassOf: Attribute
  EquivalentTo: Player
```

En cuanto a los tipos básicos ya hemos visto en las descripciones de la entidad *Zombie* y del componente *Controller* como pueden tener limitaciones de rango. Por ejemplo, el atributo que controla la velocidad no querríamos tener una velocidad negativa obviamente, pero además queremos controlar y que el juego quede realista por lo que a su vez no permitiremos que las entidades que usen dicho atributo vayan más rápido de 5 m/s:

```
DataProperty: speed
  SubPropertyOf: BasicAttribute
  Range: float[>= 0f, <5f]
```

Por otro lado, el atributo que declara el tipo de física que usa la entidad es un string que solo puede tomar tres valores diferentes que definan si la entidad es *estática, dinámica* o *cinemática*:

```
DataProperty: physicsType
  SubPropertyOf: BasicAttribute
  Range: string and {"static","dynamic","kinematic"}
```

Por último, la descripción de la instancia prototípica del *zombie* sería:

```
Individual: iZombie
  Types: Entity
  Facts: isA iZombie,
         hasComponent iGraphic,
         hasComponent iPhysics,
         ...
         interpretMessage iAttack,
         interpretMessage iThrowObject,
         ...
         hasAttribute itarget,
         fsmFile "basicZombieBehaviour.xml",
         graphicModel "zombie.mesh",
         health "150"^^unsignedInt,
         id "0"^^unsignedInt,
         jumpHeight 0.5f,
         name "zombie",
         radio 1f,
         strenght 20f,
         speed 1.5f
```

Como se ve, la descripción necesita la existencia de individuos de los componentes que usa (*iGraphic* o *iPhysics*), de los mensajes (*iAttack* o *iThrowObject*) y de los atributos (*itarget*).

Todo el trabajo relatado en el resto de la tesis se basa en estas descripciones. Con ellas se ha podido inferir gran cantidad de conocimiento para aplicación de técnicas específicas de la metodología (Capítulo 5) y han sido la base de una completa herramienta que soporta la metodología aquí contada (Capítulo 6).

Capítulo 5

Técnicas de soporte a la metodología

En este capítulo se presentan tres técnicas específicas que se apoyan en el dominio formal para sustentar la metodología propuesta. En la Sección 5.1 se presenta una técnica que infiere, a partir de la ontología y las descripciones de las entidades, una distribución de funcionalidad en componentes que maximiza su cohesión y minimiza el acoplamiento. En la Sección 5.2 se explica cómo se realiza la generación de contenido de juego a partir del dominio y cómo eso mejora el desarrollo gracias no solo al contenido generado sino también a la posibilidad de refactorizar a alto nivel. Por último, la Sección 5.3 describe el uso del dominio formal para poder comprobar, en tiempo de edición, si el dominio, o los niveles editados, tienen posibles inconsistencias que puedan provocar errores durante la ejecución del juego.

5.1. Identificación automática de componentes software

5.1.1. Introducción

La arquitectura basada en componentes (Sección 2.4.2) promueve la flexibilidad, reusabilidad y facilita la extensibilidad de código pero al mismo tiempo hace el código más difícil de entender ya que, con esta arquitectura, la funcionalidad y el comportamiento de las entidades están distribuidos en diferentes componentes software que se comunican entre sí, pero que sólo se unen en tiempo de ejecución. La arquitectura de componentes provoca, por tanto, problemas en la comprensión de las entidades debido a la pérdida de la distribución jerárquica de las mis-

mas (Capítulo 4). Esa distribución permitía entender el conjunto de las entidades de juego de un vistazo pero, sin embargo, la distribución de la funcionalidad de una entidad mediante dichos componentes software no resulta sencilla de entender ni realizar.

En esta sección nos concentraremos en el proceso no trivial de, partiendo de una descripción jerárquica de entidades como las comentadas en el Capítulo 4, extraer una distribución de funcionalidad implementada en componentes software. Para poder realizar este proceso aplicamos una novedosa técnica que identifica la mejor distribución de componentes entre las entidades descritas, ayudándose de análisis formal de conceptos (FCA por sus siglas en inglés, Formal Concept Analysis) (Ganter y Wille, 1997), un método matemático para el análisis de datos.

En primer lugar, en la Sección 5.1.2 se introduce FCA para poder explicar en las siguientes secciones nuestra técnica. En la Sección 5.1.3 se explica cuál es el proceso que se sigue para transformar la ontología de entidades creada por los diseñadores y programadores en una entrada válida para la aplicación de FCA (un contexto formal). A continuación se describe la técnica básica para la extracción de componentes software a partir del retículo extraído. Ésta consta de dos fases: la primera de ellas (Sección 5.1.4) es un proceso automático que presenta un conjunto candidato de componentes al programador mientras que en la segunda fase (Sección 5.1.5) se le ofrecen al experto ciertos mecanismos para poder alterar la distribución de funcionalidad en componentes automáticamente generada. Tras ésto, en la Sección 5.1.6 se pone un ejemplo práctico de la técnica explicada.

En todas esas secciones se trabaja con un modelo de desarrollo en cascada, más sencillo pero poco realista en el desarrollo de videojuegos. Por tanto, en la siguiente sección (Sección 5.1.5) se adapta la técnica propuesta para que soporte esa iteratividad y esos cambios en la definición del dominio sin provocar una pérdida de todo lo que se ha hecho en iteraciones previas. Se describe también un ejemplo práctico de la técnica en su modo iterativo (Sección 5.1.8) y, por último, el la Sección 5.1.9, se evalúa el trabajo realizado.

5.1.2. Análisis formal de conceptos

El análisis formal de conceptos o FCA (de su nombre en inglés *Formal Concept Analisys*) es un método matemático para el análisis de datos, la representación de conocimiento y gestión de la información. El término fue introducido por Rudolf Wille (Wille, 1982) a principio de los años 80 y durante las últimas tres décadas ha sido usado en diferentes domi-

nios como la psicología, la medicina, la lingüística o la informática entre otros. En general, FCA puede ser útil en inferencia de conocimiento en bases de datos (Stumme, 2002).

FCA se aplica a cualquier colección de elementos (u *objetos formales* de acuerdo con la nomenclatura de FCA) descritos por un conjunto de propiedades (o *atributos formales*). Cuando se aplica FCA sobre un conjunto de objetos, la información se estructura y agrupa en abstracciones formales llamadas *conceptos formales*. Estos *conceptos formales* pueden ser vistos como un conjunto de objetos que comparten un conjunto común de atributos y, por tanto, el resultado de la aplicación de FCA sobre una colección de elementos proporciona una vista interna de la estructura conceptual y permite reconocer patrones, uniformidades y excepciones en ese conjunto de elementos. Además, los *conceptos formales* pueden ser ordenados usando las relaciones de subconcepto y superconcepto, donde el conjunto de objetos que se encuentran en los subconceptos son subconjuntos de los objetos que se encuentran en los superconceptos y, de manera recíproca, el conjunto de atributos pertenecientes a los subconceptos son superconjuntos de los atributos que se encuentran en los subconceptos.

Los elementos y atributos se proporcionan a la técnica de FCA en forma de *contexto formal*. Este *contexto formal* se define como una tripla $\langle G, M, I \rangle$ donde G es el conjunto de *objetos formales*, M el de *atributos formales* e $I \subseteq G$xM es una relación binaria que expresa qué atributos describen cada objeto (o qué objetos son descritos mediante un atributo). Por tanto, $(g, m) \in I$ si el objeto g tiene el atributo m, o m es un descriptor del objeto g. Cuando el conjunto es finito, el contexto puede ser especificado a través de una tabla cruzada, donde las filas representan los objetos mientras que las columnas representan los atributos. Un celda cualquiera se marca cuando el objeto de esa fila tiene el atributo de esa columna.

Un *concepto formal* se puede definir mediante una *extensión* (del inglés extent) y una *intensión* (intent). Se denomina *extensión* de un concepto al conjunto formado por todos los objetos que caen bajo el concepto mientras que la *intensión* consta de todas las propiedades que son comunes a todos los objetos que caen bajo el concepto. Formalmente, se define el operador *prima* de manera que cuando se aplica sobre un conjunto de objetos $A \subseteq G$ devuelve los atributos que son comunes a todos ellos:

$$A' = \{m \in M \mid (\forall g \in A)(g, m) \in I\}$$

Y, cuando se aplica sobre un conjunto de atributos $B \subseteq M$, el resul-

tado es el conjunto de objetos que tiene dichos atributos:

$$B' = \{g \in G \mid (\forall m \in B)(g, m) \in I\}$$

Con esta definición, podemos decir que un par (A, B), donde $A \subseteq G$ y $B \subseteq M$, se denomina *concepto formal* del contexto $\langle G, M, I\rangle$ si $A' = B$ y $B' = A$. En esta definición A y B hacen referencia a la *extensión* y la *intensión* del *concepto formal*, respectivamente.

El conjunto de todos los conceptos formales de un contexto $\langle G, M, I\rangle$ se indica como $\beta(G, M, I)$ y es la salida del FCA. La estructura más importante sobre $\beta(G, M, I)$ es dada por la relación subconcento - superconcepto que se denota por $\leq$ y es definida como: $(A1, B1) \leq (A2, B2)$ donde $A1$, $A2$, $B1$ y $B2$ son conceptos formales y $A1 \subseteq A2$ (que es equivalente a $B2 \subseteq B1$ (Ganter y Wille, 1997)). Esta relación toma la forma de un *retículo de conceptos* donde los nodos del retículo representan conceptos formales y las líneas que los unen simbolizan la relación de subconcepto - superconcepto. El retículo tiene dos nodos especiales: el supremo y el ínfimo que representan a los conceptos de más arriba y de más abajo del retículo. Mientras que el supremo ($\top$) tiene todos los objetos formales en su extensión (y, en ocasiones, uno o unos pocos atributos en su intensión), el ínfimo ($\perp$) tiene todos los atributos en su intensión (y quizá algún objeto formal en su extensión).

Un retículo de conceptos puede ser dibujado para simplificar el análisis realizado por humanos, donde el supremo aparece en la parte superior, por encima de todos sus subconceptos, que cuelgan de él y terminan en el ínfimo (Figure 5.2). Esta representación usa los llamados *intensión reducida* y *extensión reducida*. La *extensión reducida* de un concepto formal es el conjunto de objetos que pertenecen a la extensión de dicho concepto formal y no pertenecen a la extensión de ninguno de sus superconceptos. Por otro lado, la *intensión reducida* se compone de los atributos de la intensión que no pertenecen a ninguno de sus superconceptos. En ese sentido, para rescatar la extensión de un concepto formal se necesita seguir la pista de todos los caminos que van desde el concepto en cuestión hasta el ínfimo añadiendo todos los objetos formales de cada concepto formal en el camino. Por el contrario, si se añaden todos los atributos formales de la intensión reducida de los conceptos que hay desde el concepto concreto hasta el supremo obtenemos la intensión.

5.1.3. Metodología para la obtención del retículo

Según la metodología explicada en el Capítulo 4, el primer paso para comenzar la creación del dominio semántico del juego es que los dise-

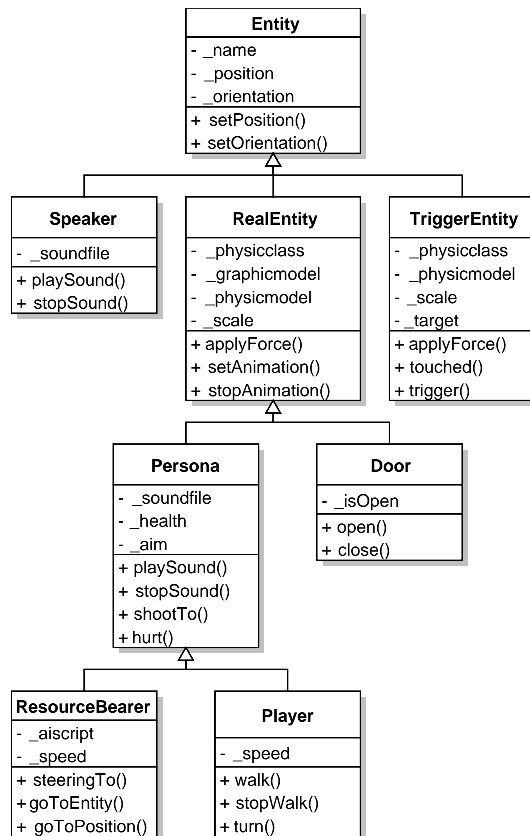

Figura 5.1: Descripción ontológica de las entidades de juego

ñadores especifiquen la ontología de entidades de juego. Dichos diseñadores deben también proporcionar a las entidades un estado y una funcionalidad en forma de atributos y acciones. Por ejemplo pueden definir un *jugador* que pueda *girar* y *caminar* a cierta *velocidad* mientras que una *puerta* puede *abrirse* o *cerrarse* y a su vez guardar el estado de si *está abierta* (Figura 5.1). Cuando los programadores recogen el dominio creado por los diseñadores añaden detalles propios de la implementación, como que las entidades puedan tener una *escala* o una *posición*, y después de esto su misión es, a partir de esa descripción, generar un conjunto de componentes software entre los que se reparta toda la funcionalidad y estado (acciones y atributos) que hay en las diferentes entidades. Ese es justo el problema que queremos abordar: cómo inferir automáticamente la mejor distribución de acciones y atributos en componentes que maximicen su cohesión y minimicen el acoplamiento.

El primero de los pasos será transformar la ontología de entidades en un *contexto formal* con la intención de aplicar FCA. Los tipos de en-

	setPosition	setAnimation	touched	_physicsclass	_aiscript	⋰
Entity	■					...
Trigger	■		■	■		...
Persona	■	■		■		...
ResourceBearer	■	■		■	■	...

Tabla 5.1: Parte del *contexto formal* del dominio de juego

tidades como el *jugador* o la *puerta* se convierten en *objetos formales* y las diferentes características como acciones de *andar*, *girar*, *abrirse*, etc. y atributos tipo *velocidad*, *escala*, etc. pasan a ser *atributos formales*. Por tanto, nuestro *contexto formal* $\langle G, M, I \rangle$ se construye de manera que G contiene a todos los tipos de entidad, mientras que M tiene todas las acciones y atributos que se usan en la descripción de entidades. Finalmente, I es la relación binaria $I \subseteq G\text{x}M$ que expresa qué atributos de M describen cada objeto de G o qué objetos son descritos usando atributos. Para rellenar el contexto formal se recorre la ontología de entidades, anotando la relación entre los tipos de entidad y sus relaciones con acciones y atributos, teniendo en cuenta las relaciones *es un* de la ontología. Estas relaciones pueden verse como un tipo de herencia, donde las subclases incluirían todos los *atributos formales* de sus superconceptos. Dado que un *jugador es un personaje*, el *jugador*, además de *andar* o *girar*, será capaz de *disparar* o de *sufrir daño* como los *personaje*s. En la Tabla 5.1 se muestra una vista parcial del *contexto formal* extraído de la ontología mostrada en la Figura 5.1.

Si aplicamos las técnicas de FCA sobre este *contexto formal*, lo que obtenemos como resultado de este proceso es un *retículo* (Figura 5.2) que se compone de un conjunto de conceptos y sus relaciones: $\beta(G, M, I)$. Cada *concepto formal* representa a todos los tipos de entidades que forman parte de su *extensión* ya que requieren de todas las acciones y atributos que aparecen en su *intensión*. Empezando con el *retículo*, y con el objetivo de extrapolar los *conceptos formales* a una abstracción de programación, una solución inocente es generar una jerarquía de clases para usar la aproximación arquitectónica de la Sección 2.4.2.1. Desafortunadamente, el resultado es una jerarquía que hace uso extensivo de la herencia múltiple, lo cual es a menudo considerado como indeseable debido a su complejidad. En la siguiente sección se muestra que extrac-

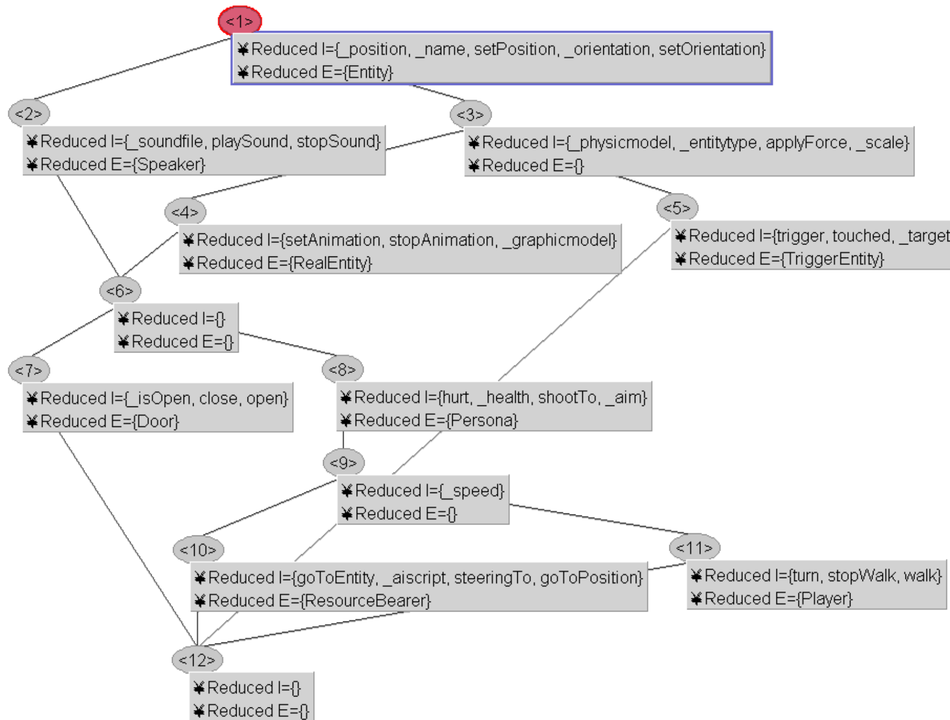

Figura 5.2: Retículo de conceptos

ción de conocimiento proponemos, más orientada a la composición en lugar de a la herencia.

5.1.4. Infiriendo componentes del retículo

Por tanto, aunque la aproximación de convertir conceptos formales en clases ha sido satisfactoriamente llevado a cabo por otros (Hesse y Tilley, 2005; Godin y Valtchev, 2005), no es válido en nuestro contexto. Al fin de cuentas, nuestro objetivo final es eliminar la herencia y identificar *características comunes en las entidades* con el objetivo de crear una clase independiente (un componente) por cada conjunto de características que se encuentran siempre juntas. Una vez tengamos esas identificaciones hechas, se creará una simple y genérica clase *entidad* que mantendrá una lista de componentes, como promueve la arquitectura basada en componentes. La adición de nuevas características se hace mediante la adición de componentes a la entidad, de manera que esos componentes son independientes unos de otros. La consecuencia de esta independencia es que, ahora, compartir la implementación de una misma característica en entidades diferentes no está cableado en el código, sino que las diferentes características se componen en ejecución, evi-

tando así problemas comunes que se producen a largo plazo jerarquías de clases en poco flexibles.

Usando FCA, podemos alcanzar nuestro objetivo si en vez de fijarnos en los objetos (extensión reducida de los conceptos formales) nos fijamos *en los atributos* (intensión reducida). La idea se basa en el hecho de que cuando un concepto formal tiene una intensión reducida no vacío significa que el concepto contribuye al retículo añadiendo algunos atributos y/o acciones que no habrían aparecido antes (si hacemos un recorrido de el retículo de arriba a abajo). Por tanto, esos conceptos formales que tienen una intensión reducida no vacía dan fruto a un nuevo componente, que tendrá los atributos que se encuentren en esa intensión reducida y que tendrá la habilidad de llevar a cabo las acciones que allí también se encuentren. Al mismo tiempo se puede inferir que todos los tipos de entidad que se encuentren en la extensión de ese concepto formal necesitarán ese estado y esa funcionalidad, por lo que incluirán en su descripción el nuevo componente creado.

Por ejemplo, cuando se analiza el concepto formal número *11* de la Figura 5.2, nuestra técnica extraerá un nuevo componente que contendrá las características *turn*, *stopWalk* y *walk*, y anotará que todas las instancias del tipo de entidad *Player* necesitarán incluir un componente de este tipo (y todos los componentes extraídos de los conceptos formales que se encuentran sobre él, como se explica más adelante).

Presentado de una manera más formal, el proceso general parte de la ontología O, creada por los expertos, y acaba con una colección de componentes y entidades que los contienen. El proceso es el siguiente:

```
G = tipo_de_entidades(O)
M = atributos_y_acciones(O)
I = construir_relaciones(O)
L = β(G,M,I)
P = lista_vacia_de_componentes
for each concepto formal C in L
  B_r = intensión_reducida(C)
  if B_r is empty then continue
  X = componente(B_r)
  add(P, X)
  A = extensión(C)
  for each objeto Y in A
    add(X, Y)
  end for
end for
```

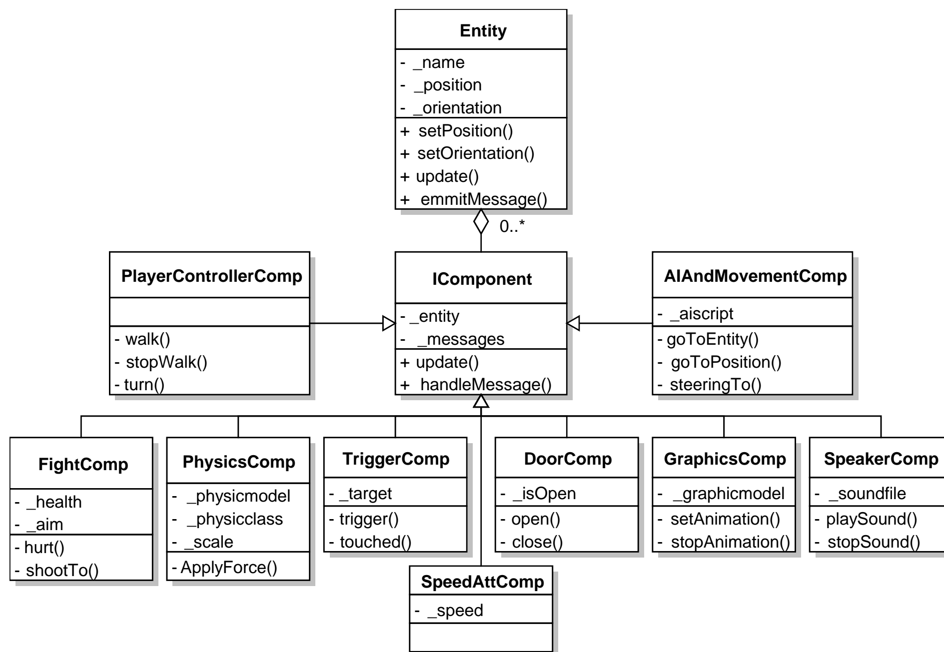

Figura 5.3: Conjunto de componentes obtenidos de la aplicación de nuestra técnica

Todas las líneas anteriores son explicativas por si mismas excepto la línea X = *componente(* B_r *)*. La función *componente* recibe la intensión reducida del concepto formal y construye el componente en si con sus atributos y acciones.

En algunos casos el concepto supremo ($\top$) tiene una intensión reducida no vacía, por lo que ese concepto también genera un componente con todas las características que haya en la intensión reducida. Por ejemplo, en la Figura 5.2, $\top$ tiene los atributos *name,position* y *orientation*. Ese componente se añadiría en *todos los tipos de entidad* así que, en vez mantenernos en un arquitectura basada en componentes pura con una clase entidad vacía, podríamos optar por mover todas esas características de $\top$ a la entidad. Esta decisión depende del tipo de implementación que se quiera tomar, mientras que el resto de componentes siempre se generan.

La Figura 5.3 muestra los componentes extraídos por esta técnica usando el retículo de la Figura 5.2. Los componentes generados son renombrados automáticamente concatenando los nombres de sus atributos y, cuando no hay atributos en el componente, concatenando el nombre de las acciones que es capaz de llevar a cabo. Por ejemplo, el nombre original del componente *FightComp* era *C_health_aim*.

Resumiendo todo el proceso, cuando se analiza el retículo de conceptos, todo concepto formal que proporciona una nueva característica (teniendo una intensión reducida no vacía) no representa una nueva entidad sino un nuevo componente. La única excepción podría ser el concepto formal supremo $\top$, que representaría las características que tendría la clase genérica de la entidad, la cual contiene acciones y atributos que comparten todos los tipos de entidad. De esta manera, obtenemos un conjunto de componentes candidatos, los cuales deben ser añadidos a las entidades que se encuentran en la extensión del concepto formal del que proceden.

De esta forma, podemos fácilmente obtener un conjunto de componentes candidatos junto con su clase entidad genérica, pero aun tenemos que describir los tipos de entidades en función de dichos componentes. Cómo se ha comentado anteriormente, los tipos de entidades a los que pertenecerá componente, creado a partir de un concepto formal, son aquellos que aparecen en la extensión de dicho concepto formal. Sin embargo, en la Figura 5.2 solo se denota la extensión reducida por lo que, para gráficamente ver que tipo de entidades pertenecen, hay que recorrer el retículo. Empezando por cada concepto cuya extensión reducida contenga un tipo de entidad, se usa la relación de superconcepto para recorrer ascendentemente hasta llegar a $\top$. En este recorrido habría que añadir todos los componentes creados a partir de los conceptos formales por los que se va pasa. Por ejemplo, el tipo de entidad *Persona* estaría formada por los componentes creados a partir de los conceptos formales *8*, *4*, *3* y *2* (el número *6* tiene una *intensión reducida* vacía por lo que no generará ningún componente). Por otro lado, el tipo de entidad *ResourceBearer* tendrá los mismos componentes pero además los generados a partir de los conceptos *10* y *9*. Además, dependiendo de nuestro tipo de arquitectura de componentes, ambos tipos de entidades tendrán el componente creado a partir del concepto número *1* o los atributos de éste pasarán a ser parte de la entidad genérica contenedora de componentes.

Hay que tener en cuenta que la representación ontológica obtenida no es equivalente al retículo ya que no se almacena la relación de subconcepto - superconcepto. No obstante esta información es también almacenada por otro lado usando OWL[1] lo que nos proporcionará funcionalidad extra como se explica en la siguiente sección.

[1] `http://www.w3.org/TR/owl-features/`

5.1.5. Refinamiento experto

El proceso automático detallado en la anterior sección termina con una colección de componentes propuestos que tienen un nombre generado automáticamente y, en función de las decisiones tomadas, una entidad que está vacía o que engloba características comunes a todas las entidades. Este resultado se presenta a los programadores, junto con la posibilidad de modificar la distribución propuesta en función de su experiencia previa. Alguno de los cambios afectarán al retículo que está por debajo (que no se presenta *nunca* a los usuarios) de manera que, la relación entre este retículo y el contexto formal inicial extraído de la ontología, se romperá. En esta etapa del proceso, esto no supone un problema ya que no volveremos a usar FCA sobre el dominio. Por otro lado, los cambios pueden ser tan grandes que el retículo pueda no volver a ser un retículo válido (según la definición matemática del retículo (Ganter y Wille, 1997)). Afortunadamente, en nuestro dominio, la información ya no se almacena en un retículo sino que está almacenada en OWL, que puede usarse para representar estructuras más ricas que meros conjuntos parcialmente ordenados. En cualquier caso, por simplicidad, durante el resto de las secciones seguiremos asumiendo que hay un retículo que modificamos, aunque internamente no lo estemos usando directamente.

Los usuarios podrán por tanto realizar cambios sobre la distribución de componentes propuesta. Cada cambio puede ser visto como un operador de transformación. Soportamos los cinco siguientes:

1. **Renombrar**: Los componentes reciben un nombre automático de acuerdo a sus atributos o acciones. Este operador nos permite dar nombres más representativos a los componentes.

2. **Partir**: En algunos casos, dos funcionalidades que no están relacionadas entre si pueden acabar en el mismo componente debido a la definición de tipos de entidades (FCA no tiene conocimiento semántico así que agrupará dos funcionalidades cuando ambas aparezcan siempre juntas en cada tipo de entidad de la ontología). En este caso, se permite a los programadores la oportunidad de repartir la funcionalidad del componente en dos nuevos componentes. Será el usuario el que decida qué atributos y qué funciones permanecerán en el componente original y cuáles se moverán al nuevo componente (al cual deberán dar un nombre). Un ejemplo, tomando como referencia la Figura 5.2, podría ser la división del nodo *9* (*FightComp*) de manera que *_health* y *hurt()* se moviesen a un nuevo componente mientras que *_aim* y *shootTo* se quedarían en el componente actual (*FightComp*).

Describiendo el operador de una manera más formal, éste modificará el retículo que se encuentra por debajo creando dos nuevos conceptos $(A1, B1)$ y $(A2, B2)$ que tendrán los mismos subconceptos y superconceptos que el concepto formal original (A, B) donde $A \equiv A1 \equiv A2$ y $B \equiv B1 \cup B2$. El concepto original es obviamente destruido. Aunque esto no es correcto matemáticamente hablando, ya que debido a estas operaciones ya no tendríamos conceptos formales, seguimos usando el término en éste y en los siguientes operadores por simplicidad.

3. **Mover características**: Éste sería el operador opuesto a *partir*. Algunas veces ciertas características recaen en diferentes componentes pero los expertos pueden querer que pertenezcan al mismo componente. En este contexto, las características de un concepto (algunos elementos de la intensión reducida) pueden ser transferidos a un componente diferente. En el retículo, esto significa que algunos atributos se mueven de un nodo a otro. Cuando este movimiento de características se hace de arriba abajo en el retículo (por ejemplo desde el nodo *9* al nodo *10*) hay que tener cuidado ya que se podrían generar inconsistencias de las que se debería informar al programador (las entidades extraídas del nodo *11* acabarían perdiendo características). Esto es debido a que al hacer este movimiento, puede haber entidades en el concepto de partida que acabasen sin esas características (ya que fueron eliminadas). En este caso se sugiere al usuario que mantenga las características en los dos lugares. Se puede también mover características a más de un componente, de manera que estas se clonen en los componentes de destino.

   Si los programadores mueven todas las características de un componente, resulta un componente vacío que debe ser por tanto eliminado del sistema. La aplicación de este operador también modifica el retículo. Siendo F las características movidas y (A, B) el concepto formal donde $F \subset B$, Cada subconcepto directo $(A2, B2)$ de (A, B) se transforma en $(A2, B2 \cup F)$ y el concepto (A, B) se convierte en $(A, B - (B \cap F))$.

4. **Añadir características**: A veces puede ser necesario añadir algunas características a los componentes. Cuando se aplica FCA, los atributos o acciones recaen en un único concepto formal aunque puede darse el caso de que haya varios componentes que requieran una misma característica (por ejemplo dos componentes usan un mismo atributo). En nuestro ejemplo se detecta que todas las

entidades que se pueden renderizar (con un componente gráfico *GraphicsComp* que sale del concepto número *4*) tienen también cualidades para la simulación de físicas y poder colisionar (*PhysicsComp*, concepto número *3*). Ambos estarían interesados en saber cuál es la *escala* (*_scale*) de la entidad pero, debido a la aplicación de FCA, este atributo sólo recaería en uno de esos componentes, en este caso en el *PhysicsComp*. Este operador ofrece la posibilidad al programador de añadir esas características que el considere de alguna manera “perdidas” en la aplicación del FCA.

5. **Clonar**: Ante una serie de características relacionadas con las entidades, la aplicación de FCA distribuye dichas características en conceptos formales. Sin embargo, desde el punto de vista de la creación de juego puede que una serie de funcionalidades puedan llamarse igual y tener un comportamiento semántico similar pero que la manera de llevarse a cabo difiera según qué entidad realiza dichas acciones. De alguna manera lo que pretendemos es añadir polimorfismo a la distribución, pero FCA no es capaz de distinguir cuando una característica se ejecuta de una u otra manera. Para poder indicar que queremos que unas entidades lleven a cabo una funcionalidad de una manera y otras la lleven a cabo de otra lo que podremos hacer es clonar un componente y decir qué entidades mantendrán el componente original para realizar la funcionalidad y qué componentes usarán el componente nuevo. Un claro ejemplo se puede ver con el componente *FightComp*, que proviene del nodo *9* al que se le aplicó como ejemplo un operador *Partir*. Podríamos querer que diferentes entidades disparen de manera distinta ya que puede haber entidades en el juego que disparen con un arma de fuego mientras otras pueden disparar flechas con un arco. Si duplicamos el componente *FightComp* podremos crear los dos comportamientos.

La interacción del experto es totalmente necesaria que exista, en primer lugar porque hay que dar un nombre representativo a los componentes pero también porque el sistema ignora el conocimiento semántico y la información que tiene el programador debido a su experiencia previa. De todas maneras, cuanto más grande es la ontología (y en un juego se supone que el número de tipos de entidades es muy grande) más parecido es el conjunto de componentes propuesto con la solución final tras el refinamiento hecho por el experto, simplemente porque el sistema tiene mayor conocimiento para distribuir la funcionalidad. Además, hay que tener en cuenta que la representación en OWL nos permite

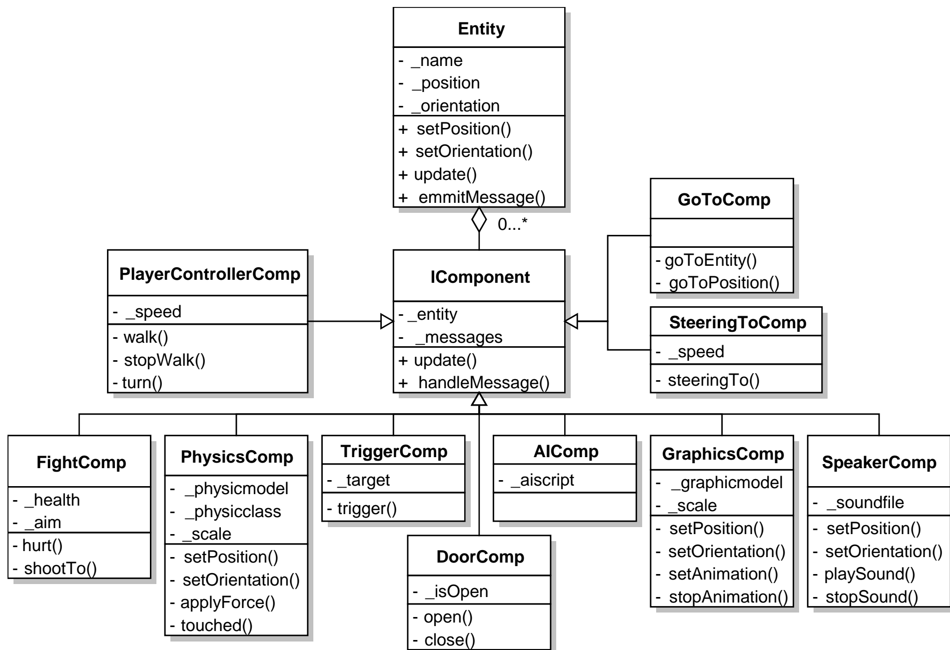

Figura 5.4: Conjunto de componentes creados desde cero por un programador experto sin el uso de ninguna técnica

mantener la coherencia y detectar inconsistencias en el dominio como se comenta en la Sección 5.3, importante ya que después de la creación de los componentes el siguiente paso es generar el contenido del juego y el código que se debe completar (Sección 5.2).

5.1.6. Ejemplo de sugerencia de componentes

La Figura 5.3 mostraba el resultado de la distribución de componentes propuesta tras la aplicación de FCA sobre la ontología de entidades (Figura 5.1) y tras aplicar nuestra técnica sobre el retículo. Una vez propuesto el conjunto de componentes, éstos pueden ser manipulados por el programador para afinar algunos aspectos. El primer cambio que se lleva a cabo es el renombrado de los componentes mediante el operador *renombrar* que ha sido, de hecho, aplicado previamente en la figura para facilitar su lectura.

Una distribución de la funcionalidad en componentes hecha a mano por un experto (partiendo de los mismos datos que se encuentran en la ontología) terminó con un resultado como el de la Figura 5.4 que, si lo comparamos con el resultado obtenido en secciones anteriores (Figura 5.3), apreciamos que es bastante similar. De hecho, cuando se usa

una ontología más rica en cuanto al número de entidades y las características de éstas, la diferencia entre la distribución propuesta por el experto y la propuesta inferida mediante FCA es aún más sutil.

Con el propósito de demostrar cómo un programador usaría los operadores para transformar el conjunto de componentes propuestos, vamos a aplicar los modificadores a la distribución automáticamente generada con el fin de transformarla en la distribución a la que llegó el programador experto. En primer lugar, consideremos el componente *SpeedAttComp*, que tiene el atributo *_speed* pero ninguna funcionalidad. En términos de diseño esto es aceptable, pero rara vez tiene sentido desde el punto de vista de la implementación ya que los componentes se suponen pequeños trozos de funcionalidad (aunque hay excepciones para componentes que realizan tareas de manera autónoma sin responder a eventos externos). La *velocidad* se usa de manera separada en los componentes *PlayerControllerComp* y *AIAndMovementComp* para ajustar el movimiento por lo que decidimos aplicar el operador *mover características* moviendo (y clonando) el atributo *_speed* a ambos componentes y eliminando el componente *SpeedAttComp*. Este operador es coherente con el retículo (Figura 5.2): movemos la intensión del nodo número *9* a los subconceptos (*10* y *11*).

Tras esto, una nueva aplicación del operador *mover características* da lugar a mover la acción *touched* desde el componente *TriggerComp* al componente *PhysicsComp*. Ésto se hace por razones técnicas con la intención de mantener toda la información física en un mismo componente. El siguiente paso consiste en aplicar el operador *partir* sobre el componente *AIAndMovementComp* dos veces. Debido a la falta de tipos de entidades (para que el ejemplo fuese de fácil entendimiento se ha cogido solo una parte del dominio de entidades), varias características residen en el mismo componentes mientras que en la implementación real están divididos. En la primera aplicación del operador, las acciones *goToEntity* y *goToPosition* se mueven a un nuevo componente que llamaremos *GoToComp*. En la segunda aplicación el resultado es un nuevo componente llamado *SteeringToComp* con la acción *steeringTo* y el atributo *_speed*. El componente original es renombrado usando el operador *renombrar* y pasa a llamarse *AIComp*, manteniendo simplemente el atributo *_aiscript*.

En el caso de las figuras mostradas, se ha optado por tener una entidad genérica con una funcionalidad y estado básicos que es común a todos los tipos de entidad (aquellos que se encuentran en el concepto $\top$). Sin embargo, hay componentes que necesitan reaccionar ante la aplicación de una acción (envío de un mensaje) en la entidad. Debido a ésto, se

aplica el operador *añadir características* al los componentes *GraphicsComp*, *PhysicsComp* y *SpeakerComp* con el fin de añadir las acciones *setPosition* y *setOrientation* a esos componentes.

5.1.7. Desarrollo iterativo usando FCA

En la sección anterior hemos presentado una técnica semi-automática para inferir una distribución de componentes a partir de una descripción de entidades contenida en una ontología. El objetivo es ayudar a los programadores a enfrentarse a este tipo de sistema distribuido, que es ampliamente usado en el desarrollo de videojuegos. Mediante el uso de FCA, la técnica divide el comportamiento de las entidades en componentes candidatos y ofrece ciertos mecanismos al programador para que pueda afinar la distribución a su gusto, lo que en cierta manera altera el retículo (o más bien la información que se guardó de él en la ontología).

Tal y como se ha presentado esta técnica es válida para el primer paso de un desarrollo software, pero como se lleva insistiendo en esta tesis, los requerimientos de un videojuego cambian durante su desarrollo, por lo que las entidades y componentes están continuamente modificándose. Cuando los programadores se enfrentan a esos cambios pueden optar por usar (y modificar) la nueva ontología propuesta por los diseñadores y usar la técnica para inferir una nueva distribución de componentes, que seguramente no variará mucho de la distribución anterior. El problema surge si en la iteración anterior los programadores hicieron cambios en la distribución sugerida. Ante una nueva distribución, todos esos cambios deberían ser replicados en el nuevo retículo. Y lo que es peor aun, seguramente ya habría código generado y relacionado con los componentes previamente inferidos lo que complicaría en gran medida la nueva iteración (ver posteriormente la Sección 5.2).

Nuestra intención en esta sección es extender la técnica anterior con el fin de permitir un diseño de software iterativo. En esta nueva aproximación, las modificaciones que el programador realiza sobre un retículo se almacenan para poder ser extrapoladas a otros retículos en futuras iteraciones. Hay que tener en cuenta que los operadores de dominio (Section 5.1.5) son aplicados sobre componentes que han sido creados a partir de un concepto formal por lo que esos operadores pueden ser aplicados sobre conceptos similares de otro dominio distinto, en caso de que ambos dominios compartan la parte del retículo afectada por los operadores.

Desde un punto de vista de alto nivel, para preservar los cambios aplicados sobre una sugerencia de componentes, el sistema compara el

nuevo retículo, obtenido mediante la aplicación de FCA en el dominio modificado, con el anterior. La metodología identifica la parte del retículo que no ha sufrido cambios significativos entre las dos aplicaciones de FCA y por tanto, los operadores que se hubiesen aplicado sobre esa parte del retículo pueden se re-aplicados en el nuevo retículo.

La identificación de la parte del retículo de interés es un proceso semi-automático, donde los conceptos formales de ambos retículos están relacionados por pares. En la parte automática se identifica la parte *constante* del retículo, la cual para nuestro propósito se trata del conjunto de pares de objetos formales que tienen la misma intensión reducida. Los componentes extraídos de los objetos formales que no han sido emparejados se presentan al programador, que puede emparejar antiguos componentes con las nuevas sugerencias si considera que aportan el mismo comportamiento y quiere que se repliquen sobre ellos las modificaciones.

Se debe mencionar que es posible que algún operador aplicado sobre antiguos retículos no pueda ser aplicado ante los cambios en el dominio ya que tras varias iteraciones puede haber algún cambio sustancial que haga que esas transformaciones sean obsoletas. Por ejemplo si se decide prescindir de una funcionalidad del juego, el o los componentes que la implementaban desaparecerán en nuevas versiones y, por tanto, los operadores que se aplicaban sobre ellos no tendrán sentido, aunque tampoco supondrá un problema su no aplicación.

5.1.8. Ejemplo de desarrollo iterativo

En la Sección 5.1.6 se partía de un conjunto de componentes (Figura 5.3) que había sido automáticamente propuesto mediante la aplicación de FCA sobre una descripción ontológica (Figure 5.1). El dominio resultante fue modificado por el programador, mediante el uso de operadores, para terminar con un sistema basado en componentes como el que muestra la Figura 5.4.

Recuperando ahora ese ejemplo supongamos que el diseño del juego ha cambiado y requiere de nuevas características. Los diseñadores quieren añadir dos nuevos tipos de entidades que son *BreakableDoor*, que representa simplemente una puerta que puede ser rota mediante el uso de armas, y un *Teleporter*, el cual mueve a las entidades, que se introducen en él, a otro lugar objetivo que se encuentra generalmente lejos. En la ontología los diseñadores añaden dichas entidades y las describen de manera sencilla. La entidad *BreakableDoor es un Door* que añade la posibilidad de ser *dañada* (mediante la acción *hurt*) y que tiene un nivel de

salud (*_health*) mientras que la entidad *Teleporter es un RealEntity* con su representación gráfica y física pero que, además, puede ejecutar acciones *teleportTo* y tiene cableado su posición de destino, a donde envía las entidades, con el atributo *_destination*. Los programadores recogen este nuevo modelo, pero además se dan cuenta de que, para la correcta implementación de la entidad *ResourceBearer*, ésta debe almacenar cuál es el *enemigo actual* al que está persiguiendo, lo cual se realiza añadiendo el atributo *currentEnemy*.

Tras la modificación del dominio el programador decide usar nuevamente la técnica de sugerencia de componentes para obtener una correcta distribución. La aplicación de FCA sobre el dominio modificado resulta en el retículo de la Figura 5.5, donde los conceptos formales están identificados con letras de la *a* a la *n*.

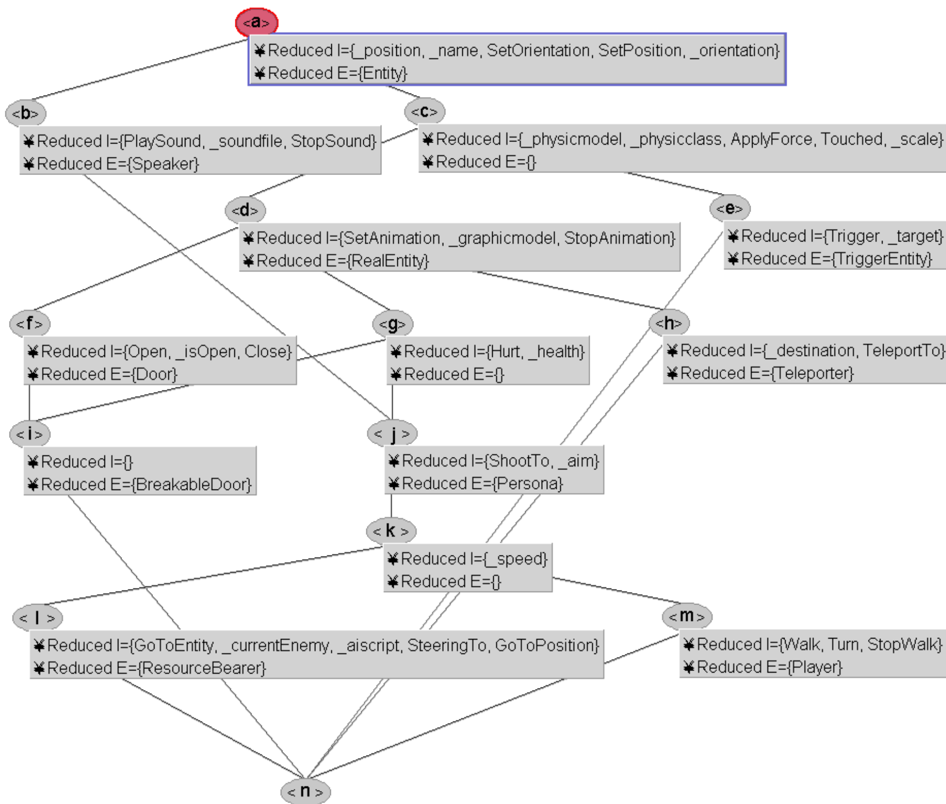

Figura 5.5: El nuevo retículo de conceptos formales

Comparando el nuevo retículo con el retículo de la anterior aplicación de FCA (Figura 5.2), el proceso determina que los pares de conceptos formales *<1,a>*, *<2,b>*, *<4,d>*, *<7,f>*, *<9,k>* y *<11,m>* se mantienen de una iteración a otra (son parte *constante* del retículo). Una vez

terminado el proceso automático de emparejado de conceptos formales, los conceptos formales que quedaron desemparejados, y que no tienen una intensión reducida vacía, son presentados al programador. En este momento, el programador identifica, debido a su experiencia y la similitud, que los pares de conceptos *<3,c>*, *<5,e>*, *<8,j>* y *<10,l>* deben ser añadidos a la parte *constante* del retículo ya que estos conceptos son muy similares entre si (solo hay algún cambio de atributos). En este caso, los conceptos formales *g* y *h* quedan desemparejados y pasarán a ser nuevos componentes.

Por tanto, en estos pasos, la parte del retículo que no ha cambiado de manera significativa ha sido identificada y se pueden extrapolar todos los modificadores que se habían aplicado en iteraciones anteriores y aplicarlos en este nuevo retículo (los cambios en este caso son los discutidos en la Sección 5.1.6). Una vez aplicados, el nuevo conjunto de componentes (bastante similar al anterior) es presentado al programador. La Figura 5.6 muestra estos componentes, donde se puede comparar el resultado con los componentes de la Figura 5.3. La estructura general se mantiene aunque algunas acciones y atributos se han movido entre componentes y han surgido dos nuevos componentes (*C_destination* y *C_health*). Las características resaltadas denotan nuevos elementos (o elementos que se han movido de componente) mientras que las características tachadas hacen referencia a aquellas acciones o atributos que ya no forman parte del componente (*FightComp*). En este punto, el programador puede continuar la iteración aplicando nuevos operadores a este conjunto de componentes (en este caso bastaría con simplemente renombrar los nuevos componentes a *TeleportComp* y *HealthComp*).

5.1.9. Conclusiones

En esta sección se ha explicado como funciona una característica bastante importante en la metodología propuesta, que acelera y permite crear distribuciones software de calidad que proporcionan a la larga una mayor velocidad de desarrollo. Un buen escenario en el que aplicar las técnicas aquí descritas, aparte de en la metodología propuesta, es en casos de re-ingeniería, donde un conjunto de entidades pertenecientes a una jerarquía de clases debe ser transformada en un sistema de componentes. Por este proceso han tenido que pasar alguna saga de juegos como la del Tony Hawks (West, 2006). En cierta manera, una jerarquía puede verse como una ontología donde la herencia sería una relación de tipo *es un* y los atributos y métodos de la clase pueden ser vistos como el estado y la funcionalidad que describe a dicha entidad.

La técnica aquí propuesta ha sido probada con la base de código

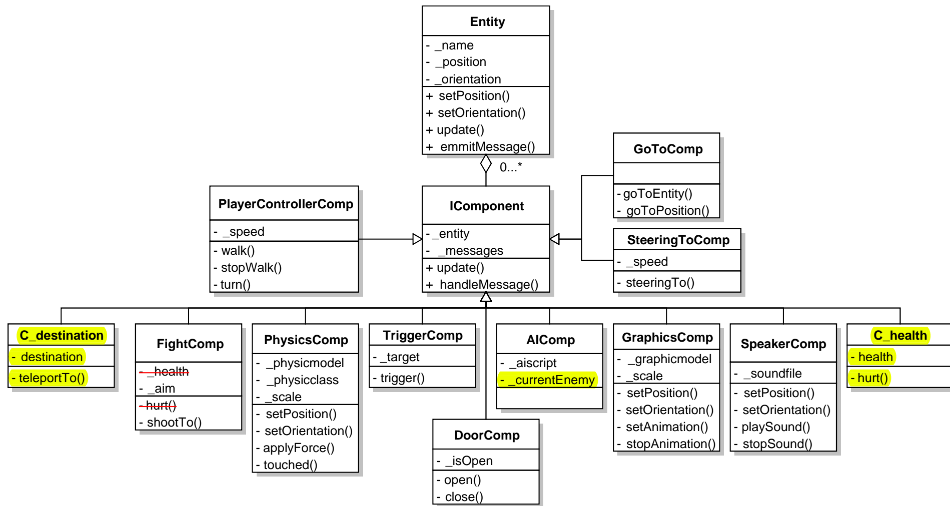

Figura 5.6: Nuevo conjunto de componentes propuesto

de un proyecto del grupo de investigación GAIA[2]. El proyecto llamado Javy 2 (Gómez-Martín et al., 2007) fue originalmente desarrollado usando una jerarquía de clases y luego traducido de forma manual en una arquitectura basada en componentes. De hecho, los ejemplos mostrados a lo largo de esta sección son porciones de la jerarquía original del juego (Figura 5.1) y del conjunto de componentes al que se llego tras la refactorización manual (Figura 5.3). Tras ver que la diferencia existente, entre lo que decidió el programador en la refactorización manual y las soluciones ofrecidas por nuestra técnica, comprobamos que éstas son bastante similares. Podemos por tanto concluir que, si hubiésemos dispuesto de esta técnica cuando decidimos hacer la distribución de funcionalidad en componentes, hubiésemos ahorrado mucho tiempo y esfuerzo (la técnica fue probada con subconjuntos de código mayores con también buenos resultados).

La Sección 7.2 muestra un experimento que consistía en validar esta técnica con desarrolladores de videojuegos y ver si la aplicación de esta técnica mejoraba la calidad de las distribuciones propuestas. Los resultados, como se puede ver en dicha sección, también fueron satisfactorios.

[2]http://gaia.fdi.ucm.es/

5.2. Generación Procedimental de Contenido

5.2.1. Introducción

La metodología presentada en esta tesis plantea el modelado de un dominio formal que es independiente de la plataforma y del lenguaje de programación en el que se quiera crear el juego. De esta manera es más sencillo trabajar desde el nivel semántico, ya que no hay que tener tan presente cada detalle específico de implementación, y un mismo dominio formal podría dar lugar a dos o más implementaciones sobre diferentes plataformas y lenguajes de programación.

El dominio formal que se ha creado, descrito en OWL, es una gran fuente de conocimiento que se puede usar para acelerar la tarea de la implementación del software. En esta sección se explica una técnica que aprovecha la descripción semántica y la usa para generar de manera automática contenido del juego. El contenido que podremos generar a partir del dominio es: la base del código que gestiona la lógica de entidades del juego y ficheros que se usan tanto en el juego como en la edición de niveles (y que incluyen la descripción de prototipos de las entidades). El espíritu de esta técnica es que se puedan añadir, de manera sencilla, nuevos lenguajes de programación y plataformas para la generación procedimental de contenido sin que eso implique modificar la herramienta concreta.

Desde el punto de vista de la generación de código, en función de la plataforma y lenguaje elegido, se generará más o menos contenido pero debería resolverse toda la gestión de entidades, su creación, la inicialización de éstas, la creación de todos los ficheros de componentes, su gestión de inicialización, los mensajes, el paso de mensajes entre componentes, etc. Por otro lado también se generarán las nombradas descripciones de entidades que variarán en función de la plataforma y pueden ser ficheros de texto plano, XML, scripts o lo que sea que necesite tanto el motor de juego como las diferentes herramientas de edición de nivel. Además del ahorro de tiempo que supone la generación procedimental de contenido, una de las mayores ventajas es que, al generarse todo a partir del mismo dominio, se garantiza la consistencia entre las descripciones que tienen tanto el motor de juego como las otras herramientas de edición de contenido.

Aunque se puede generar a partir de la descripción ontológica una gran cantidad del código relacionado con los componentes (más de un 80 % en experimentos realizados con usuarios reales e, incluso, por encima del 90 % ante buenas distribuciones), no se puede inferir todo el

código necesario para que el juego funcione directamente. El programador deberá implementar la funcionalidad (el cuerpo de los métodos) que es disparada por los mensajes en los componentes. La mayor de las complicaciones radica en que, como ya se ha comentado repetidas veces, el dominio del videojuego se va transformando a lo largo del desarrollo. Si la generación procedimental de código se hiciese cada vez desde cero esto implicaría que en cada iteración los programadores deberían reimplementar una y otra vez la misma funcionalidad. Para que esto no suceda, se debe establecer una relación directa entre los elementos del dominio en OWL y el código generado para que, cuando se modifique el dominio y haya ya trabajo de implementación, se preserve el código previamente generado por los programadores.

Ese vínculo creado entre los elementos semánticos del dominio y el código y contenido real de la implementación permite incluso trabajar y hacer refactorizaciones en el dominio formal que se aplican de manera directa a la implementación, actualizando los niveles de juego, moviendo métodos o atributos de un componente a otro, partiendo componentes, juntando dos o más en uno solo, etc. Si la arquitectura basada en componentes ya es de por si muy flexible ante los cambios, gracias al dominio formal, y a esta técnica, la flexibilidad crece mucho más ya que nos permite hacer grandes refactorizaciones en muy poco tiempo y sin necesidad de pensar a bajo nivel. Lo que permitimos es realizar cambios de alto nivel sin miedo a romper el proyecto o miedo a que produzca un alto coste en carga de trabajo para el programador. Seguimos el espíritu de las metodologías ágiles.

5.2.2. Contenido generado a partir de la descripción semántica

La descripción semántica que se encuentra descrita en la ontología está muy orientada a la descripción de una arquitectura basada en componentes, aunque aportando más conocimiento semántico acerca de la relación entre las entidades, definición de su funcionalidad y dependencias o rangos y tipos de sus atributos. Esta representación conceptual, aunque esté orientada a la composición de funcionalidad mediante componentes, es independiente del lenguaje de programación y de la plataforma y, gracias a esa independencia, nos permite generar procedimentalmente gran parte del contenido asociado al videojuego para diversas plataformas y lenguajes.

Gran parte del contenido generado es el código de programación que en otra circunstancia tendrían que desarrollar los programadores

basándose en la descripción del dominio pero, además, se genera información asociada a la descripción de las entidades usada por el motor del juego y también usada para que se simplifique el trabajo de los diseñadores cuando tienen que montar los diferentes niveles del juego. En las siguientes secciones se explica qué contenido se genera y siguiendo qué patrones.

5.2.2.1. Código fuente de la capa lógica

El dominio que se ha modelado en OWL por los diseñadores y programadores no sólo no está atado a un lenguaje de programación o plataforma, sino que no impone una estructura o implementación concreta. Hay diferentes tipos de implementaciones de arquitecturas basadas en componentes y, en función del lenguaje, la plataforma, el juego, el equipo de desarrollo, etcétera, se decide cuál es la mejor implementación.

Se puede partir de un modelo puro, en el cual ni siquiera existe una clase *entidad* sino simplemente componentes con identificadores comunes (y que hacen ese papel de entidad) almacenados en alguna estructura, o se puede optar por la existencia de una clase *entidad* que contenga algunos elementos comunes a todas las entidades (por ejemplo posición, un nombre, un identificador...) y una colección genérica de componentes que le otorguen el comportamiento esperado. Luego, los atributos que especializan la funcionalidad pueden encontrarse en los propios componentes que los necesitan o estar en una lista de pares clave-valor en la entidad (u otra estructura) y que los componentes accedan a ellos, compartiendo así los mismos valores entre diferentes componentes. Tampoco hay un estándar para definir que tipo de representación toman los mensajes, que son los que lanzan la ejecución de acciones primitivas, ya que estos pueden ser objetos de clases concretas, estructuras de datos o simplemente invocaciones a métodos por medio de introspección.

La ventaja de tener el dominio definido formalmente es que podemos adaptarnos a cualquiera de estas implementaciones y generar código de manera procedimental. Como se anticipaba en el Capítulo 4, para realizar la generación procedimental a partir del conocimiento de la ontología, hacemos uso de plantillas que definan tanto el lenguaje de programación como el modelo de implementación deseado. De esta manera se puede además controlar temas como la inicialización, de las entidades y los componentes, o su ciclo de vida, ya que todo se generará en base a plantillas. La lógica de juego tendrá una parte estática, que comprenderá toda la parte de gestión de entidades, mapa del nivel, entidad genérica (aunque no necesariamente ya que puede no haberla o generarse dinámicamente con algún atributo básico), clase base o inter-

faz de la que heredan todos los componentes y que media en el envío de mensajes, etc. Esta parte estática del dominio será idealmente la misma para diferentes juegos. Por otro lado, la implementación tiene una parte dinámica que será la que agregue comportamiento al juego y que, desde el punto de vista del código, viene marcada básicamente por los mensajes que se pueden enviar y, sobretodo, por los componentes, aunque puede englobar más elementos como ficheros de proyecto o la entidad genérica.

La parte estática del juego, como es de suponer, no varía, por lo que desde el punto de vista de la generación de contenido nos centramos en la parte dinámica. Ésta es la parte que caracteriza a un juego concreto, que varía de uno a otro y que sufre las mayores modificaciones a lo largo del desarrollo debido a los cambios de mecánicas, elementos de juego y de reglas. Por tanto, una vez elegido un motor de juego o plataforma, los programadores deberían crear las plantillas necesarias para la generación de los diferentes elementos dinámicos del código. El tipo de plantilla a usar depende del gusto de los programadores, nosotros hemos probado con diferentes tecnologías como *XSL Transformations* (XSLT)[3], un sistema de plantillas propio que permite sustituir etiquetas de un conjunto de plantillas por texto generado a través de un API que genera el programador o *Apache Velocity*[4], un motor de plantillas basado en Java que permite hacer referencia a métodos definidos dentro del código Java.

Una vez se han creado las plantillas, se ha comenzado el desarrollo del juego y se ha generado por primera vez el contenido dinámico de la lógica de juego, el programador ya no tendrá que escribir toda la capa lógica del juego y de la gestión de entidades y componentes, sino que simplemente deberá implementar las funcionalidades requeridas por los mensajes que se envían (acciones y sensores descritos en el domino).

En este sentido, el desarrollo basado en el uso de plantillas para el código fuente se agiliza mucho y ahorra mucho trabajo; prueba de ello son los experimentos que se llevaron a cabo en esta tesis (Capítulo 7) ya que sin esta técnica hubiese sido imposible. Sin embargo, el desarrollo mediante este sistema exige seguir una disciplina de trabajo. El código se define por regiones que ligan fuertemente la implementación con los conceptos formales de la ontología y los lugares en los que se debe escribir el código están delimitados para su futuro reaprovechamiento. La representación semántica es básica en la metodología propuesta por lo que todo cambio o adición de contenido que se requiera hacer desde

[3]urlhttp://www.w3.org/TR/xslt

[4]http://velocity.apache.org/

el punto de vista de la funcionalidad e implementación del juego debe hacerse primero sobre el modelo formal, la ontología, y luego generar el código asociado, en lugar de implementar directamente la funcionalidad (nuevo método, mensaje, atributo, etc.) en el código fuente.

Básicamente la idea es que la generación resuelva la gestión de entidades y componentes, sus métodos de ciclo de vida (inicialización, actualizaciones periódicas de estado, etc.) y el paso y recepción de los mensajes. A su vez, por cada componente descrito en el dominio se generan ficheros de código en los que automáticamente se debería garantizar la implementación automática de la inicialización del componente, asignación de los valores leídos del mapa para los atributos, gestión de la aceptación y procesado de los mensajes que se espera recibir y también las *cáscaras* de la funcionalidad, que viene a ser una región o método vacío por cada mensaje que acepte dicho componente. Idealmente, lo único que debería hacer el programador es rellenar esos huecos vacíos en los que se implementa la funcionalidad del componente, aunque por cuestiones de implementación puede ser necesario escribir algo más de código.

El código generado queda estructurado y los diferentes ficheros o secciones quedan fuertemente ligados a representación ontológica mediante meta-anotaciones. Por ejemplo, la funcionalidad descrita para un mensaje concreto quedará implementada en un método del componente que, a su vez, se asocia (mediante el uso de regiones u otras técnicas similares) al identificador del mensaje descrito en la ontología. Por cuestiones de implementación puede ser necesario escribir código más allá de esa funcionalidad, y es que para que ésta funcione se puede requerir, por ejemplo, inclusiones de otros ficheros. Además, aunque se ha insistido que todo cambio en el juego debe ser reflejado primero en la ontología, puede haber ciertos atributos o métodos de muy muy bajo nivel que no se considere necesario (e incluso se considere perjudicial) reflejar en la representación semántica. Debido a esto es interesante permitir unas regiones especiales para declarar e implementar esos pequeños detalles concretos que generalmente tienen que ver con el lenguaje usado. Todas estas regiones donde el programador deberá escribir su código también son marcadas con meta-anotaciones y es que, aunque de momento estemos hablando de un desarrollo en cascada donde primero se genera el *esqueleto* de código y luego se implementa, el desarrollo será realmente iterativo, donde se regenerará el código bastante más a menudo. La importancia de realizar estas anotaciones será más que justificada por dicho desarrollo iterativo, que se describe en la Sección 5.2.3.

En cuanto a la gestión de las plantillas hay que ser flexibles también

ya que en función del motor, lenguaje y plataforma que se use para el desarrollo no se necesitarán el mismo número de plantillas. Por ejemplo, en un entorno C++ con un motor propio se requerirán muchas más plantillas que en un desarrollo C# realizado en un motor comercial como *Unity*[5] ya que para empezar, por cada componente es necesario generar dos ficheros en C++ (.h y .cpp) mientras que en C# es suficiente con uno. Pero no sólo eso ya que un motor comercial como *Unity* da resueltas muchas cosas como ficheros de configuración de proyecto, paso de mensajes, ciclo de vida de los componentes con su inicialización, etc. lo que supone mayor cantidad de código y contenido a generar en el caso de C++. Por tanto, se debe montar un sistema de gestión interna de plantillas que permita dicha flexibilidad y que además esté independizada del entorno para el que se crean dichas plantillas, ya que para incorporar un nuevo grupo de plantillas para un nuevo motor no se debería tener que cambiar el sistema.

Esta manera de programar, donde hay ciertas normas que es necesario seguir, puede resultar molesta en un principio pero ayuda a tener un código ordenado, donde es todo más sencillo de localizar (ya que todos los ficheros del mismo tipo siguen el mismo esquema) y es totalmente necesario para poder realizar un desarrollo iterativo en el que, como veremos, los cambios deben aplicarse sin alterar el trabajo previo realizado por los programadores. En el Capítulo 6 se darán detalles concretos de una implementación de esta técnica que hemos llevado a cabo, dando más detalles de las plantillas que se han creado para diferentes lenguajes y plataformas y como éstas se han gestionado.

5.2.2.2. Descripción de las entidades

A parte del código fuente generado para la capa lógica del juego o de ficheros de proyecto, también es necesario generar ficheros que describan a las entidades del juego y que sean compatibles con el motor de juego concreto. Idealmente, si construyésemos nuestro propio motor de juego, las descripciones de las entidades podrían inferirse directamente de la ontología, la cual podría usarse luego durante el desarrollo del juego para inferir mucho más conocimiento. Sin embargo, con la finalidad de ser flexibles, permitimos que las descripciones del dominio se adapten a cualquier motor mediante la realización de plantillas y, estos motores, tendrán sus propios ficheros de descripción de entidades. Por tanto, de la misma manera que se genera el resto del código se generará ese o esos ficheros que describan las entidades. Este tipo de ficheros o

[5] urlhttp://unity3d.com/

descripciones de las entidades prototípicas de un juego pueden recibir diversos nombres como *blueprints* y *archetypes* (el primero contiene la descripción de las entidades en función de los componentes y el segundo los valores por defecto de los atributos) o *prefabs* (descripción en función tanto de los componentes como de los valores de los atributos).

La mayor cantidad del contenido generado tiene relación directa con los programadores, código y motor del juego. Sin embargo hay otra serie de herramientas que pueden aprovecharse del conocimiento del dominio. Cuando los diseñadores del juego diseñan un nivel deben tener una herramienta que les permita distribuir las entidades del juego a lo largo de un escenario. Esas entidades deberán obviamente coincidir con las modeladas en el dominio, y deben estar descritas acorde a su funcionalidad y atributos, de manera que el diseñador pueda aportar el grano fino en la instanciación de las entidades (añadiendo un comportamiento especial, modificando la vida, la escala, etc.). Del mismo modo, las herramientas usadas para diseñar comportamientos de personajes de juego (mediante máquinas de estado, árboles de comportamiento, etc.) mejoran su usabilidad y reducen futuras inconsistencias si están descritas las habilidades que el personaje puede realizar y cuales son sus atributos básicos. De esta manera, durante la edición de un personaje, se muestra al diseñador sólo las acciones básicas que puede realizar el personaje, los eventos que procesa y los atributos que tiene y se pueden consultar. Casos de herramientas más genéricas, donde se presenta toda la batería de acciones, eventos y atributos del juego o incluso donde se permite editar los identificadores, son más propensas a quedarse desfasadas durante el desarrollo y provocar errores durante la ejecución.

En grandes desarrollos las herramientas para la edición de niveles y comportamientos se realizan por el mismo equipo que desarrolla el motor de juego y, por tanto, suelen usar la misma descripción del dominio para todas ellas. En un desarrollo que use nuestra metodología, todas esas herramientas necesarias para la creación del juego podrían desarrollarse de manera que se nutriesen de la ontología modelada. Sin embargo, desarrollos algo más pequeños suelen aprovechar herramientas ya existentes para desarrollar las diferentes tareas (motor, niveles, comportamientos, etc.), de manera que las descripciones del dominio de cada una difieren, habiendo que duplicar el esfuerzo durante el desarrollo; si se añade o modifica una entidad en el juego, los diseñadores deberán actualizar el dominio de sus herramientas. Como se ha dicho esto suele traer problemas ya que no se garantiza la consistencia entre los diferentes dominios.

Continuando con la idea del uso de plantillas para generar el conte-

nido del juego, se pueden crear plantillas para todas esas herramientas que usan los diseñadores (y programadores). Al crearse todas ellas a partir del mismo dominio no se corre el riesgo de perder la consistencia entre las diferentes descripciones y se ahorra trabajo. Una vez más, la idea de este tipo de desarrollo es centralizar los esfuerzos en el desarrollo de la ontología, describiendo el contenido semántico del juego sólo una vez, y extrayendo de ahí todo el conocimiento posible para las diferentes fases del desarrollo y para las distintas tareas.

5.2.3. Iteratividad

La Generación de código y contenido de manera procedimental basándose en descripciones ontológicas no es una idea nueva en otras áreas de la informática (Tetlow et al., 2006), aunque si que es una técnica poco madura pese a sus grandes ventajas (Deline, 2006). En las secciones anteriores se ha propuesto una manera de generación de contenido de juego basándose en la descripción formal realizada por diseñadores y programadores y se ha propuesto también un método concreto para la generación de ese contenido, el uso de plantillas. Este proceso acelera enormemente el primer paso del desarrollo de un juego, mejora calidad y estructura del código y fuerza a que la implementación sea coherente y consistente con el dominio modelado, dejando mucho menor espacio a la ambigüedad y malinterpretación de la descripción del juego. Aun así, se pretende dar un soporte a la metodología propuesta en el Capítulo 4 donde uno de los pilares básicos es la iteratividad, por lo que deberemos ser capaces de permitir generaciones de contenido iterativas e incrementales.

Si cada vez que se hiciese un cambio en el dominio formal generásemos desde cero toda la infraestructura y contenido estaríamos haciendo un flaco favor al desarrollo, ya que se tiraría por tierra toda la implementación de funcionalidad realizada por los programadores hasta dicho momento. Por tanto, las generaciones sucesivas no son independientes y no solo se generarán a partir de la descripción semántica del dominio formal, sino que también se deberá tener en cuenta el software ya generado en iteraciones previas.

En las anteriores secciones se hablaba de que el desarrollo que proponemos debe seguir un método y disciplina de programación, donde el orden de los contenidos es muy importante y la diferente funcionalidad debe estar implementada en las regiones reservadas a dicho efecto. Estas regiones tenían meta-anotaciones que las relacionaban con conceptos de la ontología o con secciones concretas de los ficheros. En este momento, estas regiones delimitadas cobran su importancia y es que,

para generaciones iterativas, se buscará y mantendrá el código realizado por los programadores siempre y cuando éste se encuentre dónde se espera. Por ejemplo, si en un componente se han hecho inclusiones de ficheros externos para implementar funcionalidad y no se ha descrito en la región esperada y meta-etiquetada para dicho efecto, en la siguiente generación de código dicho contenido se perderá, con los consiguientes problemas de compilación.

Desde una generación de un dominio a otra se realizan cambios en las entidades, en los mensajes y, lo que es más conflictivo, desde el punto de vista de la generación, en los componentes. En un desarrollo ideal, la evolución de éste a lo largo del tiempo simplemente significaría la adición o supresión de componentes o la adición o supresión de funcionalidad y atributos en los componentes (y obviamente creación o modificación de las entidades de juego). En este caso sencillo, la generación incremental durante las iteraciones es muy complicada:

- Se generaran los componentes declarados en la ontología (como se ha comentado en la Sección 5.2.2.1)
- Para cada hueco o región de la distribución generada en la que se espera que el programador escriba código, se busca si en la antigua distribución, donde ya hay funcionalidad implementada, existe un hueco, método o región homónima.
- Si existe región homónima se coge y se añade el código a la nueva implementación, sino se deja en blanco o con la información por defecto.

En estos tres pasos se ha hecho una nueva generación, pero de cara a los programadores parece que se han mantenido los mismo ficheros, a los cuales se les han añadido (o suprimido) nuevas acciones, sensores o atributos y se han creado nuevos ficheros para los nuevos componentes, además de haberse modificado el conjunto de mensajes o la descripción de las entidades.

Sin embargo, en un desarrollo real, las modificaciones que se pueden realizar a los componentes son bastante más drásticas, ya que éstos pueden cambiar de nombre y pueden cambiar de nombre sus atributos o los mensajes que eran capaces de llevar a cabo (y por tanto los métodos que implementan la funcionalidad cambiarían de nombre en la nueva generación). Además, también pueden sufrir refactorizaciones, de manera que componentes grandes con mucha funcionalidad se pueden partir en dos o más componentes o, el caso contrario, dos o más componentes se pueden fusionar en uno solo o se puede trasladar funcionalidad de

uno a otro. Todos estos cambios en el dominio complican la relación de equivalencia entre los componentes y mensajes de un dominio con los componentes y mensajes del mismo dominio en una iteración anterior, lo que a su vez complica mantener y colocar adecuadamente el código antiguo en la nueva distribución generada.

Para solventar este problema contamos con dos herramientas. La primera es que hemos dicho previamente que el código generado está estrechamente ligado con los diferentes elementos de la ontología. Cada elemento de la ontología (entidades, componentes, mensajes o atributos) tienen un identificador único que nunca cambia pese a que cambie su nombre o sus elementos. Gracias a esto, cada implementación de componente está anotado con el identificador del concepto que le representa en el dominio formal y no solo eso, sino que también cada método de un componente que aporta funcionalidad al mismo está anotado con el identificador del mensaje al que se corresponde en la ontología. Esto permite desligar los diferentes conceptos de sus nombres de manera que, si se modifica el nombre de un componente o mensaje de una iteración a otra, se podrá seguir recuperando el código asociado a los componentes y mensajes ya que los identificadores nunca cambiarán.

La segunda herramienta con la que se cuenta es que, como se ha comentado también en la Sección 5.1.7, las modificaciones que el programador realiza sobre el dominio se almacenan de una iteración a la siguiente, así que se puede crear un mapa de enlaces que indique de qué componentes de una iteración antigua proviene un componente de la iteración actual. Los casos que se pueden dar son los siguientes:

- En el caso más claro, en el que un componente *a* no ha sufrido una refactorización de una iteración a la siguiente, la relación es obvia: el componente *a* de la nueva iteración se corresponde con el componente *a* de la anterior iteración. El código con el que rellenar la funcionalidad y regiones del nuevo componente se obtendrá de los métodos y regiones del componente *a* de la iteración anterior, donde los mensajes que ya se implementaron, y se mantengan en la nueva descripción, tendrán el mismo identificador en ambos componentes.

- En el caso en el que un componente *a* de una iteración previa haya sufrido un proceso de división en dos o más componentes (pongamos como ejemplo dos, *b* y *c*): tanto el componente *b* como *c* provienen del componente *a* de la anterior iteración. Ambos componentes nuevos obtendrán el código necesario del mismo componente, lo cual tiene lógica ya que lo único que se ha hecho es dividir

la funcionalidad. Las regiones especiales, como las que definen las inclusiones de ficheros realizadas o declaraciones e implementaciones de elementos con poco interés semántico, se copian desde el componente a a los componentes b y c, ya que ambos pueden requerir dicha información.

- En el caso en el que varios componentes de una iteración anterior (pongamos a y b) se fusionen en un nuevo componente c: el nuevo componente c está directamente relacionado con los componentes a y b, de manera que a la hora de rellenar la funcionalidad, se busca la existencia de código tanto en uno como en otro componente ya que sus métodos pueden provenir de mensajes que ejecutase cualquiera de los dos o ser una nueva funcionalidad (y en ese caso, al no estar en ninguno de los previos se deja el método vacío). En este caso, las regiones especiales se complementan con la suma del código extraído de los dos componentes antiguos ya que será necesario para la realización de las diferentes funcionalidades.

- Por último, en el caso de que un componente a traslade funcionalidad a otro b: el componente a de la nueva generación provendrá y obtendrá su código del componente con el mismo identificador de la iteración anterior (a) mientras que el componente b provendrá y obtendrá piezas de código tanto del componente a como del componente b y las regiones especiales de este último se generan como en el caso anterior con la suma del código extraído de los dos componentes antiguos.

Gracias a la meta-información que se tiene en los ficheros del código fuente, que relaciona la implementación de los diferentes apartados con los conceptos contenidos en el dominio formal, y gracias también al mapeo que se realiza entre los componentes de una iteración con respecto a los componentes de una iteración anterior, podemos garantizar la generación de contenido de manera iterativa para que agilice el desarrollo no solo en su fase inicial, sino también durante el resto del desarrollo, manteniendo el código que van creando los programadores.

5.2.4. Conclusiones

El apoyo que ofrece esta técnica en el desarrollo de videojuegos es útil la primera iteración del juego. Sin embargo, donde la técnica realmente demuestra su valía es en las siguientes iteraciones. Cuando el dominio formal se modifica, gracias a la metodología seguida, esta técnica es capaz de rastrear las diferencias entre el dominio antiguo y la

nueva definición, refactorizando la distribución de componentes, y todo el código asociado a éstos, manteniendo el trabajo realizado por los programadores. Este trabajo se añade a los ficheros, generados procedimentalmente, a partir de las plantillas, moviendo esos trozos de código a nuevos ficheros o localizaciones cuando se necesita.

Permitir y garantizar el desarrollo iterativo es uno de los valores de esta técnica ya que permite a los programadores hacer refactorizaciones de alto nivel garantizando el correcto funcionamiento de las mecánicas que ya estaban implementadas sin tener que modificar nada del código de manera manual. Sin embargo no es el único; la carga de trabajo que es automatizada en la generación de contenido es muy grande, hasta el punto de que, ante buenas distribuciones de funcionalidad, el código dinámico de la lógica de juego que se genera (aquél que cambia de un juego a otro) supera el 90 %. Esto ha sido clave para poder realizar varios experimentos con usuarios reales que de otra manera no hubiesen sido posibles y que se encuentran detallados en el Capítulo 7.

Por todas estas razones podemos concluir que esta técnica tiene un inestimable valor dentro del ciclo de desarrollos ágiles debido, no solo al ahorro de trabajo en generación de contenido, sino porque este tipo de desarrollos es propenso a errores y refactorizaciones. De esta manera, se evita tener que bajar a nivel de código para realizar ciertas modificaciones.

5.3. Chequeo de inconsistencias

5.3.1. Introducción

Con una arquitectura dirigida por datos como la arquitectura basada en componentes (Sección 2.4.2) se promueve la flexibilidad, reusabilidad y facilita la creación de nuevos tipos de entidades. Sin embargo, al mismo tiempo introduce un alto factor de riesgo, donde las cosas pueden ir realmente mal si se crean entidades inconsistentes que no funcionen en tiempo de ejecución. Antes de entrar en detalles, podemos hacer una analogía con el uso de lenguajes de script para crear comportamientos. Estos lenguajes extraen el comportamiento fuera del motor de juego, lo cual lo hace mucho más independiente del juego y facilita y agiliza la elaboración de comportamientos ya que no hay que hacer un proceso de compilación e incluso, en algunos casos, se pueden modificar en medio de la ejecución. Sin embargo, un script con un error puede provocar que el juego se ejecute de una manera incorrecta, siendo bastante más compleja la depuración de scripts que de código. En este sentido, un error

en un fichero de entrada provoca un comportamiento erróneo en tiempo de ejecución. Cuando usamos una aproximación basada en componentes sucede un poco lo mismo, al dejar información fuera de la aplicación (como la descripción de las entidades en función de sus componentes o los valores por defecto de los atributos de estas entidades) existen más oportunidades de que la ejecución falle debido a aspectos externos, ya que el compilador no puede asegurar la consistencia. En ambos casos el motivo es que los desarrolladores pierden el *feedback* acerca de si las cosas están bien o no ya que el juego siempre es capaz de ejecutarse, perdiendo el ideal de "Si el compilador no avisa, mi programa debería funcionar" (Sweeney, 2006).

El origen del problema introducido en esta sección fue detectado por el propio autor de esta tesis al empezar a trabajar con la arquitectura basada en componentes, momento en el que sufrió lo complicado que es detectar inconsistencias en entidades formadas, en tiempo de ejecución, mediante la composición de componentes software. El principal motivo es que se pierde el análisis del compilador. Nuestros primeros intentos para anticipar la detección de inconsistencias y mejorar su depuración fueron publicados en (Llansó et al., 2009b) donde se detectaban inconsistencias tanto a nivel de entidad y sus componentes como a nivel de mapa cuando se asociaban entidades junto con estructuras complejas como *árboles de comportamiento* (BT debido a su nombre en inglés *behaviour tree*) o *máquinas de estado* (FSM de *finite state machine*). En este estudio se proponía el uso de lo que llamamos *componentes auto-reflexivos*, que aportaban mecanismos de implementación, desarrollados por los programadores, que permitían a los componentes describirse a sí mismos, indicando qué mensajes eran capaces de aceptar y de qué resolución de mensajes dependían para poder llevar a cabo ciertas peticiones (que acciones debían poder resolver otros componentes de la misma entidad para que la entidad fuese consistente). Los componentes reflexivos empezaron a tener también un peso semántico (Sánchez-Ruiz et al., 2009a) que derivó en la tesis de máster Llansó et al. (2009a).

En esta sección nos centraremos en ver cómo podemos usar el conocimiento ontológico para, razonando sobre él, poder detectar posibles inconsistencias. En primer lugar (Sección 5.3.2) se indica qué tipo de inconsistencias podríamos detectar a nivel de entidad, ya sea en sus componentes (Sección 5.3.2.1) o en sus atributos (Sección 5.3.2.2) y se explica con un claro ejemplo los tipos de casos con los que nos podríamos encontrar y cómo se debería reaccionar ante ellos. En la Sección 5.3.3 se va un paso más allá y se introducen técnicas de detección de inconsistencias en la edición de niveles que crean los diseñadores, teniendo

en cuenta que en los editores de niveles se pueden modificar los prototipos de entidades descritas en el dominio ontológico y que se pueden crear estructuras que controlen el comportamiento de los personajes, tales como FSMs o BTs, que son un nuevo foco de inconsistencias. Esta sección concluye con un ejemplo donde se reproducen ejemplos típicos que se pueden producir cuando se edita un nivel. Por último, en la Sección 5.3.4, se evalúa el trabajo realizado.

5.3.2. Inconsistencias a nivel de entidad

Nuestra metodología se basa en una arquitectura altamente dirigida por datos, lo que idealmente agiliza la producción y recorta tiempos de desarrollo. El problema radica en que, al sacar los datos del motor de juego, se pierde el control sobre la especificación de los mismos. Sólo se conoce en tiempo de ejecución, cuando las diferentes piezas especificadas en los datos se van creando, parametrizando y ensamblando unas con otras. A nivel de entidad, los videojuegos que se hacen desde cero, sin usar un motor de juego, suelen definir en ficheros de texto la descripción de las entidades en función de sus componentes y de los valores por defecto que tienen los atributos de la entidad. Al especificar esos datos en un fichero de texto normal el conocimiento que se obtiene es muy limitado y no está libre de errores tanto sintácticos como semánticos. Los errores sintácticos de la especificación pueden ser fácilmente subsanados creando una pequeña aplicación que de el formato a los valores introducidos pero los errores semánticos son más complicados de detectar.

Gracias a nuestra metodología, la especificación de las entidades se encuentra descrita en una ontología, con un alto conocimiento semántico, en vez de en ficheros de texto, lo cual nos va a permitir detectar ciertas dependencias y restricciones que de otra manera sería imposible. La semántica OWL es de *mundo abierto*, por lo que es posible representar información parcial o incompleta de manera que, si un elemento no está declarado, no quiere decir que su existencia sea falsa, simplemente no se conoce si está o no está. Ésto no nos resulta especialmente útil en la descripción semántica del dominio de nuestro juego así que asumimos una representación de *mundo cerrado* dónde, sí un concepto o relación no está descrito, se asume su falsedad. Por tanto, lo que hacemos para detectar inconsistencias es simplemente tratar el dominio descrito en OWL como un *schema* o lenguaje de validación para *RDF data*. Al procesar nuestro dominio, si hay alguna inconsistencia con la *asunción del mundo cerrado* quiere decir que la descripción de las entidades establecidas por los programadores no son consistentes. Ésto nos ofrecerá a su

vez un *feedback* indicando qué parte de la definición ha sido violada, de manera que podremos trasladar al programador cuál es el problema de dicha inconsistencia.

En este apartado nos vamos a centrar en identificar qué tipo de inconsistencias existen a nivel de entidad, siendo más específicos a nivel de componentes y de atributos, y vamos a ver cómo podemos detectarlas gracias a la representación ontológica de los conceptos que hemos elegido. La detección de inconsistencias en el dominio es útil para los dos roles que intervienen en la metodología de desarrollo, programadores y diseñadores, ya que ambos requieren de ese *feedback* que les indique si están haciendo las cosas de manera adecuada. De todas formas, a este nivel, las inconsistencias más complejas, aquellas que se producen a nivel de componentes, son sólo interesantes para los programadores ya que los componentes son un paradigma de programación que solo concierne a este tipo de rol.

5.3.2.1. Relacionadas con los componentes

Los componentes son idealmente piezas de funcionalidad autocontenidaque se comunican entre ellos mediante el paso de mensajes, activando funcionalidades ajenas. A veces, cuando una funcionalidad es suficientemente grande puede partirse en funcionalidades más pequeñas, haciendo que componentes de alto nivel invoquen a componentes de más bajo nivel o que una funcionalidad global esté repartida en diferentes componentes. Esto es importante también para introducir polimorfismo dentro de las entidades, de manera que diferentes componentes puedan responder al mismo mensaje, realizando la misma acción de alto nivel, pero con diferentes implementaciones.

Todas estas comunicaciones que se deben producir entre los componentes son completamente necesarias pero, a su vez, son una fuente de errores debido a las dependencias que se forman entre componentes. Ciertos componentes requieren de la existencia de otros componentes para funcionar correctamente. Cuando trabajamos con arquitecturas más tradicionales, como puede ser la herencia de clases, las clases derivadas dependen de sus antecesoras para funcionar de manera adecuada. Por ejemplo, supongamos una entidad *NPC* (Figura 5.7) que implementa un método de *patrulla* que va enviando a la entidad por una serie de puntos invocando a un método abstracto *ir-a*. De esa entidad *NPC* podrían heredar dos tipos de personaje con cualidades diferentes, donde uno de ellos es volador mientras el otro es terrestre por lo que la implementación del método *ir-a* diferiría, aunque el método de *patrulla* es el mismo para ambos.

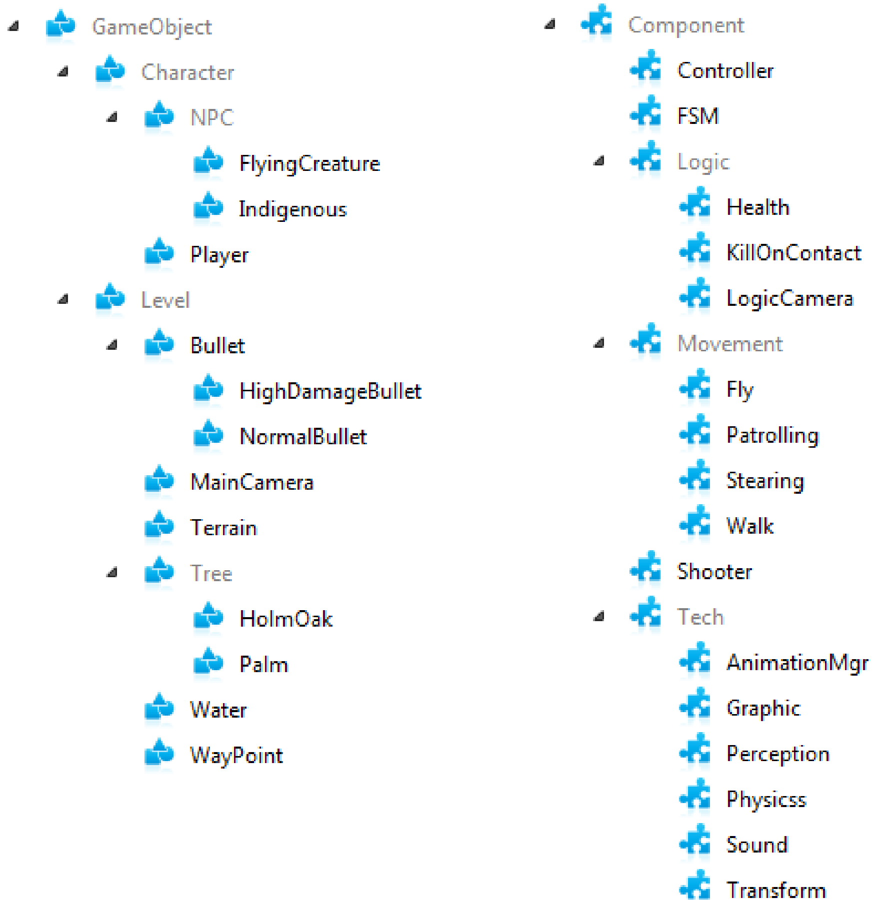

Figura 5.7: Jerarquía de entidades de juego

Cuando nuestro juego lo desarrollamos con una arquitectura basada en componentes, estas capacidades de las entidades se parten en diferentes componentes que, en vez de comunicarse mediante la invocación de un método, lo hacen mediante el paso de mensajes (Figura 5.8). Un caso como el anterior resultaría en un componente de *patrulla* que enviaría mensajes de tipo *ir-a* para que otro componente (bien el componente de *vuelo* o el de *andar*) interpretase el mensaje y realizase la acción requerida.

En ambos casos existe una dependencia, en las jerarquías basadas en herencia las entidades hijas de *NPC* deben implementar un método *ir-a* mientras que en las entidades que tengan un componente de *patrulla*, debe haber otro componente que interprete los mensajes de tipo *ir-a*. Como se describe en el Capítulo 4, las dependencias en las descripciones ontológicas de los componentes se realizan de la siguiente manera:

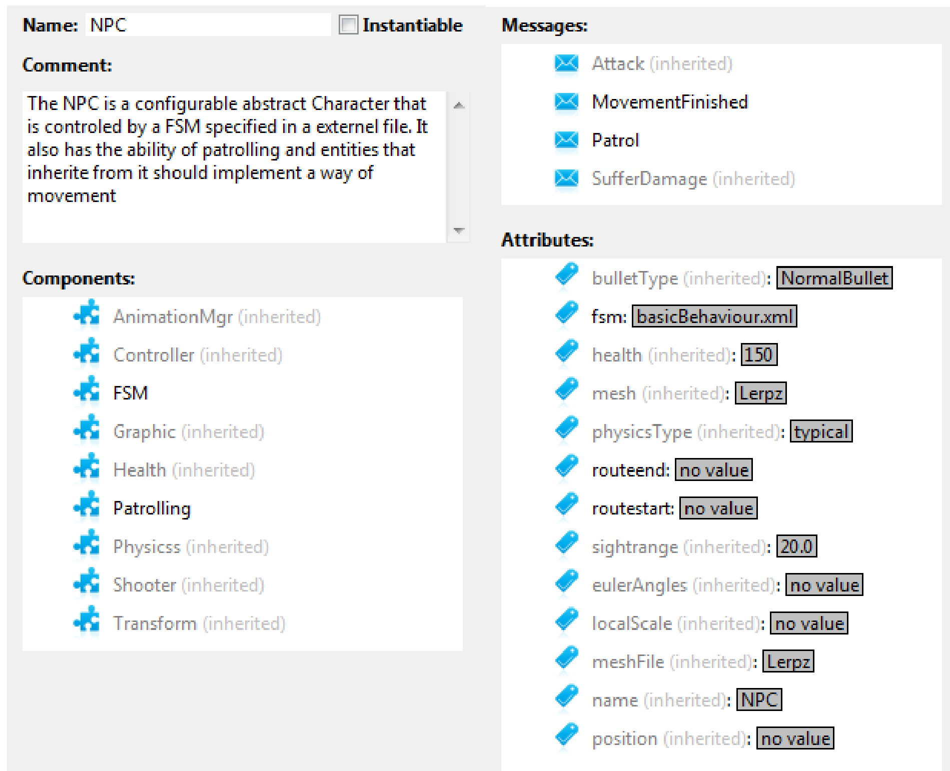

Figura 5.8: Entidad NPC definida en función de sus componentes, mensajes (acciones y actuadores) y atributos

```
Class: Patrolling
  SubclassOf: Component and (interpretMessage some Patrol)
                  and (isComponentOf only (hasComponent some
                              (interpretMessage some MoveTo)))
```

Si nos fijamos, las dependencias que existen no son directamente entre componentes (un componente no requiere a otro componente específico), pero tampoco entre mensajes (no es que en la definición de un mensaje se requiera la ejecución de otro para funcionar). La dependencia existente es que un componente, para poder realizar una acción, puede requerir que otras acciones se lleven a cabo por la entidad (otros componentes). En el ejemplo, para llevar a cabo la acción de *patrulla* el componente debe pertenecer a una entidad que tenga un componente capaz de ejecutar mensajes de tipo *ir-a*. El mensaje de tipo *ir-a* podría ser implementado de diferentes maneras por distintos componentes, de forma que, por ejemplo, el componente de *vuelo* podría llevar a cabo la acción por si mismo, pero el componente de *movimiento* por tierra podría requerir de la colaboración de otros componentes para encontrar la

mejor ruta entre dos puntos de la patrulla (ya que en tierra suponemos que hay muchos más obstáculos y no se puede ir en línea recta).

En cierta manera este tipo de dependencias recuerdan a conceptos que habitualmente aparecen en planificación de dominios (Ghallab et al., 2004), donde el componente tiene un objetivo que cumplir y para ello describe una serie de subobjetivos que se deben llevar a cabo por otros componentes de la misma entidad. Lo importante en esta situación es que, cuando el programador está implementando un componente que depende de otros, nadie puede garantizarle que todas las entidades que van a terminar teniendo ese componente vayan a tener también esos componentes que implementen las acciones declaradas como subobjetivos. Este tipo de dependencias no suelen estar reflejadas en ningún documento ni estructura, por lo que es habitual que cuando un programador cree una nueva entidad (simplemente añadiendo componentes en un fichero de texto) se olvide de algún componente que era necesario para el correcto funcionamiento de otro.

Cuando se trabaja con una arquitectura tradicional basada en herencia, este tipo de inconsistencias son fácilmente detectadas debido al *feedback* que nos proporciona el compilador. Cuando una entidad no declara un método que es invocado por otro método de la misma entidad, salta un error en tiempo de compilación. Sin embargo, en una arquitectura basada en componentes, cuando un componente envía un mensaje a la entidad a la que pertenece y no hay un componente hermano que pueda procesar dicho mensaje, ningún tipo de error es lanzado en tiempo de compilación. Incluso, en muchas implementaciones ni siquiera es lanzado en tiempo de ejecución, simplemente la funcionalidad esperada no se ejecuta. Debido a está ausencia de un error explícito, la tarea a depuración es especialmente dura y es difícil averiguar cual es la razón del fallo.

5.3.2.2. Relacionadas con los atributos

En el apartado anterior nos hemos referido a las inconsistencias que pueden existir al definir las entidades en función de los componentes que la componen, ya que estos establecen dependencias en sus descripciones. En una representación de grano fino, las entidades no se definen sólo a partir de sus componentes sino que estas entidades tendrán unos atributos (gracias a la descripción de los componentes) a los cuales el programador o el diseñador querrán darles valores iniciales.

Durante la creación de la entidad, además de ensamblarse todos los componentes, también se proporciona una lista de pares atributo-

valor, que inicializarán los atributos de los componentes. Estos pares de atributo-valor que se establecen para las entidades surgen de una mezcla de los valores por defecto que se establecen durante la definición del dominio con los valores específicos que se le han dado a una instancia de entidad en un fichero de mapa. Durante la inicialización de estos atributos la entidad asume que los valores de éstos son consistentes: dichos valores se corresponden con el tipo de datos esperado (y en nuestro dominio se asumirían ciertas restricciones extra). Aunque este problema no es particular de la arquitectura basada en componentes, sino que existe también en otras, es una gran fuente de inconsistencias que nuevamente sólo se detectan en tiempo de ejecución, que es cuando se asignan los valores. Lo que aún complica más la depuración en la arquitectura basada en componentes es que, aunque los atributos se definen para la entidad, en ejecución son los componentes lo que requerirán a las entidades ciertos atributos que la entidad podría no tener.

Además de poder tener inconsistencias debidas al tipo de datos de los atributos, nuestro dominio permite definir restricciones extra sobre dichos tipos lo que a su vez incrementa las posibles inconsistencias. Un ejemplo de atributo básico podría ser aquél que mide la vida de un personaje, que en nuestro caso sería un simple valor entero:

```
DataProperty: health
  SubPropertyOf: BasicAttribute
  Range: integer
```

Como se especifica en el Capítulo 4, el escenario se puede complicar un poco ya que los atributos de las descripciones ontológicas pueden incluir restricciones de rango. Por ejemplo, un atributo real que represente la puntería podríamos querer que tome un valor entre cero (incluido) y uno (no incluido). Su representación sería:

```
DataProperty: aim
  SubPropertyOf: BasicAttribute
  Range: float[>= 0f, <1f]
```

También se pueden definir los únicos posibles valores que puede tomar un atributo. Supongamos un atributo que configure el tipo de entidad física que va a estar asociado a la entidad, éste atributo podría tomar tres valores diferentes que definan si la entidad es *estática*, *dinámica* o *cinemática*:

```
DataProperty: physicsType
  SubPropertyOf: BasicAttribute
  Range: string and {"static","dynamic", "kinematic"}
```

En los casos anteriores mostrábamos ejemplos de atributos de tipos básicos, pero podemos también definir atributos que hagan referencia a otros elementos del dominio como, por ejemplo, un atributo que marca el *principio de ruta* puede ser instanciado como una entidad *WayPoint* concreta:

```
Class: routestart
  SubclassOf: Attribute
  EquivalentTo: WayPoint
```

La declaración de atributos de por sí no presenta inconsistencias, éstas vienen cuando queremos asignar valores específicos a estos atributos en las entidades. Para representar en OWL esta asignación de atributos lo que tenemos es un individuo por entidad, que tendrá sus diferentes elementos y los valores específicos para cada uno de los atributos. Un ejemplo que aúna los atributos anteriormente descritos sería la clase *personaje*:

```
Individual: iCharacter
  Types: Character
  Facts: ...
         health "50"^^unsignedInt,
         aim "0.6f",
         physicsType "kinematic",
         hasAttribute iroutestart,
```

En este punto, con estas descripciones de elementos del dominio, podemos controlar inconsistencias a nivel del dominio del juego, comprobando los valores por defecto que se establecen para los atributos. Las comprobaciones de consistencia de nivel, para los valores que los diseñadores asignan a instancias concretas de entidades, son explicadas posteriormente en la Sección 5.3.3.

5.3.2.3. Ejemplo de detección de inconsistencias a nivel de entidad

Con la intención de validar la detección de inconsistencias hemos desarrollado un pequeño ejemplo de juego en *Unity* usando nuestra metodología. El ejemplo consiste en un simple juego de acción 3D donde el jugador maneja un avatar y el sistema controla varios NPC de diferentes clases, que pueden hacer diferentes cosas como proteger un área, explorar el entorno o buscar al jugador. Los diseñadores han generado un dominio de juego definiendo las entidades necesarias en función

de acciones, sensores y atributos y poniéndolas en una jerarquía. Con esta información los programadores han completado el dominio y han repartido dicha funcionalidad y atributos en componentes software.

Ciertos detalles del juego ya han sido dados en los apartados anteriores. Más específicamente tenemos la distribución de entidades y componentes en la Figura 5.7, un ejemplo de entidad en la Figura 5.8 y diversos elementos representados en OWL. Las entidades y componentes que aparecen en color gris en la Figura 5.7 solo son relevantes desde el punto de vista semántico, lo que quiere decir que no existirán entidades de este tipo en el juego final, en éste solo aparecerán los elementos de color negro.

Centrándonos ya en el ejemplo, veamos una definición de una entidad aparentemente inofensiva como NPC y alguno de sus componentes, acciones y atributos (Figuras 5.7 y 5.8). Cuando los diseñadores y programadores definen los elementos del dominio, éste se almacena en OWL, donde podemos encontrarnos definiciones como las que se muestran a continuación.

```
Class: NPC
  SubclassOf: Entity
    and iaA Character
    and (hasComponent some FSM)
    and (hasComponent some Patrolling)
    and (hasComponent some AnimationMgr)
    and (hasComponent some Controller)
    and (hasComponent some Graphic)
    and (hasComponent some Health)
    and (hasComponent some Physics)
    and (hasComponent some Shooter)
    and (hasComponent some Transform)
    and (interpretMessage some MovementFinished)
    and (interpretMessage some Patrol)
    and (interpretMessage some Attack)
    and (interpretMessage some ThrowObject)
    and (interpretMessage some Jump)
    and (interpretMessage some RunTo)
    and (interpretMessage some WalkTo)
    and (interpretMessage some Turn)
    and (interpretMessage some Wander)
    and (interpretMessage some PlayAnimation)
    and (interpretMessage some StopAnimation)
    and (interpretMessage some SetPosition)
```

```
    and (interpretMessage some CollisionReceived)
    and (fsmfile some string)
    and (health some unsignedint)
    and (aim some float[>= 0f, <1f])
    and (physicsType some string and
                     {"static","dynamic","kinematic"})
    and (meshFile some string)
    and (hasAttribute some routestart)
    and (hasAttribute some routeend)
    and (hasAttribute some bulletType)
    and (isNotInstantiable)
```

La entidad *NPC* define un *personaje* (*Character*) que tiene dos habilidades extra: poder *patrullar* y ejecutar una *máquina de estados* que controla a la entidad. Esto se traduce en que la entidad puede ejecutar dos tipos de mensajes nuevos y tiene varios atributos nuevos. Esta entidad no es instanciable, lo que significa que no se podrán crear entidades de este tipo en un mapa pero que es importante desde el punto de vista semántico, para que luego hereden entidades que comparten ciertas características. Unos de los ejemplos que podemos ver como componente es el de patrulla, que se almacena de la siguiente manera:

```
Class: Patrolling
  SubclassOf: Component
    and (interpretMessage some MovementFinished)
    and (interpretMessage some Patrol)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some MoveTo)))
    and (hasAttribute some routestart)
    and (hasAttribute some routeend)
    and (isInstantiable)
```

Este componente es el encargado de patrullar entre dos puntos del nivel cuando se le envía el mensaje de *patrulla*. Sin embargo, éste es un buen ejemplo de un componente que no puede realizar la funcionalidad por sí mismo sino que parte de dicha funcionalidad se delega en otros componentes. En este caso, el componente decide hacia donde moverse, pero la implementación de ese *movimiento* la realizará otro componente. De esta manera el movimiento de las entidades que patrullen puede ser totalmente diferente, pero todas transitaran por los puntos especificados. El mensaje que el componente mandará para poder realizar los movimientos es:

```
Class: MoveTo
  SubclassOf: Message
    and (target some vector3)
    and (isInstantiable)
```

Donde lo interesante del mensaje es el envío de la posición a donde se debe dirigir.

Finalmente, para poder asignar valores por defecto a los atributos tenemos un individuo *iNPC* con la definición de habilidades y los valores de los atributos que asignaron los programadores y diseñadores.

```
Individual: iNPC
  Types: NPC
  Facts: isA iCharacter,
         hasComponent iAnimationMgr,
         hasComponent iController,
         hasComponent iFSM,
         hasComponent iGraphic,
         hasComponent iHealth,
         hasComponent iPatrolling,
         hasComponent iPhysics,
         hasComponent iShooter,
         hasComponent iTransform,
         hasAttribute ibulletType,
         fsmfile "basicBehaviour.xml",
         health "150"^^unsignedInt,
         aim 3f,
         physicsType "typical",
         meshFile "Lerpz"
```

Según la definición es fácil ver cuáles son las habilidades de la entidad, pero es difícil de apreciar es si la entidad presenta alguna de las inconsistencias definidas en las secciones anteriores (sobre todo si no se miran el resto de definiciones OWL). En la descripción de *iNPC* hay varias inconsistencias que debemos detectar y presentar al usuario:

- La ejecución del mensaje *Patrol* en el componente *Patrolling* depende de que otro componente pueda ejecutar mensajes de tipo *MoveTo*, pero no hay ningún componente en la entidad capaz de desempeñar la funcionalidad de mover la entidad de un punto a otro. Para ser más específicos, la cláusula violada es la siguiente:

```
iNPC subClassOf (isComponentOf only
                   (hasComponent some
                     (interpretMessage some MoveTo)))
```

ya que no se puede inferir que ninguno de los componentes que tiene la entidad *NPC* pueda interpretar mensajes de tipo *MoveTo*. En este caso, al *iNPC* ser solo una entidad conceptual y no instanciable podríamos mostrar un simple aviso (por si el programador lo quiere tener en cuenta) pero no sería un error en sí mismo si las entidades que se pueden instanciar como *FlyingCreature* o *Indigenous* tienen un componente que sí que interpreta el mensaje *MoveTo*. En este caso, las entidades tienen los componentes *Fly* y *Walk* correspondientemente, así que esas entidades no son inconsistentes ya que ambos componentes implementan la funcionalidad de *movimiento*.

- El atributo de vida *health* tiene un valor *unsignedInt* mientras que el valor esperado es un entero (Sección 5.3.2.2). En este caso nos dice que la ontología es inconsistente ya que el literal *"150"^^unsignedInt* no pertenece al tipo de datos de los enteros.
- El atributo *aim* que muestra la puntería de una entidad toma para el *NPC* el valor de 3.0, lo cual supone un problema ya que se había especificado (Sección 5.3.2.2) que su valor oscilaba entre 0 y 1. En este caso volvemos a tener un problema al violar el rango de definición del atributo con el valor asignado.
- Por último, el atributo que define el tipo de entidad física *physicsType* (Sección 5.3.2.2) presenta un error similar a los anteriores ya que los valores que puede tomar este atributo están limitados a *static,dynamic* o *kinematic*, pero nunca *typical*. Nuevamente se vuelve a violar el rango que se había definido para el atributo.

En el Capítulo 6 se muestra una implementación más concreta de esta técnica, con un razonador específico funcionando sobre una herramienta que soporta nuestra metodología.

5.3.3. Inconsistencias debidas a la configuración del nivel

Como se comenta en el Capítulo 4, los diseñadores son responsables de tres tareas principales en el diseño de un nivel.

- Crear el *entorno* del nivel usando mallas estáticas como paredes, pasillos, terrenos, árboles, edificios o lo que sea que el juego necesita.

- Añadir al nivel interactividad mediante la adición de entidades. A partir de la definición ontológica realizada por los diseñadores y completada por los programadores, éstos últimos habrán generado *prefabs* o entidades prototípicas de las entidades acordadas con los componentes necesarios para realizar su funcionalidad. Los diseñadores deberán añadir estas entidades por el nivel y ajustarlas, modificando sus parámetros por defecto cuando sea necesario o, incluso, añadiendo componentes que añadan nueva funcionalidad a ciertas entidades, para probar conceptos que se puedan incorporar en futuras iteraciones.

- Definir los comportamientos de los personajes mediante el uso de técnicas como máquinas de estado (FSM) o árboles de comportamientos (BT) o cualquier otra técnica de inteligencia artificial que sea de uso sencillo para los diseñadores. Estos comportamientos deben ser luego incorporados en un componente específicamente creado para ejecutar ese tipo de ficheros.

En el apartado anterior (Sección 5.3.2) hemos visto como el modelo especificado contiene información acerca de las habilidades de una entidad (que acciones puede realizar como *atacar* o *ir-a*), sensores (percepción como *enemigo perdido* o *daño sufrido*) y estado (atributos como *salud* o *tipo de vida*). Los prototipos de entidad que se crean y se almacenan en esa representación OWL serán usados en todos los niveles del juego y se considera como el *modelo estático* del dominio. Sin embargo, el dominio formal *no* contiene *cuándo* una entidad usa esos recursos, *cuándo* ataca o *cuándo* pierde daño ya que esa información depende del nivel concreto y de su dificultad. Cuando se crea un nuevo nivel, los diseñadores crean nuevas entidades, cambian ciertos valores de atributos y crean inteligencias artificiales para los personajes no controlados por el jugador. Toda esta información debe ser integrada también en nuestra representación OWL para chequear la *consistencia* del nivel y constituye *modelo dinámico* del dominio.

Las entidades por lo tanto son las piezas con las que se construye el nivel y los diseñadores de juego deben crear comportamientos concretos para esas entidades, usando por ejemplo un editor de FSM. Para comprobar la ausencia de inconsistencias en los FSM realizados por los diseñadores tenemos que traducirlos, al menos parcialmente, en OWL. Nuestra propuesta es crear dinámicamente un nuevo componente ficticio en OWL por cada definición de FSM, recogiendo en ellos la información que sea relevante desde el punto de vista semántico para la detección de posibles inconsistencias. Una FSM desde el punto de vista del

diseño de comportamientos es, simplificando, un conjunto de estados, donde se realizan acciones, y una serie de transiciones que, en función de eventos externos o de consulta a tributos, permiten el cambio entre estados.

La pregunta entonces es: ¿Qué es lo que nos interesa reflejar desde el punto de vista semántico para garantizar la consistencia?. Los estados y transiciones en sí mismas no nos interesan, lo que nos interesa es que se puedan llevar a cabo las acciones reflejadas en esos estados y que se garantice que las diferentes transiciones pueden tener lugar. El componente *FSM* no tiene más función que esa, ejecutar la FSM, por lo tanto deberá confiar que el resto de componentes de la entidad le permitan realizar las acciones y que definan los estados.

En primer lugar, para comprobar que el componente ficticio que crearemos puede transitar entre estados, tenemos que definir tanto las percepciones o eventos recibidos como los atributos que se van a consultar. Las percepciones que se recibirán llegarán en forma de mensajes por lo que de alguna manera, este nuevo componente, será capaz de interpretar esos mensajes. Por otro lado, los atributos que se consulten desde las transiciones deberán estar presentes en la entidad a la que pertenezca el nuevo componente por lo que se pondrá como restricción (ya sean atributos simples o complejos). Por último, habrá que asegurarse que las acciones que se encuentran en los estados pueden ser llevadas a cabo por otros componentes de la entidad. Estas acciones serán invocadas por el componente ejecutor de FSM mediante el envío de un mensaje a la entidad a la que pertenece. Por tanto, añadiremos una restricción al componente que indique que sólo se mantiene la consistencia si este tipo de componente pertenece a una entidad en la que haya otros componentes que puedan llevar a cabo todas las acciones que se encuentran en los estados. De una manera más esquemática podemos ver a continuación cómo se definiría una FSM en OWL como un componente ficticio:

```
Class: FSMName
  SubclassOf: Component
    and (interpretMessage some event/sense)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some action)))
    and (isComponentOf only
          (basic atributte some type))
    and (isComponentOf only
          (hasAttribute some complex attribute))
```

Este mismo proceso podría ser llevado a cabo con otro tipo de técnicas de inteligencia artificial como por ejemplo los *BT* ya que, al fin de cuentas, lo único que hace falta saber es que acciones se han de realizar (aquellas que se encuentran en las hojas de los BT) y que elementos van a producir cambios en la ejecución del comportamiento (en este caso condiciones y consultas realizadas en las guardas de los BT).

Por otro lado, los diseñadores pueden sentirse tentados a construir sus propias entidades juntando varios componentes de los que han creado los programadores en vez de usar sólo las entidades previamente aceptadas y generadas (*prefabs*). Crear nuevas entidades durante el prototipado no es necesariamente algo malo ya que proporciona a los diseñadores una manera de experimentar y probar nuevas ideas sin que eso afecte al dominio formal del juego. Desafortunadamente, cuando se crean entidades directamente en el editor de niveles, las dependencias que existen entre los componentes y sus atributos no son tenidas en cuenta y pueden ser una gran fuente de inconsistencias, llevando al diseñador a la frustración ya que no tiene por qué entender acerca de dichas dependencias. Los componentes son artefactos útiles para los programadores, pero posiblemente demasiado técnicos para los diseñadores. La idea es ofrecer apoyo también a los diseñadores en esta fase, mirando si hay inconsistencias en el nivel, analizando las nuevas entidades en busca de inconsistencias.

Debe recalcarse que, una vez el diseñador haya creado una entidad interesante para el juego, debe ser incorporada al dominio formal del juego para que en la siguiente iteración este tipo de entidades sea un *prefab* y que su descripción semántica pueda ser aprovechada en otras facetas del juego. Las nuevas entidades pueden provocar refactorizaciones de código por lo que deberán ser aprobadas tanto por los diseñadores (que las proponen) como por los programadores.

De hecho, la asistencia proporcionada para la detección de inconsistencias es útil desde el principio del desarrollo ya que de esta manera los diseñadores pierden el miedo a realizar cambios en los mapas ya que las entidades prototípicas *antiguas* (de iteraciones pasadas) son automáticamente actualizadas teniendo en cuenta nuevas distribuciones de componentes, acciones o atributos. Los valores de los atributos son reevaluados teniendo en cuenta los valores establecidos en el nivel y las FSM son también evaluadas nuevamente contra las nuevas versiones de las entidades. En éstas se prueba si aún siguen pudiendo llevar a cabo todo tipo de acciones y sensores ya que, de una a otra actualización del dominio, hay entidades que pueden haber perdido cualidades que aun se sigan sin embargo usando en una FSM.

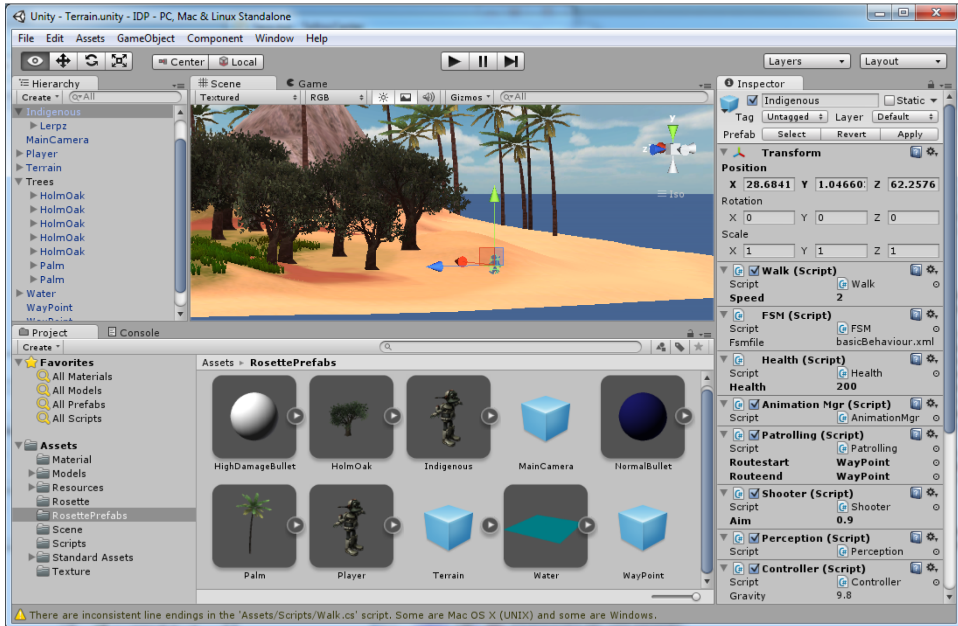

Figura 5.9: Ejemplo de un nivel con el dominio implementado en Unity

Todo esta comprobación de inconsistencias (similar en espíritu al *desarrollo dirigido por test*) hacen que tanto para programadores como para diseñadores sea posible llevar a cabo un desarrollo basado en metodologías ágiles. En cierto modo, el diseño del juego es inmune a cambios y adiciones durante los ciclos o iteraciones.

5.3.3.1. Ejemplo de detección de inconsistencias a nivel de mapa

Retomando el ejemplo que se comenzó en la Sección 5.3.2.3, veamos como continua el flujo de desarrollo cuando los programadores terminan de modelar el dominio que allí empezaron. Una vez que los programadores han corroborado que el dominio es consistente generan el contenido necesario para la creación del juego. En este caso, donde el juego estará implementado en *Unity*, los programadores generarán la funcionalidad necesaria en forma de *scripts* (componentes) y montan los prototipos de entidad en forma de *prefabs* (ayudados con la generación de contenidos explicada en la Sección 5.2). La Figura 5.9 muestra el nivel que, a partir de los recursos generados por los programadores, han montado los diseñadores usando *prefabs* o entidades creadas al vuelo juntando componentes. El la parte derecha de la figura, en el inspector, se ve un personaje con los componentes/*scripts* que los programadores han creado.

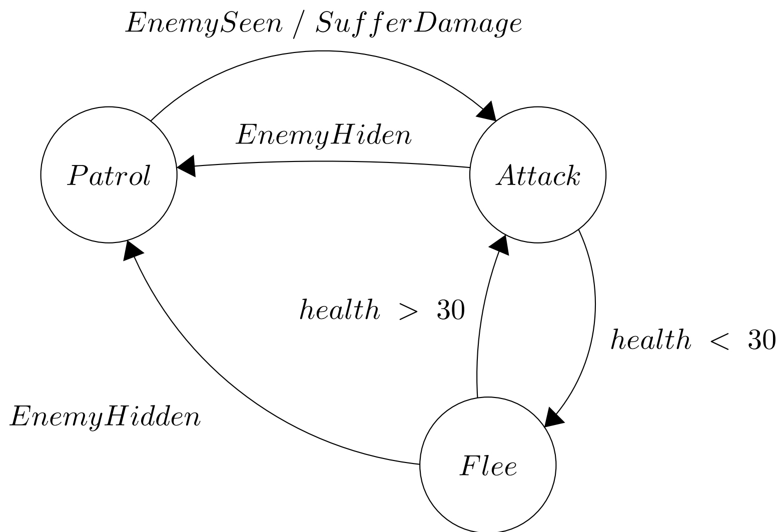

Figura 5.10: Simple FSM

Para el personaje concreto que aparece en el inspector de *Unity*, el diseñador ha creado un simple FSM (Figura 5.10) en un editor externo, para determinar como debe comportarse esta entidad concreta durante la ejecución del juego. La FSM creada mantiene al NPC patrullando por la escena y, cuando recibe un ataque o ve a un enemigo, su comportamiento cambia y ataca a dicho enemigo. Si, estando atacando pierde la pista del enemigo, el NPC retorna a su comportamiento de patrulla pero si durante el combate su vida se reduce por debajo de 30, el NPC pasa a un estado de huida. En ese estado de huida hay dos posibilidades, si, durante la huida, el personaje recobra vida vuelve al estado de ataque pero si logra su objetivo y pierde de vista al enemigo entonces regresa a su estado de patrulla. Por otro lado, en el editor de *Unity*, se puede apreciar como el diseñador de niveles ha modificado algunos valores por defecto de los añadidos en la definición del dominio. Por ejemplo, uno de los cambios ha sido cambiar el valor de la velocidad de 0.5 a 20, para personalizar la entidad y adaptar sus valores al comportamiento asociado.

El diseño de niveles se compone de diversas tareas, donde los diseñadores deben expresar su talento poniendo diferentes piezas juntas. Al combinar piezas de funcionalidad muy diferentes, la edición de niveles suele causar un gran número de inconsistencias:

- Durante el diseño del dominio, diseñadores y programadores crean atributos que serán añadidos a entidades y componentes. Un atri-

buto es definido como un tipo básico o complejo y puede tener alguna restricción como que un atributo acepte sólo valores dentro de un rango. Por ejemplo, en nuestro ejemplo el rango del atributo *speed* puede adoptar sólo valores de 0.0 a 1.0:

```
DataProperty: speed
  SubPropertyOf: BasicAttribute
  Range: float[>= 0f, <1f]
```

- Montar entidades *al vuelo*, poniendo juntos una serie de componentes, puede provocar fallos durante la ejecución del juego debido a que hay componentes que requieren otros componentes para trabajar correctamente. Por ejemplo, el componente de *patrulla* necesita que otro componente sea capaz de desplazar la entidad, por lo que la entidad requiere componentes que acepten mensajes de tipo *MoveTo* tales como *Fly* o *Walk*.

- Los diseñadores crean FSMs y las asocian a entidades concretas, lo cual puede causar inconsistencias y errores similares a los del punto anterior. Una FSM habitualmente necesita ejecutar acciones en sus estados y transitar en función de eventos que emite la entidad, gracias a sus sensores, y de consultas a atributos de la propia entidad. Un diseñador puede asociar sin darse cuenta un FSM a una entidad que no sea capaz de ejecutar alguna de esas acciones o de transitar correctamente a algún estado. La FSM de la Figura 5.10 necesita ejecutar las acciones *Patrol*, *Attack* y *Flee* por lo que al menos un componente de la entidad debería ser capaz de llevar a cabo dichas acciones y a su vez garantizar que se pueden llevar a cabo las transiciones.

Este tipo de inconsistencias causan terribles dolores de cabeza a los diseñadores debido a la dificultad de encontrar dónde está el problema. Como los problemas suceden durante la ejecución del juego, el *feedback* es inexistente y a veces el problema es causado por razones técnicas que los diseñadores deberían obviar. Debido a esto, proporcionamos la posibilidad de que los diseñadores chequeen si el nivel es consistente con el dominio formal especificado anteriormente. Previamente se ha explicado qué parte del dominio es *estática* (aquella relacionada con atributos, mensajes, componentes y entidades) y qué parte es *dinámica*, creada en función de un nivel concreto (relacionada con las FSM, cambios de valores de atributos y las entidades creadas *al vuelo*).

Los ejemplos de la parte estática del dominio han sido en su mayoría ya dados en la Sección 5.3.2.3. Relacionado con la parte dinámica del

dominio OWL, a continuación vemos un ejemplo de una FSM creada por el diseñador (Figura 5.10). La FSM se representa como un componente que define los eventos que disparan transiciones de estado en la FSM como mensajes que pueden ser llevados a cabo por el componente ficticio. En este caso la FSM reacciona a *EnemySeen* y *EnemyHidden* pero, cuando estos eventos suceden, la FSM tiene que ejecutar una acción que debería ser llevada a cabo por otro componente por lo que, en este caso, la entidad debe contener componentes que sean capaces de ejecutar las acciones *Patrol*, *Attack* y *Flee*. Las transiciones también se pueden llevar a cabo debido a la consulta de atributos por lo que se debe reflejar que la entidad, a la que pertenecerá este componente, debe tener un atributo de tipo *health*.

```
Class: SimpleFSM
  SubclassOf: Component
    and (interpretMessage some EnemyHidden)
    and (interpretMessage some EnemySeen)
    and (interpretMessage some SufferDamage)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some Patrol)))
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some Attack)))
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some Flee)))
    and (isComponentOf only (health some integer))
    and (isComponentOf only (fsmfile some string))
    and (isInstantiable)
```

Como permitimos que los diseñadores compongan sus entidades con componentes existentes y también que creen las FSM que controlan a los personajes (los cuales en OWL son simplemente componentes) debemos generar nuevas definiciones de entidades de manera dinámica. Un ejemplo de entidad creada dinámicamente sería aquella que incorpora la FSM que acabamos de definir.

```
Class: NPCWithSimpleFSM
   SubclassOf: Indigenous
    and (hasComponent some SimpleFSM)
    and (hasComponent some FSM)
    and (hasComponent some Patrolling)
    and (hasComponent some AnimationMgr)
    and (hasComponent some Controller)
    and (hasComponent some Graphic)
    and (hasComponent some Health)
    and (hasComponent some Physics)
    and (hasComponent some Shooter)
    and (hasComponent some Transform)
    and (interpretMessage some MovementFinished)
    and (interpretMessage some Patrol)
    and (interpretMessage some Attack)
    and (interpretMessage some ThrowObject)
    and (interpretMessage some Jump)
    and (interpretMessage some RunTo)
    and (interpretMessage some WalkTo)
    and (interpretMessage some Turn)
    and (interpretMessage some Wander)
    and (interpretMessage some PlayAnimation)
    and (interpretMessage some StopAnimation)
    and (interpretMessage some SetPosition)
    and (interpretMessage some CollisionReceived)
    and (fsmfile some string)
    and (health some unsignedint)
    and (aim some float[>= 0f, <1f])
    and (physicsType some string and
                        {"static","dynamic","kinematic"})
    and (meshFile some string)
    and (hasAttribute some routestart)
    and (hasAttribute some routeend)
    and (hasAttribute some bulletType)

    and (isInstantiable)
```

La *SimpleFSM* se asignó durante el diseño de un nivel a un individuo de la clase *Indigenous* que, a su vez, especializa *NPC* permitiéndole desplazarse por tierra mediante el uso del componente *Walk*. Esta y otras definiciones de la misma índole pasan a completar el dominio estático y, con la mezcla de ambas definiciones, comprobamos si el dominio es

consistente o si, como es el caso, viola alguna propiedad:

- El atributo *speed* ha recibido en el editor de niveles un valor no permitido ya que se sale del rango especificado anteriormente (de 0.0 a 1.0). Como el diseñador de niveles fijó el valor a 2.0, el detector de inconsistencias le informa del error para que ponga un valor adecuado.

- La FSM asociada al personaje no se podrá ejecutar ya que dicho personaje debería tener otros componentes que puedan ejecutar las acciones que se encuentran descritas en los diferentes estados. La entidad es capaz de llevar a cabo la acción *Patrol*, mediante el componente *Patrolling*, que a su vez puede realizar la acción gracias a que hay otro componente *Walk* en la entidad que puede realizar acciones de tipo *MoveTo* (Como se definió en la representación del componente *Patrolling* en la Sección 5.3.2.3, dicho componente necesita de la existencia de otro que lleve a cabo acciones *MoveTo*). El NPC también puede realizar la acción *Attack* mediante el componente *Shooter*, sin embargo no hay ningún componente en esa entidad que pueda llevar a cabo la acción *Flee*. En este caso simplemente se le informaría al diseñador de niveles que debe añadirse algún componente que pueda llevar a cabo la acción de huida y recomendaría la adición del componente *RunAway*, el único capaz de realizar la acción *Flee* en nuestra colección a estas alturas del desarrollo.

- Habría que tener en cuenta también que en la FSM se pueda transitar. En este caso no hay problemas ya que el componente de percepción *Perception* generaría los eventos necesarios (*EnemySeen* y *EnemyHidden*), se nos informaría también de que se ha recibido un daño (*SufferDamage*) y, el atributo que se consulta (*health*) se encuentra en la entidad ya que ésta tiene un componente llamado *Health* que lo declara.

El *feedback* proporcionado es muy útil para los diseñadores de niveles no solo cuando crean nuevos niveles sino también para comprobar si antiguos niveles siguen siendo consistentes tras una refactorización del dominio del juego. De esta manera se promueve el uso de metodologías ágiles simplificando los problemas entre iteraciones.

5.3.4. Conclusiones

En esta sección se ha detallado cómo, a partir de una representación ontológica y con ayuda de un razonador, detectamos inconsistencias en

el dominio de las entidades de juego y en los niveles creados por los diseñadores, donde sitúan instancias concretas de dichas entidades y les asignan algún tipo de inteligencia artificial dirigida por datos.

En el trabajo previo a estas técnicas, los componentes autoreflexivos (Llansó et al., 2009b; Sánchez-Ruiz et al., 2009a; Llansó et al., 2009a), el trabajo estaba justificado, agilizando el desarrollo al anticipar posibles errores y al ofrecer *feedback* al usuario acerca de que cosas pueden ser un problema en tiempo de ejecución. Al tener ahora un dominio mucho más rico en conocimiento, el número de inconsistencias que detectamos son mucho mayores, ya que no se quedan a nivel del componente y sus mensajes, y, además, supone menos esfuerzo ya que los programadores no tienen que realizar un esfuerzo extra creando nuevos métodos en los componentes que sirvan para que estos se auto describan. Esa (y más) información se encuentra en la ontología, por lo que la aplicación de estas técnicas incluso se anticipa más aun. Ya no se detectan las inconsistencias cuando se programan los componentes software sino cuando los programadores diseñan su distribución.

En la Sección 6.4 se comenta una implementación de código concreta de esta técnica.

Capítulo 6

Rosette

En este capítulo se presenta *Rosette*, una herramienta de autoría visual desarrollada con las ideas presentadas en esta tesis. *Rosette* soporta la metodología propuesta en el Capítulo 4 y aquí se presenta que decisiones se han tomado para la implementación. Además se explica cómo y con qué tecnología se han implementado las técnicas explicadas en el Capítulo 5.

6.1. Una herramienta de autoría para un desarrollo ontológico

Con el fin de poder probar la metodología y técnicas propuestas en capítulos anteriores, se ha desarrollado *Rosette*, una completa herramienta visual de modelado ontológico que soporta todas las ideas anteriormente presentadas. El desarrollo de la herramienta ha sido realizado en *Java* sobre la plataforma de cliente enriquecido de Eclipse (*Eclipse RCP*)[1]. En cierta manera *Rosette* introduce una metodología dirigida por modelo o, más específicamente dirigida por ontologías (Tetlow et al., 2006; Deline, 2006), que permite el desarrollo de un juego por fases mediante el uso de un dominio formal. Además presenta otras características, de las cuales podríamos considerar como principales:

- **Facilidad de modelado del dominio de juego**: Mediante diferentes vistas de la ontología, los diseñadores y programadores pueden representar los diferentes elementos de juego de una manera intuitiva que, además, ofrece mecanismos para evitar duplicidades y promover la creación de una distribución más flexible y reutilizable.

[1] `http://wiki.eclipse.org/Rich_Client_Platform`

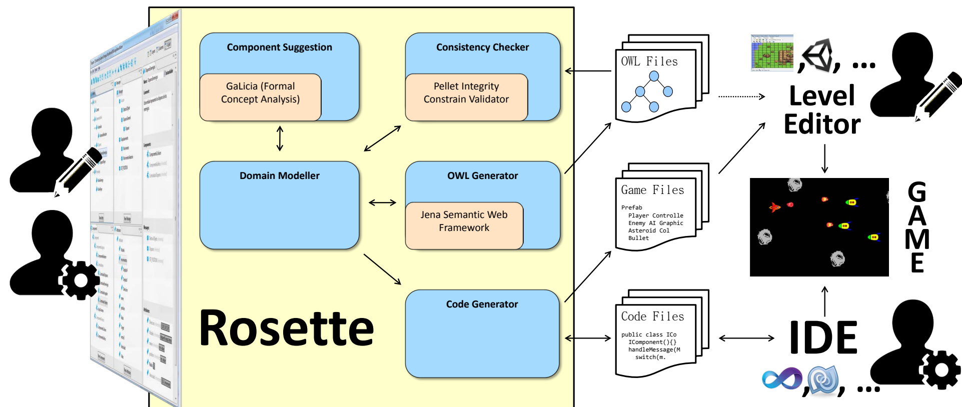

Figura 6.1: Vista general de la arquitectura de *Rosette*

- **Conocimiento semántico almacenado en un dominio formal**: El dominio formal se almacena de manera transparente a los usuarios en una representación rica en conocimiento, en concreto en OWL[2].

- **Extracción automática de componentes**: Mediante el uso de análisis formal de conceptos se sugiere la mejor distribución de funcionalidad en componentes para una jerarquía de entidades especificada por los diseñadores y programadores (Sección 5.1).

- **Generación procedimental de contenido**: A través del uso de plantillas, se permite la generación de contenido para diferentes motores (Sección 5.2). Como ejemplo se han creado plantillas para dos motores sustancialmente distintos. El primero es *Unity*[3], un motor comercial que usa scripts en C#, y el segundo se trata de un pequeño motor desarrollado en C++ para el máster de videojuegos de la Universidad Complutense de Madrid[4].

- **Chequeo de inconsistencias en el dominio de juego**: Permite validar los dominios y niveles que realizan los diseñadores y los programadores, anticipando durante la fase de diseño e implementación errores que pueden surgir durante la ejecución del juego (Sección 5.3).

- **Metodologías de desarrollo ágil**: Permite y promueve metodologías software que impliquen técnicas de prototipado o desarrollos

[2]http://www.w3.org/TR/owl-features/
[3]http://unity3d.com/unity/
[4]http://www.videojuegos-ucm.es/

iterativos, lo cual es ampliamente usado en la creación de videojuegos. En este sentido se permiten refactorizaciones en el dominio formal que automáticamente lanzan refactorizaciones de código mediante la generación de contenido.

En la Figura 6.1 se presenta un diagrama de *Rosette* y sus diferentes módulos. El trabajo empieza con el diseñador trabajando con el *modelador del dominio* a través de la interfaz gráfica. Durante este uso el módulo de *generación de OWL* (Sección 6.1) se crea el dominio formal y dicho dominio formal es usado por el módulo de *chequeo de consistencia* (Sección 6.4) para ofrecer *feedback* al diseñador. En una segunda fase, el programador recoge el dominio creado por los diseñadores y, mediante una vista que presenta una representación del dominio de más bajo nivel, completa el modelo formal usando los mismo módulos y también el módulo de *sugerencia de componentes* (Sección 6.2), que propone una distribución completa de la funcionalidad del dominio actual.

Una vez que se ha generado o actualizado el dominio en *Rosette* por los diseñadores y programadores, el módulo de *generación de contenido* crea de manera procedimental contenido para una de las plataformas soportadas. Actualmente se soportan dos; en la Sección 6.3 se describen los pasos para añadir más. Por último, ya fuera de *Rosette*, los programadores deberán completar los ficheros de código generado y los diseñadores realizar los niveles de juego, para los cuales podrán usar el módulo de *chequeo de consistencia* que los valide.

Rosette es una herramienta que permite un modelado intuitivo mediante el uso de *arrastrar y soltar*. La vista que tienen los diseñadores del dominio puede verse en la Figura 6.2, donde el espacio se distribuye en 4 regiones. A la izquierda se encuentra la jerarquía de entidades, donde los conceptos en gris hace referencia a entidades que tienen sentido desde el punto de vista semántico pero que no estarán presentes en la implementación final del juego (las llamadas entidades no instanciables). En el centro de la aplicación se encuentran las jerarquías de acciones y sensores y la de atributos donde, al igual que en la jerarquía de entidades, los conceptos en gris representan elementos no instanciables en el juego, aunque en este caso si podrían formar parte de la implementación final en herencia de clases. Por último, a la derecha de la aplicación, se muestra la descripción del elemento que se seleccione en alguna de las jerarquías anteriores.

En el caso de una entidad (como la que se aprecia en la imagen) ésta está descrita en función de las acciones o sensores que es capaz de llevar a cabo así como de los atributos que le ayudan a describir su estado,

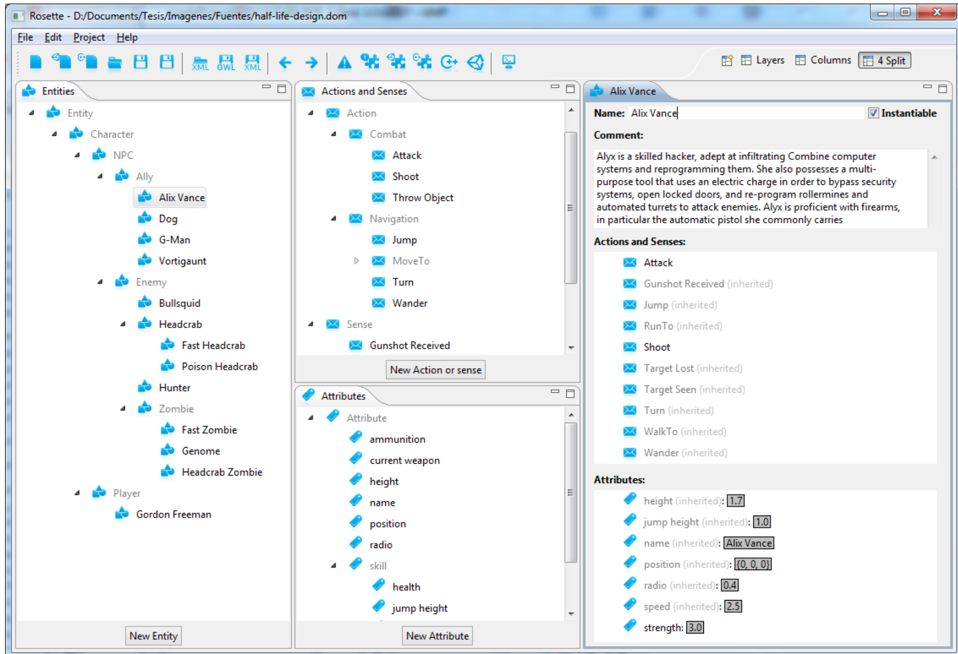

Figura 6.2: Vista de diseño de *Rosette*

a los cuales se les pueden asignar valores por defecto. Estos elementos que describen a la entidad aparecen en gris si son elementos que se han heredado del concepto superior en la jerarquía, mientras los elementos nuevos que especializan esta entidad aparecen en negro. Además se puede añadir una descripción textual que permita contextualizar o describir más específicamente algún elemento de la entidad y marcar si dicha entidad es o no es instanciable. Una acción o un sensor se visualiza de una manera similar, donde se puede definir si está parametrizado (tiene atributos) y se visualiza qué entidades son capaces de realizar la acción o capturar ese sensor. Los atributos permiten elegir el tipo, que puede ser básico (entero, real, cadena, vector, etc.) o de tipo entidad, donde se arrastra qué entidad define el tipo y luego se puede ver qué entidades están descritas por ese atributo. Las Figuras 6.4 y 6.5 que se explican más adelante presentan la visualización de los mensajes y atributos tal y como los ven los programadores. La vista de los diseñadores es una vista sesgada donde no se hace referencia a los componentes.

La vista del dominio que presenta *Rosette* a los programadores (Figura 6.3) es mas compleja y completa que la ofrecida a los diseñadores, aunque aun así guarda bastante parecido visual. Acercándose más a la arquitectura basada en componentes, además de las jerarquías de entidades y atributos que guardan la misma apariencia, la jerarquía de

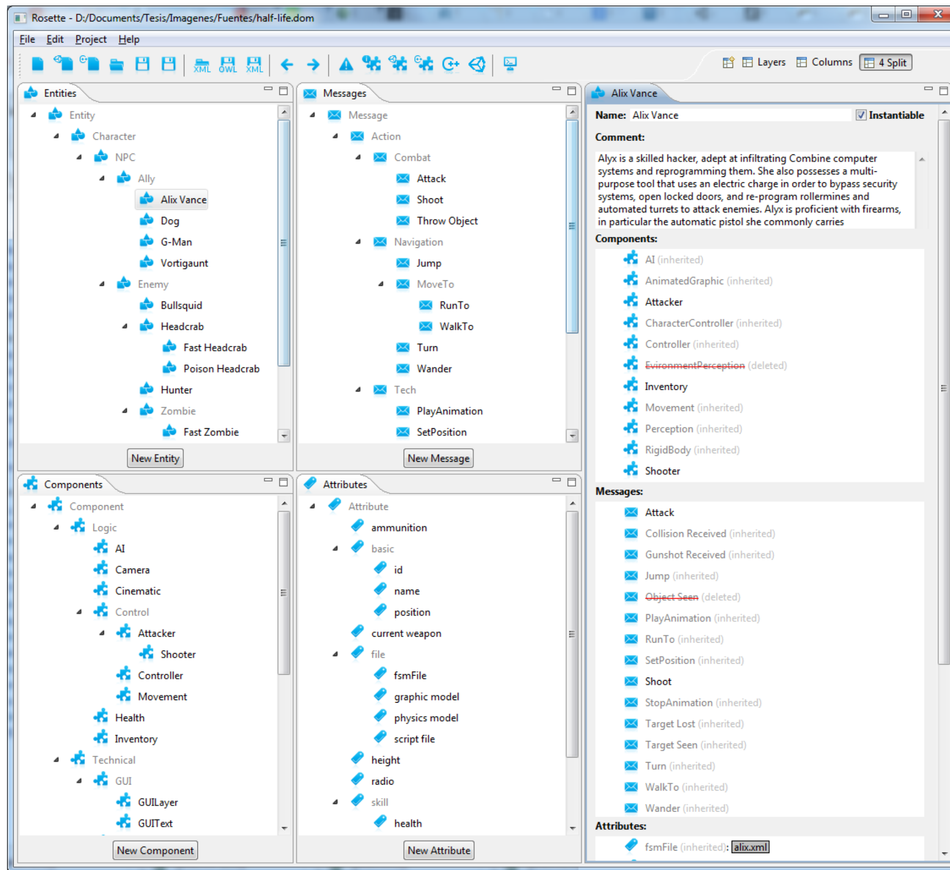

Figura 6.3: Vista de programación de *Rosette*

acciones y sensores se remplaza por la jerarquía de mensajes y hay una cuarta jerarquía, la de componentes. En las jerarquías de entidades y atributos se pueden añadir nuevos elementos, de carácter más técnico, y en la jerarquía de mensajes además se podrán añadir mensajes que no sean considerados ni acciones ni sensores. Todos estos elementos no serán visibles en la capa del diseñador ya que a ese nivel no tienen sentido. Los componentes son conceptos creados simplemente para la implementación del software y por ello su jerarquía sólo se muestra en la vista del programador. En la derecha del todo se mantiene la vista del elemento seleccionado, cuyas vistas contienen algo más de información.

La vista de una entidad desde el punto de vista del programador añade el concepto de componentes por los que está formada la entidad, además de los mensajes que es capaz de procesar y los atributos que la definen. Las entidades siguen ordenándose de manera jerárquica mediante la relación *es un* pero, como ya hemos explicado, esta relación

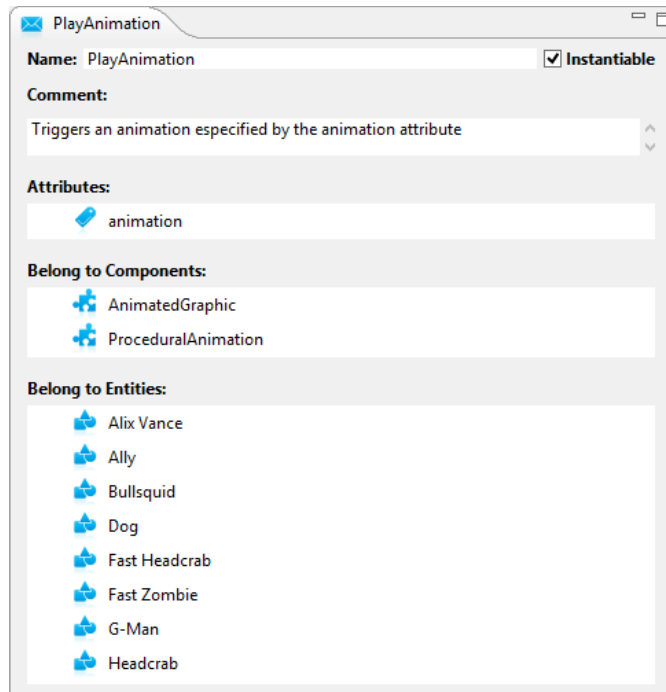

Figura 6.4: Vista de un mensaje para los programadores

es solo conceptual ya que la implementación real de las entidades será realizada mediante el uso de composición de componentes. Esto se ve por ejemplo en las entidades *Alix Vance* y *Ally* de la Figura 6.3. En ella se ve que *Alix Vance es un Ally* de manera conceptual, pero eso no quiere decir que *Alix Vance* deba obligatoriamente tener todos los componentes que tiene *Ally* ya que sino perderíamos la flexibilidad que nos da la arquitectura basada en componentes. Por ello, tal y como podemos ver en la descripción de componentes de *Alix Vance*, los componentes pueden ser heredados de la entidad padre (*Ally*) si están en gris, pueden ser componentes que especializan esta entidad respecto a su padre si aparecen en negro o bien pueden ser componentes que se eliminan si están tachados. Obviamente, eliminar componentes en clases hijas provoca que también se eliminen mensajes o atributos. Hemos querido hacer explícita estas eliminaciones ya que muestra más información al programador para distinguir unas entidades de otras, pero a su vez se puede mantener esa vista ontológica o jerarquía conceptual que ayuda a categorizar entidades.

La Figura 6.4 muestra como se ve la descripción de un mensaje en *Rosette*. Este mensaje concreto desencadena una animación que viene especificada en el atributo *animation*, presenta qué componentes son

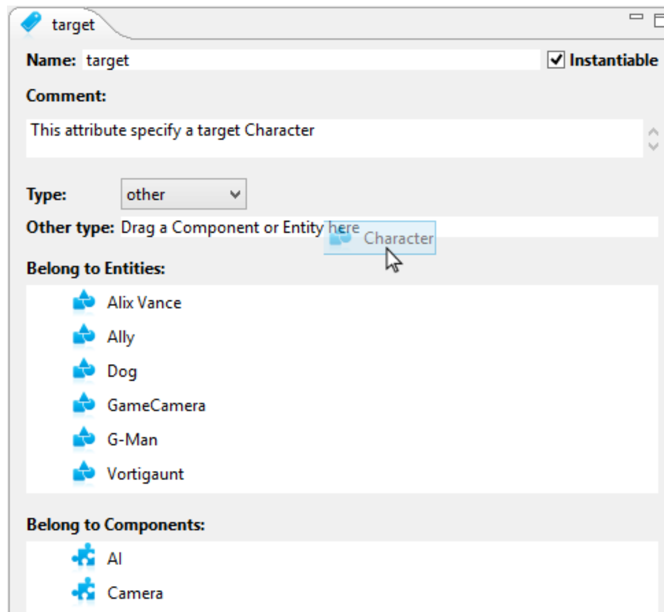

Figura 6.5: Vista de un atributo para los programadores

capaces de llevarlo a cabo e, infiriendo a partir de ellos, qué entidades. Tener toda esta información acerca de los mensajes, en lugar de tenerlos ocultos en el código, ayuda a su reutilización gracias al contexto que proporciona.

Un ejemplo de la vista que tienen los programadores de un atributo se muestra en la Figura 6.5 en la que en función del tipo elegido se permite añadir restricciones si es un tipo básico o arrastrar el tipo de entidad o componente que se quiere como tipo *other*. En la figura se muestra un atributo *target* que sirve en diferentes componentes y entidades para definir un *Character* que será el objetivo de la funcionalidad descrita en ellos.

Por otro lado, la Figura 6.6 muestra un componente que permite realizar ataques. En la lista de mensajes se puede ver como se muestran las dependencias existentes entre mensajes donde, para poder llevar a cabo la acción de ataque, otro componente debe interpretar el mensaje que lanza una animación. Se muestran también los atributos que se requiere que tenga la entidad a la que pertenezca (*strength*) y las entidades de las que este componente forma parte.

Todas estas representaciones sirven para generar y mostrar el dominio ontológico que se explicó en el Capítulo 4. En las siguientes secciones se explica que decisiones se han tomado para poder llevar a la práctica las técnicas expuestas en el Capítulo 5.

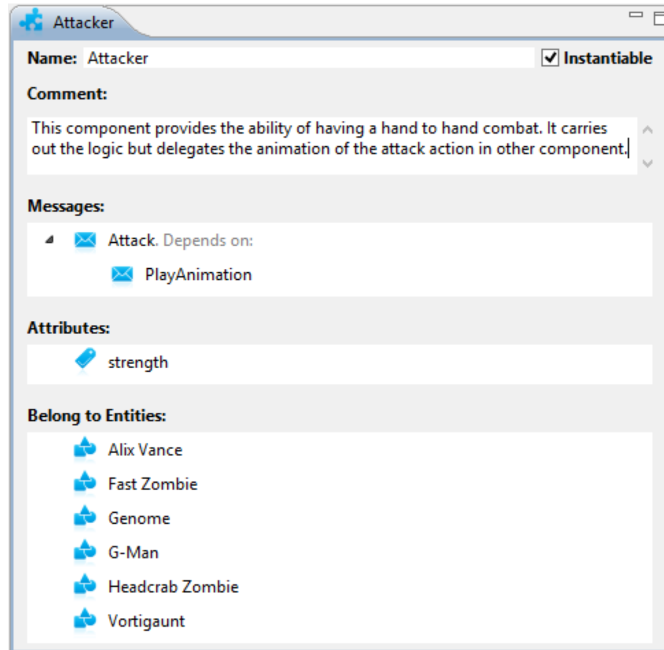

Figura 6.6: Vista de un componente para los programadores

6.2. Identificación automática de componentes software

A la hora de poner en práctica la técnica de identificación automática de componentes, e implementar un software que pudiésemos usar para medir su utilidad, tuvimos que:

1. Elegir un software con el que aplicar análisis formal de conceptos (FCA).

2. Ser capaces de almacenar dominios y los cambios que éstos han sufrido para poder realizar aplicaciones de FCA iterativas.

3. Presentar al usuario la parte del retículo no estática para que anote similitudes entre conceptos formales.

Para poder realizar la aplicación de FCA sobre el contexto formal se usa *Galicia project* (Valtchev et al., 2003), una plataforma de código abierto realizado para que investigadores en FCA puedan poner en práctica ideas teóricas. Habitualmente es usado por usuarios finales a través de su interfaz gráfica, sin embargo también permite ser usado por otras aplicaciones como *Rosette* a través de su API. Para ello es necesario generar un fichero con el contexto formal que se genera a partir

de la ontología, indicar que método se desea aplicar y se recibe, en otro fichero, el resultado de la aplicación de FCA en forma de retículo.

La idea de permitir un desarrollo iterativo implica que se deben almacenar todos los cambios que se realizan sobre el dominio de una iteración a la siguiente. En *Rosette*, para almacenar ese conjunto de cambios (y para proporcionar al usuario la posibilidad de deshacer y rehacer acciones), hemos utilizado el patrón de diseño *Command*, de manera que cualquier cambio que se quiera aplicar al dominio es un objeto que se almacena. Para poder deshacer y rehacer cambios se tienen dos pilas donde se van almacenando los comandos que se aplican sobre el dominio (haciendo o deshaciendo cambios) según decida el usuario pero, además, se tiene una lista especial donde se almacenan los cambios considerados de aplicación *necesaria* para mantener, de manera transparente al usuario, las modificaciones que él realizó sobre una distribución de FCA previa.

Gracias a esta posibilidad de deshacer los cambios realizados sobre el dominio, en *Rosette* no es necesario guardar el dominio completo de una iteración a otra, sino que cuando se va a realizar una nueva sugerencia de componentes lo que hace es:

1. Crear un nuevo dominio a partir del retículo generado por *Galicia*.

2. Identificar conceptos equivalentes entre el dominio recién generado en el paso 1 y el que se tenía previamente a esa generación.

3. Deshacer todos los cambios que se realizaron en el dominio desde la anterior aplicación de FCA.

4. Identificar conceptos equivalentes entre el dominio generado en el paso 1 y el dominio obtenido en el paso 3.

5. En caso que queden componentes que no han sido automáticamente anotados como equivalentes en el dominio antiguo se presentan al usuario para que manualmente realice las anotaciones que considere oportunas.

6. Se aplica toda la lista *especial* de modificadores sobre el nuevo dominio ahora que se han identificado las equivalencias entre componentes.

En los pasos 2 y 4 se suelen identificar todos o casi todos los conceptos que son equivalentes entre los diferentes dominios por lo que la sugerencia de nuevos componentes suele ser trasparente al usuario, de

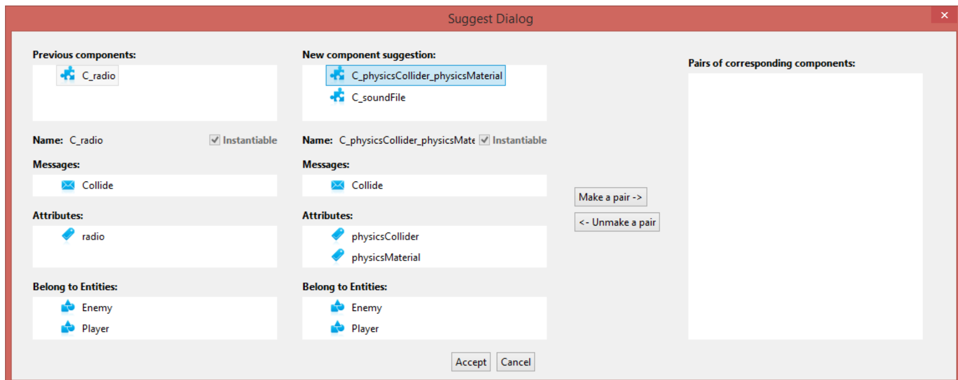

Figura 6.7: Pantalla donde los programadores anotan similitudes entre componentes

manera que al pulsar el botón pidiendo una nueva distribución automáticamente se modifican sus componentes previos y aparecen los nuevos componentes (con sus nombres autogenerados) en la jerarquía de componentes.

Sin embargo, en el caso de que se necesite realizar el paso 5, *Rosette* muestra un cuadro de diálogo que presenta al usuario sólo aquellos componentes a los que no se le han identificado componentes equivalentes en el otro dominio. La Figura 6.7 presenta ese cuadro de diálogo, el cual muestra en la parte superior los componentes sin equivalente en uno y otro dominio y, debajo de éstos, se presentan las características de los componentes seleccionados, en uno y otro dominio, para que el usuario pueda elegir y emparejar componentes. Según va seleccionando pares aparecen en la parte derecha del cuadro, que también se puede deshacer, y cuando decide terminar el emparejamiento solo hay que aceptar los emparejamientos.

6.3. Generación procedimental de contenido

Rosette permite crear modelos de juego que son independientes de la plataforma o tecnología con la que éstos se implementarán. Así *Rosette* sirve para crear un dominio semántico que no contiene detalles específicos de un lenguaje de programación concreto o del motor de juego que se tenga en mente, pudiendo incluso usar el mismo dominio para diferentes implementaciones. Sin embargo, como hemos contado anteriormente en la Sección 5.2, todo el conocimiento creado en *Rosette* y almacenado en una ontología puede ser usado para generar procedimentalmente contenido de juego.

Lo que queríamos proporcionar con *Rosette* es un método que permitiese a los desarrolladores crear contenido para el motor y lenguaje de programación que ellos deseen, sin tener que modificar la herramienta en si para ello. Con ese fin, el uso de plantillas parece que es una decisión acertada ya que nos permite la flexibilidad necesaria para generar contenido radicalmente distinto mediante el uso de etiquetas que modifican el comportamiento en función de los elementos concretos del dominio.

La generación de contenido puede variar bastante en función de la plataforma y motor seleccionados, no solo debido al lenguaje usado en el código fuente y a la naturaleza de la descripción de los elementos de juego para los diferentes editores, sino también debido a cuáles son los elementos se deben generar y en qué diferentes ficheros y directorios se deben almacenar. Por tanto, antes de plantearnos qué tipo de plantillas usaremos, debemos crear en *Rosette* una infraestructura que nos ofrezca la flexibilidad necesaria para que posteriormente los desarrolladores que usen *Rosette* puedan añadir las plantillas que ellos necesiten.

En función del motor o plataforma elegidos querremos que el contenido generado se almacene en unos u otros directorios por lo que el primer paso es permitir elegir las rutas relativas del proyecto en las que se generará el contenido. Pero además, en función del lenguaje de programación se pueden necesitar uno o más ficheros para crear el contenido de un elemento del dominio (por ejemplo C++ necesita dos ficheros por clase mientras que en Java o C# es suficiente con uno). Por tanto se deberá permitir asociar más de una plantilla a la generación de un mismo elemento. *Rosette* permite ser configurado para poder modificar dichas características y, por ello, cada conjunto de plantillas debe ir acompañado de un fichero de configuración donde se seleccionará qué uso quiere hacerse de dichas plantillas y dónde se generarán los ficheros resultantes.

Durante el desarrollo de *Rosette* hemos probado con diferentes tecnologías para el desarrollo de plantillas. En primer lugar usamos *XSL Transformations* (XSLT)[5], un lenguaje XML que en principio está pensado para generar otros ficheros XML pero que en nuestro caso lo usábamos con otros fines. Usamos también un sistema de plantillas propio que permite sustituir etiquetas de un conjunto de plantillas por texto generado a través de un API que genera el programador. Sin embargo, finalmente hemos optado por el uso de *Apache Velocity*[6], un motor de plantillas basado en Java que permite hacer referencia a métodos y

[5]urlhttp://www.w3.org/TR/xslt

[6]`http://velocity.apache.org/`

elementos definidos dentro del código Java. Este mecanismo nos permite mayor flexibilidad respecto a los métodos anteriores y evita que los desarrolladores tengan que extender *Rosette* por medio de APIs. Además, como *Rosette* ha sido íntegramente desarrollado en Java, desde las diferentes plantillas se puede acceder a los elementos del dominio que representan los conceptos de la ontología OWL.

Para comprobar el potencial de la técnica, y probar la flexibilidad de adaptación a diferentes motores, hemos generado plantillas (y sus correspondientes ficheros de configuración) que permiten la generación de contenido para dos motores sustancialmente distintos. El primero es *Unity*[7], un motor comercial donde los scripts generados para el código se encuentran en C# y los recursos para la edición de nivel se generan para el editor del motor. El segundo se trata de un pequeño motor desarrollado para el máster de videojuegos de la Universidad Complutense de Madrid[8], donde el código generado es C++ y donde también se generan recursos para los niveles.

Apache Velocity usa referencias para incluir contenido dinámico dentro de una plantilla, donde una variable es un tipo de referencia que puede referirse a un elemento definido dentro del código Java o a un elemento definido en la propia plantilla. En *Apache Velocity* las referencias comienzan con $ y se utilizan para obtener un valor mientras que las directivas comienzan con # y se utilizan para definir el flujo que se sigue para rellenar la plantilla. La Figura 6.8 muestra un ejemplo parcial de plantilla para la creación de un componente C# en *Unity*. Por cada componente declarado en la ontología se generará, a partir de esta plantilla, un fichero nuevo listo para ser usado en una ruta especificada en el fichero de configuración. Desde el punto de vista del código no hace falta ninguna plantilla más ya que *Unity* soluciona todo lo relacionado con las entidades de juego y la gestión y envío de los mensajes. Sin embargo sí son necesarias otras plantillas que generan contenido de juego, que en este caso son ficheros que modifican el editor de *Unity* para ofrecer nuevas funciones y añadir *prefabs* (entidades prototípicas) al editor.

La plantilla usada para la creación de componentes en *Unity* no es demasiado extensa ni compleja ya que tanto el motor como el lenguaje de programación simplifican y reducen el código necesario por componente. Para generar el mismo componente dentro de nuestro motor en C++ se necesita mayor cantidad y complejidad de código ya que el lenguaje es de más bajo nivel y el motor no gestiona por si mismo cosas como el paso y procesado de mensajes. Además, las diferentes clases

[7] `http://unity3d.com/unity/`
[8] `http://www.videojuegos-ucm.es/`

```
using UnityEngine;

// Region for the user "using"s.
${esc.hash}region UserDecInclusions
$!AditionalDecInclusions
${esc.hash}endregion //UserDecInclusions

/// <summary>
/// @file ${component.Name}.cs
/// ${component.Description}
/// @author Rosette
/// </summary>
[AddComponentMenu("Rosette/$component.Name")]
public class ${component.Name} : ${component.Parent.Name}
{
${esc.hash}region AutogeneratedAttributes
\#foreach( $attribute in $component.Attributes )
\#CSType($attribute.type) $attribute.name;

\#end
${esc.hash}endregion //AutogeneratedAttributes
...
} // class ${component.Name}

\#macro( CSType $attribute )
\#if ($attribute.type == "tint")
int
\#elseif ($attribute.type == "tfloat")
float
\#elseif ($attribute.type == "tstring")
string
\#elseif ($attribute.type == "tunsignedint")
uint
\#elseif ($attribute.type == "tbool")
bool
\#elseif ($attribute.type == "tvector3")
Vector3
\#else \#\#($attribute.type == "other")
$attribute.other.toString()
\#end
\#end
```

Figura 6.8: Una plantilla para la cabecera de los componentes en C# de *Unity*

```
${esc.hash}ifndef __Logic_${component.Name}_H
${esc.hash}define __Logic_${component.Name}_H

${esc.hash}include "Logic/Entity/${component.Parent.Name}.h"

// Region for the user "include"s.
${esc.hash}pragma region UserDecInclusions
$!AditionalDecInclusions
${esc.hash}pragma endregion UserDecInclusions

namespace Logic {
/**
  @file ${component.Name}.h
  ${component.Description}
  @author Rosette
*/
  class ${component.Name} : public $component.Parent.Name {
    DEC_FACTORY($component.Name);
    ${esc.hash}pragma region AutogeneratedAttributes
    \#foreach( $attribute in $component.Attributes )
      \#CPPType($attribute.type) $attribute.name;
    \#end
    ${esc.hash}pragma endregion AutogeneratedAttributes
  }; // class $component.Name
  REG_FACTORY($component.Name);
} // namespace Logic
${esc.hash}endif // __Logic_${component.Name}_H

\#macro( CPPType $attribute )
\#if ($attribute.type == "tint")
int
\#elseif ($attribute.type == "tfloat")
float
\#elseif ($attribute.type == "tstring")
std::string
\#elseif ($attribute.type == "tunsignedint")
unsigned int
\#elseif ($attribute.type == "tbool")
bool
\#elseif ($attribute.type == "tvector3")
Vector3
\#else \#\#($attribute.type == "other")
$attribute.other.toString()*
\#end
\#end
```

Figura 6.9: Una plantilla para la cabecera de los componentes en C++

```
/**
@file ${component.Name}.cpp

@author Rosette
*/

${esc.hash}include "${component.Name}.h"

${esc.hash}include "Logic/Entity/Entity.h"
${esc.hash}include "Logic/Maps/Map.h"
${esc.hash}include "Map/MapEntity.h"

// Region for the user "include"s.
${esc.hash}pragma region UserImpInclusions
$!AditionalImpInclusions
${esc.hash}pragma endregion UserImpInclusions

namespace Logic
{
  IMP_FACTORY(${component.Name});

  bool ${component.Name}::spawn(CEntity *entity, CMap *map,
                                const Map::CEntity *entityInfo) {
    if(!${component.Parent.Name}::spawn(entity,map,entityInfo))
      return false;
\#foreach( $attribute in $component.Attributes )
    if(entityInfo->hasAttribute("$attribute.Name"))
      \_$attribute.Name = entityInfo->
                                getAttributeValue("$attribute.Name");
\#end
    return onSpawn();
  } // spawn

\#foreach( $message in $component.Messages )
${esc.hash}pragma region $message.Id
  private void $message() {
    $!body.getCode($message.Id)
  }
${esc.hash}pragma endregion $message.Id
\#end

} // namespace Logic
```

Figura 6.10: Una plantilla para la implementación de los componentes en C++

que representan conceptos como los componentes no pueden estar en un único fichero sino que en este caso por cada clase se generarán dos, el que contiene la declaración de la clase (Figura 6.9) y el que contiene la implementación de la misma (Figura 6.10). Además de los componentes son necesarias plantillas para la declaración e implementación de los mensajes, entidad, etc. y, aparte del código en si mismo, se deben generar también los ficheros de proyecto de *Visual Studio* mediante otras plantillas. El contenido de juego generado para la edición de niveles en este caso es más austero; al no disponer de un editor de juego tan completo como *Unity* lo único que se genera son las descripciones de las entidades de manera que puedan ser entendidas por el motor de juego y por el editor de mapas. Esto se reduce a los ficheros *blueprints* y *archetypes*.

Para poder mantener una generación iterativa en el código se realizan meta-anotaciones, como las que se pueden ver especificadas en las Figuras 6.8, 6.9 y 6.10, que delimitan regiones en las cuales el programador debe escribir el código que corresponda. Gracias a esas meta-anotaciones ese código queda asociado a los elementos del dominio formal de manera que podrá identificarse. Cuando se trabaja de manera iterativa con *Rosette*, ante una nueva generación de código, *Rosette* identifica las porciones de código que han sido creadas por los programadores y los pasa como un elemento Java a las plantillas en futuras generaciones de código. En la Figura 6.10 se ve un ejemplo de uso en la línea en la que se invoca a `$!body.getCode($message.Id)`, donde ese método devuelve el cuerpo asociado a un mensaje concreto en un componente. En función de los cambios y refactorizaciones que haya sufrido el dominio, el código de una nueva generación de contenido se mantendrá en el componente original o cambiará de componente. Las refactorizaciones y modificaciones del dominio quedan todas registradas en *Rosette* de una iteración a otra como se explica con detalle en la Sección 5.2.3 y por tanto eso nos permite mantener y recolocar las piezas de código en función de los cambios que se realizaron en el diseño, siendo todo esto siempre transparente para el usuario final. Gracias al uso de plantillas y las meta-anotaciones de éstas, todo este proceso es transparente y se realiza con independencia del motor y lenguaje que se esté usando sin necesidad de modificar *Rosette*.

Cabe decir que una de las grandes ventajas del uso de *Apache Velocity* es que, una vez que se conoce su sintaxis, se realizan las plantillas necesarias para un nuevo motor y lenguaje de una manera rápida y sencilla. Debido a ésto, para obtener todos los beneficios que proporciona la generación procedimental de contenido, el precio a pagar por los pro-

gramadores es muy pequeño los beneficios que se obtiene con ello son muy grandes.

6.4. Detección de inconsistencias

Como se ha comentado previamente, el lenguaje que hemos decidido usar para la descripción de nuestra ontología es OWL. Un error común acerca de OWL es pensar que puede ser fácilmente utilizado como un lenguaje de esquema expresivo, los cuales se suelen utilizar para validar los datos en bases de datos relacionales o datos XML. Debido a la *asunción de mundo abierto*, y la ausencia de *asunción de nombre único*, adoptada por OWL los axiomas en una ontología OWL pretenden inferir nuevos conocimientos en lugar de lanzar inconsistencias.

Nuestra intención sin embargo es usar OWL tanto para inferir nuevos conocimientos a través de la inferencia como para validar los conceptos que se definen en la ontología OWL. *Pellet Integrity Constraint Validator* (*Pellet ICV*)[9] nos permite hacer ambas cosas ya que ofrece una semántica alternativa para que los axiomas OWL sean interpretados con la *asunción de mundo cerrado* y una forma débil de *asunción de nombre único*. La interpretación de la *asunción de mundo cerrado* significa que suponemos que una afirmación es falsa si no sabemos explícitamente si es verdadera o falsa mientras que la *asunción de nombre único* débil significa que si dos conceptos no se infiere que sean el mismo, entonces se supone que son distintos.

En *Rosette* por tanto hacemos uso de *Pellet ICV*, el cual puede interpretar el modelo semántico que hemos creado en OWL y proporcionar *feedback* al usuario final acerca de que inconsistencias se han detectado. *Pellet ICV* analiza cualquier tipo de ontología OWL y proporciona una respuesta textual que indica si el dominio es consistente o si, por el contrario, se ha violado alguna restricción. A día de hoy su uso se realiza lanzando *Pellet ICV* en segundo plano e interpretando la respuesta textual que se recibe de éste. Si el mensaje recibido no contiene error se comunica al usuario que el dominio es consistente, en caso contrario se traslada directamente el error recibido. Una mejora a este proceso sería analizar la respuesta obtenida y, en función de ello, señalar cuales son los elementos inconsistentes y ofrecer al usuario alternativas o soluciones.

Con la intención de probar como funciona el sistema vamos a presentar ciertas inconsistencias. Para este propósito hemos modelado un

[9] `http://clarkparsia.com/pellet/icv/`

pequeño ejemplo en el que definimos un par de atributos reales:

```
DataProperty: strength
  SubPropertyOf: BasicAttribute
  Range: float

DataProperty: success
  SubPropertyOf: BasicAttribute
  Range: float
```

A su vez tenemos un par de mensajes para realizar ataques y reproducir animaciones:

```
Class: Attack
  SubclassOf: Message
    and (isInstantiable)

Class: PlayAnimation
  SubclassOf: Message
    and (animation some string)
    and (isInstantiable)
```

Un componente que permite realizar ataques, siempre y cuando exista otro componente en la misma entidad capaz de realizar animaciones:

```
Class: Attacker
  SubclassOf: Component
    and (interpretMessage some Attack)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some PlayAnimation)))
    and (strength some float)
    and (success some float)
    and (isInstantiable)
```

Y por último una entidad *torreta*, que tendrá sólo el componente descrito y por tanto podrá llevar a cabo mensajes de tipo *Attack* y tendrá los atributos *strength* y *success*:

```
Class: Turret
  SubclassOf: Entity
    and isA EnemyBuilding
    and (hasComponent some Attacker)
    and (interpretMessage some Attack)
    and (strength some float)
    and (success some float)
    and (isInstantiable)

Individual: iTurret
  Types: Turret
  Facts: strength "strong",
```

Con la definición de este ejemplo tan sencillo tratemos ahora de validarlo con *Rosette*, que lanza *Pellet ICV* con el fichero OWL en el que se almacena el dominio formal del juego. Este pequeño modelo tiene 3 inconsistencias, donde la más obvia es que el atributo *strength* ha sido definido como real pero se le ha impuesto un valor de cadena ("strong"). La forma en la que se presenta ese error *Pelet ICV*, y por tanto también *Rosette*, es la siguiente:

```
Ontology is inconsistent!
Literal value "strong"^^string does not belongs to
datatype float
```

Cuando se soluciona esta inconsistencia surge una nueva que se debe a la ausencia de asignación de valor en el individuo *iTurret* para el atributo *success*. Como en la anterior inconsistencia *Rosette* proporciona *feedback* al usuario retornando la salida de *Pellet ICV*:

```
VIOLATION: iAttacker violates AI subClassOf (Component
and (isComponentOf only (strength some float))
and (isComponentOf only (success some float)))
  INFERRED: iAttacker type Attacker
    ASSERTED: iAttacker type Attacker
  NOT INFERRED: iAttacker type (Component and
  (isComponentOf only (strength some float))
(isComponentOf only (sucess some float)))
    NOT INFERRED: iAttacker type (isComponentOf only
    (sucess some float))
      INFERRED: iAttacker isComponentOf
      iTurret
        ASSERTED: iTurret hasComponent
```

```
        iTurret
        ASSERTED: hasComponent inverseProperty
        isComponentOf
      NOT INFERRED: iTurret type
      (sucess some float)
        NOT ASSERTED: iTurret sucess _
```

Donde, si se analiza detalladamente, nos damos cuenta que *Pellet ICV* informa que el atributo *success* debe ser definido en el individuo *iTurret*. Si se solventa también esta inconsistencia aparece la tercera que es algo más difícil de ver. La entidad *Turret* tiene el componente *Attacker* que puede ejecutar mensajes de tipo *Attack* pero, como se dijo anteriormente, este componente delega la animación que se debe producir enviando un mensajes de tipo *PlayAnimation* a la entidad a la que pertenece. Por tanto, la entidad *Turret* debería tener otro componente que interprete mensajes de tipo *PlayAnimation* pero sin embargo no tiene un componente de ese estilo. En este caso, *Pellet ICV* indica que no se pueden interpretar mensajes de tipo *PlayAnimation*:

```
Constraint violated  : Yes
Violating individuals (1): iTurret,
Explanation:
VIOLATION: iTurret violates Attack
subClassOf (Message and not PlayAnimation and
(isMessageInterpretedBy only (Component and
(isComponentOf only (Entity and (hasComponent some
(Component and (interpretMessage some
PlayAnimation)))))))
  ASSERTED: iWanderingCharacter hasComponent
  iAttacker
  NOT INFERRED: iAttacker type (Component and
  (interpretMessage some PlayAnimation))
    NOT INFERRED: iAttacker type (interpretMessage
    some PlayAnimation)
      ASSERTED: iAttacker interpretMessage iAttack
      NOT INFERRED: iAttack type PlayAnimation
        NOT ASSERTED: iAttack type PlayAnimation
```

Pellet ICV nos ayuda enormemente a identificar inconsistencias en la definición de nuestro dominio, que es una de las técnicas que se proponen y que ayudan a que, tanto diseñadores como programadores, tengan confianza en el trabajo que han realizado y se acelere la productividad

(al detectar errores de manera temprana). Sin embargo, a día de hoy el *feedback* proporcionado por *Rosette* son las descripciones proporcionadas por *Pellet ICV* por lo que su interpretación puede ser complicada sobretodo por los diseñadores.

Capítulo 7

Experimentos

Con la finalidad de comprobar si la metodología propuesta en esta tesis y las técnicas que soportan dicha metodología funcionan hemos llevado a cabo dos experimentos que ponen a prueba el trabajo realizado. En el primer experimento se usó la metodología con fines pedagógicos, para enseñar a alumnos del máster de videojuegos de la Universidad Complutense de Madrid qué es la arquitectura basada en componentes y cómo se debe distribuir la funcionalidad. En el segundo se realizó una evaluación empírica de la técnica de identificación automática de componentes software. Además, en los dos experimentos se usó *Rosette*, con lo que se pudo evaluar y mejorar la herramienta y poner a prueba el resto de elementos de la metodología, especialmente la generación procedimental de contenido.

7.1. Metodología para enseñanza de la arquitectura basada en componentes

7.1.1. Introducción

El máster en desarrollo de videojuegos de la Universidad Complutense de Madrid[1] ha estado especializando estudiantes licenciados en informática desde 2004 y convirtiéndoles en profesionales en programación de videojuegos. Durante los últimos 5 años hemos estado enseñando a los alumnos la arquitectura basada en componentes para la gestión de las entidades de juego ya que hoy en día es la técnica que se usa en la mayoría de desarrollos profesionales. Además, el uso de esa arquitectura ha sido un requerimiento para los proyectos fin de máster que realizan los alumnos. Los programadores en general suelen estar muy

[1] `http://www.videojuegos-ucm.es/`

acostumbrados al sistema de tradicional de jerarquías de objetos y no suelen estar tan acostumbrados a la idea de composición de objetos y agregación.

El aprendizaje de la arquitectura basada en componentes suele parecer inútil e innecesariamente complicada para los programadores, los cuales suelen ser reacios a su imposición (West, 2006). En nuestra experiencia, los estudiantes del máster de videojuegos tienen un duro tránsito hasta sentirse cómodos y ser competentes en la aplicación de esta arquitectura para gestionar las entidades dentro del videojuego. Las ideas principales de la arquitectura basada en componentes son generalmente entendidas con facilidad por un licenciado en informática pero, aunque la teoría es asimilada con facilidad, lleva tiempo y un buen número de malas distribuciones desarrollar las destrezas necesarias para que un ingeniero informático termine con un buen diseño basado en componentes en términos de reutilización, extensibilidad y ausencia de duplicación de funcionalidad.

Por esa razón, hemos diseñado una nueva metodología de enseñanza que permite a los estudiantes iterar muy rápidamente por versiones de un juego simple que incrementa progresivamente la complejidad. Dicha metodología les permite probar y evaluar un gran número de distribuciones de componentes, lo cual acelera el proceso de aprendizaje. Esta aproximación es posible gracias a el uso de *Rosette* (Capítulo 6) y de cómo usa el dominio ontológico y lo relaciona directamente con los diferentes trozos de código.

Hemos desarrollado por tanto un método iterativo para evaluar el uso de *Rosette* y cómo afecta a la comprensión del diseño de una arquitectura basada en componentes. El método conlleva el desarrollo de varias iteraciones de dos juegos sencillos, el primero de ellos desarrollado en *Unity* [2] y el segundo programado en C++.

7.1.2. Metodología

Como profesores en el máster de videojuegos de la Universidad Complutense de Madrid, durante los últimos años nos hemos enfrentado a ciertas dificultades mientras enseñamos la arquitectura basada en componentes usada en videojuegos. Para aliviar esas dificultades, desde el curso 2012-13, hemos adoptado un nuevo método el que se introduce *Rosette* y nuestra metodología de desarrollo de videojuegos pero en este caso para usarla como una herramienta pedagógica que permita a los estudiantes asimilar las técnicas que propician un buen diseño de una

[2] http://unity3d.com/

arquitectura basada en componentes.

En ediciones previas del máster los estudiantes implementaban un juego 3D usando una arquitectura basada en componentes, pero lo hacían de una manera guiada. Tras unas explicaciones básicas acerca de los aspectos teóricos de la arquitectura, se les exponía una serie de componentes que debían implementar para desarrollar un juego (que usan a lo largo del curso como banco de pruebas) y por qué debían implementar dichos componentes. Lo que finalmente recibían eran unos componentes de ejemplo con el código implementado pero con varios huecos donde tenían que escribir código específico que implementaba funcionalidad del juego.

Aunque los estudiantes parecían asimilar el funcionamiento interno de la arquitectura, cuando se enfrentaban a su proyecto de máster, donde deben implementar su propio videojuego prácticamente desde cero, sufrían muchas dificultades intentando distribuir la funcionalidad de las entidades en diferentes componentes software. Los primeros prototipos que creaban solían tener componentes muy grandes con muy baja cohesión o componentes muy específicos y nada configurables. Ninguno de estos componentes eran reutilizables por lo que cada nueva característica, nueva entidad o cambio en el diseño del juego suponía un nuevo componente (o más de uno), generalmente muy similar a uno ya existente (o aparte de uno ya existente), causando duplicación de código y errores (*bugs*).

Nuestra conclusión fue que, aunque los fundamentos de las arquitectura basada en componentes es fácil de entender, llegar a ser competente identificando y diseñando una buena distribución de componentes no es tan sencillo. Después de todo, la arquitectura basada en componentes rompe con el paradigma de la orientación a objetos y su jerarquía de clases, por lo que los estudiantes deben cambiar su manera de pensar cuando diseñan componentes.

Con la finalidad de paliar las dificultades que sufren los alumnos, hemos cambiado nuestra metodología de enseñanza. Ahora enfrentamos a los alumnos al desarrollo de dos pequeños juegos en los que no deben solo rellenar unos huecos, sino diseñar por completo la capa lógica de las entidades de juego, identificar qué entidades tendrá el juego y distribuir su funcionalidad en componentes.

Para emular un desarrollo completo de un videojuego, donde el diseño del juego suele variar durante el desarrollo, proporcionamos a los estudiantes definiciones parciales de cada juego que posteriormente se va enriqueciendo progresivamente tras varias iteraciones. Nuestra intención es forzar a los alumnos a que revalúen sus distribuciones previas,

tratando de reutilizar tantos componentes como sea posible o decidiendo cuándo deben refactorizar la distribución actual. De esta manera, imitamos los retos a los que se deberán enfrentar los estudiantes durante sus proyectos, pero en un entorno más controlado.

En lugar de dejarles implementar el juego por su cuenta usando directamente un lenguaje de programación, usan *Rosette* como ayuda. Cada iteración del desarrollo consta por tanto de las siguientes fases:

- Se les distribuye a los estudiantes un ejecutable de ejemplo de lo que deben implementar en la iteración actual (a modo de "demo"). Podríamos haber usado un documento de diseño prototípico explicando cual es la funcionalidad requerida. Sin embargo, estos documentos requieren un tiempo para ser analizados y son bastante propensos a malas interpretaciones. Un ejecutable del juego acelera el proceso de comprensión y evita problemas. Además del ejecutable los estudiantes reciben todos los recursos gráficos necesarios para implementar su propio *clon* del juego.

- Una vez que los requerimientos han sido asimilados mediante la prueba del ejecutable, los estudiantes deben concentrarse en definir las entidades de juego y distribuir la funcionalidad de éstas en componentes. Para esta tarea usan *Rosette*, el cual les ayuda a crear una colección de componentes en la primera iteración o actualizar dicha colección en las siguientes iteraciones. En esta fase los estudiantes mantienen sus mentes limpias de detalles de implementación, solo piensan en la distribución a alto nivel, lo que les permite crear distribuciones que son mejores desde el punto de vista semántico.

- Finalmente los alumnos usan *Rosette* para (re)crear el código y contenido del juego, implementan las funcionalidades que han realizado en la fase anterior y distribuyen las entidades por el mapa del nivel (para que se vea como en el ejecutable entregado). Como *Rosette* es una herramienta que soporta el desarrollo iterativo, puede generar fragmentos de código en cualquier momento, sin que eso suponga la pérdida o modificación del código fuente implementado a mano por los estudiantes en las iteraciones pasadas.

Hemos definido el proceso como tres etapas pero, durante la misma iteración, los estudiantes pueden cambiar de la fase de diseño en *Rosette* a la fase de implementación tantas veces como quieran en el caso de que no hiciesen una correcta especificación del dominio y una buena descomposición de componentes en el primer intento.

Como profesores, nuestro objetivo en estas sesiones prácticas es enseñar a los alumnos cómo distribuir funcionalidad en componentes, pero la motivación de los estudiantes es clave e introduciendo la implementación de sus dos primeros videojuegos conseguimos potenciarla. En este sentido *Rosette* incrementa en gran medida el desarrollo durante las iteraciones y simplifica la refactorización de la arquitectura basada en componentes. Esto significa que los estudiantes pueden probar de una manera rápida diferentes distribuciones de componentes sin tener que preocuparse por los cambios que eso supone en el código fuente. El tedioso trabajo de la refactorización de código es automáticamente llevado a cabo por *Rosette* lo que permite que, cuando los estudiantes modifican las distribuciones, sólo tengan que hacerlo a alto nivel (salvo que introduzcan nueva funcionalidad, que tendrán que implementarla en el código). En las siguientes secciones demostraremos que la posibilidad de trabajar a un nivel semántico ha sido un gran mecanismo para el aprendizaje (y enseñanza) de la arquitectura basada en componentes.

7.1.3. Metodología en práctica

Hemos puesto en práctica las ideas antes presentadas y hemos realizado dos experiencias separadas con el mismo grupo de personas. Todos ellos eran estudiantes de nuestro máster en desarrollo de videojuegos de la Universidad Complutense de Madrid y la mayoría de ellos eran ingenieros informáticos. De los 19 estudiantes que participaron sólo alguno de ellos tenía experiencia previa en el desarrollo de videojuegos y ninguno de ellos había creado nunca una distribución de componentes.

Antes de llevar a cabo el experimento, los estudiantes fueron enfrentados a la creación de pequeños juegos usando componentes. Para ser más específicos, en las primeras semanas del máster, el proceso de desarrollo de videojuegos se introduce con la ayuda de Unity, una plataforma de desarrollo de juegos basada en componentes. Los profesores guiaron a los estudiantes para construir, paso a paso, algunos pequeños prototipos. Después de eso, recibieron unas clases teóricas acerca de la arquitectura de juegos en general, la manera tradicional de gestionar las entidades usando implementaciones basadas en herencia y la arquitectura basada en componentes para el mismo propósito.

El experimento tuvo lugar en este contexto y fue dividido en dos partes diferentes. La primera consistía en usar *Unity* para crear un pequeño juego. *Unity* tiene una biblioteca integrada que tiene componentes comunes, por lo que los estudiantes partían con una colección de componentes para empezar y debían extenderla. *Rosette*, que trae integra-

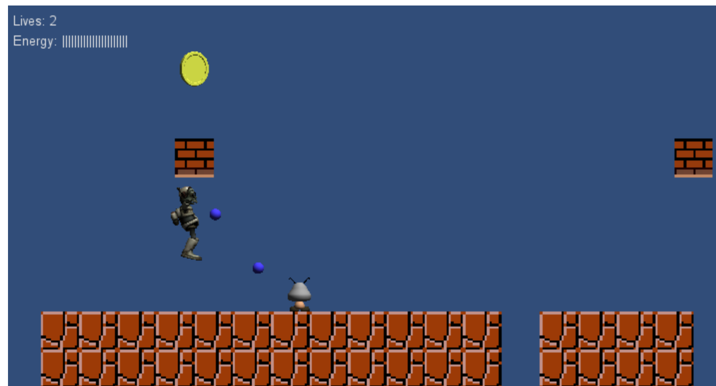

Figura 7.1: Primer juego, creado usando Unity

da una representación semántica de todos esos componentes, es capaz de generar código C# para los scripts de *Unity* y de generar también los *prefabs* o entidades prototípicas usadas para la generación de los niveles de juego. Por tanto, los estudiantes usaban *Rosette* como una herramienta externa donde modelar la ontología de juego y los componentes antes de implementar los detalles en *Unity*, donde gracias a la generación de contenido, sólo era necesario implementar unos pequeños huecos.

Tras esto pasamos a una segunda experiencia que incrementaba la dificultad, en donde la capa lógica del juego se desarrollaba desde cero, sin partir de ninguna colección de componentes y usando C++. En esta segunda experiencia también contaba con el apoyo de *Rosette* para la etapa de diseño de las iteraciones (aunque partía con un dominio vacío esta vez) y *Rosette* también soportaba la creación de contenido usada en la etapa de implementación. En esta etapa se generaba la gran mayoría de la capa lógica del juego, los componentes con los huecos donde el estudiante luego implementaba la funcionalidad y los ficheros *blueprint* y *archetype*, que definen las entidades prototípicas para poder editar los mapas.

7.1.3.1. Primer juego: usando Unity

El objetivo del estudio es enseñar a los estudiantes cómo crear un videojuego desde cero usando una arquitectura basada en componentes. Por tanto nuestra pregunta de investigación es: ¿Los estudiantes aprenden de manera efectiva a usar arquitecturas basadas en componentes cuando desarrollan juegos desde cero si introducimos nuestra metodología y *Rosette* en el proceso?

Al final de curso los estudiantes tienen un buen entendimiento de los diferentes módulos software que componen un videojuego tradicional. Sin embargo, al empezar el curso obviamente están muy lejos de tener todo ese conocimiento así que, para aliviar el proceso de aprendizaje, el primer juego que les proponemos desarrollar a los estudiantes se realiza usando *Unity*. *Unity* es un motor de juego multiplataforma e interfaz de desarrollo que facilita el desarrollo de videojuegos proporcionando un editor de juego visual, subsistemas de gestión, lenguajes de script para crear el contenido del juego y una gran cantidad de otras características. Tiene una gestión de entidades bien diseñada y basada en componentes y trae integrada la implementación de una biblioteca de componentes que resuelven los aspectos técnicos del desarrollo de videojuegos como físicas realistas, colisiones, renderizado, sonido y un largo etcétera.

Todas estas características hacen de *Unity* la herramienta ideal para nuestro primer experimento. Los estudiantes se enfrentarán a la creación de un videojuego y deberán preocuparse sólo acerca del diseño de los componentes que implementan funcionalidad específica de ese juego, ya que los componentes técnicos están implementados, tienen un editor profesional de niveles y un lenguaje de alto nivel disponible como es C#, mucho más sencillo que C++.

El juego que los estudiantes tenían que implementar era un juego de plataformas 2D de desplazamiento lateral, donde la funcionalidad se añadió progresivamente en las siete iteraciones que se detallan a continuación:

1. La primera iteración del juego no tenía más que alguna plataforma estática, una plataforma que se mueve entre dos puntos, un jugador que puede moverse a izquierda y derecha (usando el teclado) y que cae si no hay una plataforma bajo sus pies y la lógica de la cámara que se desplaza horizontalmente para mostrar siempre al jugador en pantalla (excepto cuando cae). Se trata de una funcionalidad muy básica y simple para un juego, pero suficiente para permitir a los alumnos pensar en una primera distribución de funcionalidad en componentes. La idea es que, si crean una buena distribución de componentes de partida, los siguientes pasos serán mucho más sencillos.

2. En la segunda iteración del desarrollo del juego los estudiantes debían continuar añadiendo nuevas características al juego. El jugador debía ser capaz de volar durante algunos segundos, usando sus baterías recargables, y debe morir cuando cae al vacío. Cuando el jugador muere debe perder una vida y empezar de nuevo. Si

pierde todas las vidas el juego debe de terminar. La segunda iteración es similar a la primera ya que los estudiantes solo deben distribuir la nueva funcionalidad en los componentes ya existentes o creando nuevos componentes. Sin embargo, en la siguientes iteraciones una buena distribución ayuda mucho.

3. En la tercera iteración se añade el primer enemigo del juego, el cual se mueve entre dos puntos y, si el jugador colisiona con él, el jugador muere. Si los estudiantes han hecho una buena distribución esta iteración es trivial porque el enemigo se mueve de la misma manera que la plataforma móvil y mata al jugador del mismo modo que cuando éste cae al vacío. Sin embargo, si los estudiantes no hicieron una buena distribución de funcionalidad no podrían simplemente crear el enemigo reutilizando componentes existentes pero es un momento para identificar y resolver el problema, aprendiendo de sus errores en ese momento. Las siguientes iteraciones siguen una idea similar donde se añaden nuevas características en las que parte de la funcionalidad debería ser reutilizada de las iteraciones anteriores.

4. Las iteraciones 4, 5 y 6 introducen bloques típicos en estos juegos en los que si el jugador golpea el bloque con la cabeza éste suelta algún tipo de recompensa. Concretamente, en esta cuarta iteración se incluye el primer bloque del juego, el cual una moneda que se mueve hacia arriba, luego hacia abajo y finalmente desaparece.

5. La quinta iteración introduce un nuevo tipo de bloque y el elemento que éste lanza es una estrella que se mueve hacia la derecha y, si el jugador la toca, muere, de la misma manera que muere al caer al vacío o al tocar un enemigo.

6. La sexta iteración añade el último bloque del juego que suelta un pelota que, cuando entra en contacto con el jugador, le aporta la habilidad de lanzar pequeñas piedras que desaparecen tras unos segundos.

7. La última iteración provoca que los enemigos mueran (desaparezcan) cuando una de las rocas colisiona con ellos.

Todas estas iteraciones comparten bastante funcionalidad con iteraciones pasadas por los que propicia que los estudiantes se den cuenta de sus propios errores en la distribución de componentes. Por ejemplo, un estudiante puede haber creado un componente específico y no flexible para el bloque de la cuarta iteración que lanza monedas (el bloque las

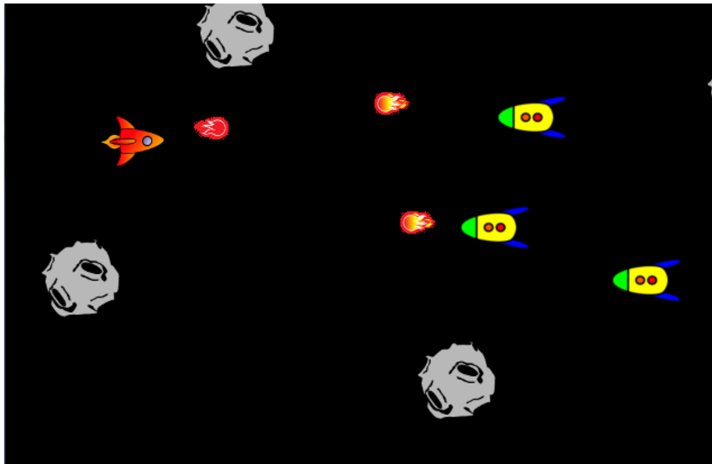

Figura 7.2: Segundo juego, construido en C++ desde cero

instancia cuando el jugador colisiona con él) pero, las iteraciones quinta y sexta le harían (en ciertos casos) parametrizar este componente para que le permitiese definir, por medio de un atributo, qué tipo de entidad se debe instanciar cuando sucede la colisión. Otros ejemplos son que el alumno puede reutilizar funcionalidad previa para mover los objetos que salen de los bloques, matar al jugador, hacer que los elementos desaparezcan, etc. En conclusión, los estudiantes aprenden de sus malas distribuciones de funcionalidad y pueden corregirlas durante las iteraciones siguientes.

Pensamos que esta primera parte del experimento es un buen punto de partida por varios motivos. *Unity* simplifica el desarrollo proporcionando una gran cantidad de soluciones integradas en forma de componentes y gestión de otros muchos aspectos del desarrollo y, por otro lado, usando *Rosette* los estudiantes pueden concentrar sus esfuerzos en la descripción semántica de las entidades, la distribución de componentes, etc. Por último, pero no menos importante, cuando los alumnos tienen que refactorizar la distribución de componentes, *Rosette* simplifica la tediosa tarea de implementación, permitiendo que puedan probar diferentes combinaciones.

7.1.3.2. Segundo juego: C++ desde cero

Como se discutió previamente, el principal propósito del máster de videojuegos es que los estudiantes aprendan todas las piezas de la arquitectura de un videojuego. Por tanto, durante todo el año, los estudiantes desarrollan un juego usando C++ donde el contenido y la funcionalidad

se crea de manera incremental. Durante el curso los estudiantes van añadiendo a su juego diferentes piezas software que se explican en las clases teórico-prácticas que reciben.

Usar tecnología C++ desarrollada prácticamente desde cero significa que los estudiantes pierden todas las facilidades que les ofrece un motor de juego como *Unity*, que consta de un editor de niveles, lenguajes de script, gestión de entidades o diferentes subsistemas gráfico, físico, de sonido, etc. ya integrados. Sin embargo, la mayor diferencia con respecto a nuestra propuesta de enseñanza es que los estudiantes comienzan con una capa lógica totalmente vacía, donde no hay componentes que vengan integrados con el motor. Los estudiantes necesitan diseñar y desarrollar esos componentes más técnicos, que Unity proporcionaba, y enfrentarse a los diferentes subsistemas (por ejemplo el motor gráfico, responsable de renderizar la escena).

Para esta segunda parte del experimento, los estudiantes tenían que crear un juego en dos dimensiones de acción y desplazamiento horizontal donde el jugador controla una nave que dispara a los enemigos, mientras estos a su vez disparan al jugador. El género de los dos juegos es parecido y comparten muchas características: ambos son juegos en dos dimensiones con un personaje/nave que se mueve y dispara. En ese sentido los estudiantes fueron capaces de extrapolar el conocimiento aprendido en la fase anterior del experimento y aplicarlo en esta segunda parte. El juego, sin embargo, es lo suficientemente diferente como para que presente un reto que haga a los estudiantes pensar acerca de cuál es la mejor distribución de componentes, ya que la anterior distribución no puede usarse tal y como había sido diseñada.

Esta fase del experimento funciona como la primera: Los estudiantes tenían que crear el videojuego en varias iteraciones. En este caso el desarrollo estaba formado por cinco iteraciones:

1. En la primera iteración los estudiantes tenían que crear una primera distribución que incluía varios asteroides estáticos, una nave que avanzaba automáticamente por el nivel pero cuyo movimiento podía alterarse moviéndose adelante, atrás, arriba o abajo mediante el uso del teclado y la lógica que hacía a la cámara avanzar a través del nivel a la misma velocidad que el jugador. Esta primera iteración se parece bastante a la primera iteración del juego anterior, pero con la dificultad extra de que tienen que crear todos los componentes (incluidos los componentes técnicos como los relacionados con la renderización).

2. La segunda y la tercera iteración prácticamente no modifican nin-

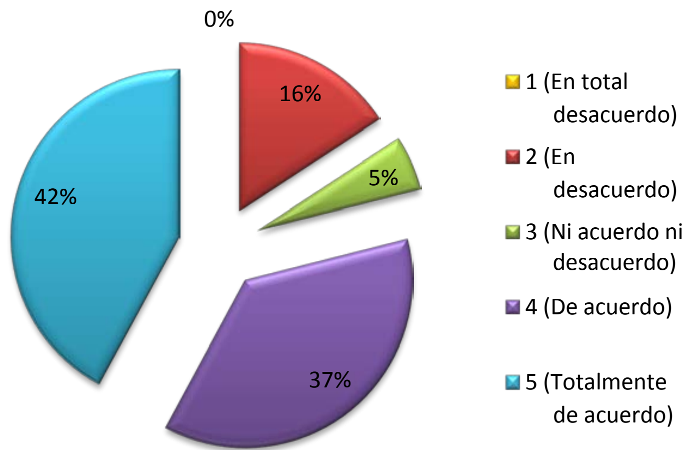

Figura 7.3: Declaración: *Rosette* y la metodología, para enseñar (o aprender) qué es una arquitectura basada en componentes, es muy útil

guna de las entidades ya existentes, pero añaden nuevas entidades. En la segunda iteración los estudiantes tenían que añadir un nuevo tipo de asteroide que se mueve de forma aleatoria hacia delante, detrás, izquierda y derecha.

3. En la tercera entraban en juego los enemigos, que son otras naves que se mueven como los asteroides de la segunda iteración pero que además disparan de manera aleatoria, característica (la de disparar) que también se le añadía a la nave del jugador cuando este pulsaba la tecla espacio.

4. La cuarta iteración introduce colisiones porque hasta este momento teníamos naves, asteroides y disparos pero nada colisionaba con nada, todos los elementos se atravesaban unos a otros. Esas colisiones implican que las naves enemigas y los disparos desaparezcan cuando colisionen con cualquier otra entidad visual, mientras que la nave del jugador es capaz de resistir hasta tres colisiones antes de morir y de que el juego termine.

5. Finalmente, en la quinta iteración, se añaden un *power-up*. El jugador en esta nueva iteración empieza sin balas con las que disparar a los enemigos y, cuando colisiona con el *power-up*, recibe 100 balas.

Igual que en el juego hecho en *Unity*, la idea era que los estudiantes fuesen iterativamente identificando similitudes entre las diferentes entidades, con el fin de construir componentes con la máxima cohesión y en

mínimo acoplamiento entre ellos para potenciar la reutilización. En este juego había un buen número de características que pertenecían a más de una entidad. Ejemplos de ésto son la velocidad constante que usan el jugador, la cámara y los disparos, la habilidad de disparar de las naves enemigas y el jugador, las habilidades de renderizarse o detectar colisiones que casi todas las entidades tienen, la capacidad de reaccionar ante colisiones para ser destruido, etc.

La clave en esto es que los estudiantes fuesen capaces de identificar buenos candidatos de componentes en los diferentes momentos del desarrollo. Un ejemplo de esto es que si ellos encapsulaban el movimiento del jugador en un componente que recibía mensajes para moverse, luego podían reutilizar ese componente para el movimiento de los asteroides y de los enemigos pero, si no lo hicieron bien en la primera iteración, podían aún hacerlo en la segunda o tercera iteración a base de refactorizar la distribución. Esto significa que podrían hacer todo bien desde el principio y finalizar el juego en un corto periodo de tiempo o, como no son expertos sino estudiantes, podrían ser capaces de aprender a distribuir las características del juego a partir de malas distribuciones que corregirían en futuras iteraciones. Además, deben pensar desde un punto de vista de alto nivel con *Rosette*, por lo que debería de ser más fácil identificar similitudes (ya que la funcionalidad no está ofuscada en el código) y también refactorizar e intentar diferentes distribuciones de componentes (ya que *Rosette* trabaja con el código por ellos).

7.1.4. Evaluación

Antes de realizar el experimento, nuestros estudiantes conocían la teoría de la arquitectura basada en componentes en videojuegos y habían trabajado con *Unity* creando algunos prototipos de manera guiada. Sin embargo, cuando se enfrentan con la creación de su primer videojuego, tienen serias dificultades para dividir la funcionalidad en componentes, lo que significa que las clases teóricas y prácticas anteriores no eran suficientes. En esta sección mostraremos los resultados del aprendizaje una vez que los estudiantes se han familiarizado con *Rosette* para crear los juegos descritos en las anteriores secciones. Lo que trataremos de hacer es medir si los estudiantes han aprendido correctamente como crear una colección de componentes.

En primer lugar, debemos confesar que una de las cosas que afectó positivamente a la enseñanza fue la alta motivación de los estudiantes. La motivación fue alta debido al hecho de que los estudiantes debían desarrollar los juegos por ellos mismos, al contrario de otros años donde tenían que completar partes de un juego desarrollado por otra gente.

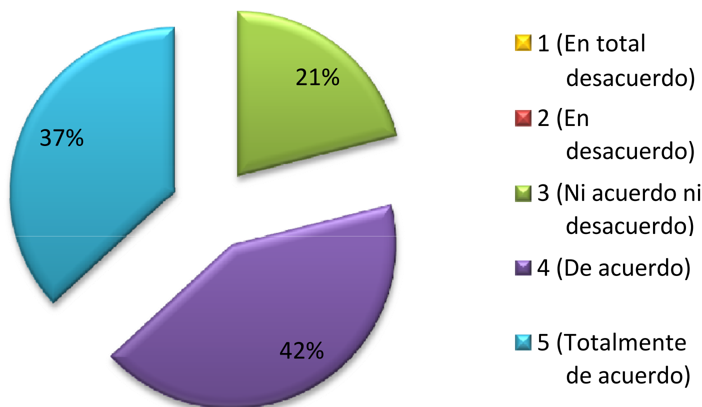

Figura 7.4: Declaración: La generación iterativa de contenido de *Rosette* es muy útil cuando el dominio es modificado, ya que preserva todo el código implementado.

Realmente creemos que es una de las claves y si hubiésemos hecho a los estudiantes trabajar sólo desde el punto de vista semántico, sin implementar el juego, el resultado no habría sido el mismo.

Para evaluar nuestra técnica de enseñanza, queremos medir la progresión de los estudiantes en dividir responsabilidades entre componentes. Para hacer esto recogimos información de dos tipos:

1. Repartimos cuestionarios al final de las clases para estimar la utilidad de la herramienta tal y como ellos la percibieron.

2. Recopilamos datos de cada iteración, incluyendo código, el modelo semántico del dominio creado en *Rosette* y trazas del uso de la herramienta.

El cuestionario se componía de varias preguntas cualitativas y cuantitativas (en escala Likert, donde se valora de 1 a 5 como de acuerdo se está con la pregunta formulada). La Figura 7.3 muestra que casi el 80 % de los estudiantes consideraron que la metodología de las clases fue útil mientras que sólo un 16 % consideró que trabajar desde un punto de vista de alto nivel (usando *Rosette*) no les aportó nada y que preferirían haber trabajado directamente con scripts de *Unity* o C++ (eso comentaron en las preguntas cualitativas). La Figura 7.4 refleja que la mayoría de ellos (nuevamente el 80 %) encontraron útil tener la posibilidad de refactorizar la distribución de componentes desde un punto de vista semántico sin pagar el precio de reimplementar la funcionalidad previamente programada y que fue movida a otro componente. En

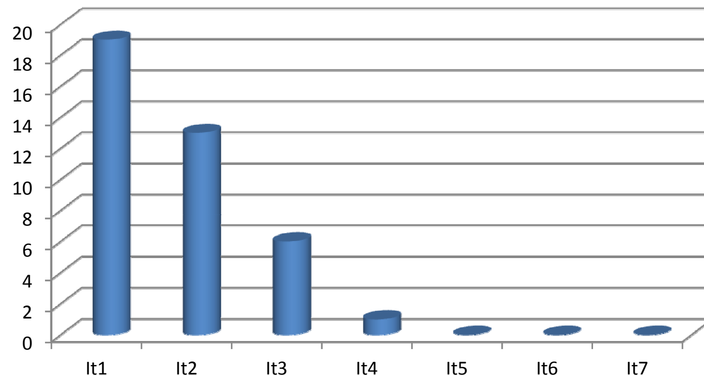

Figura 7.5: Número de estudiantes que finalizaron las diferentes iteraciones del primer juego

este caso ningún estudiante consideró esta característica como algo no deseable.

Por otro lado, hemos usado después los datos recolectados para observar el resultado de nuestra metodología enfocándola desde diferentes puntos de vista. Nuestra premisa es que si la distribución de componentes creada por los estudiantes durante las diferentes iteraciones es suficientemente buena, ellos habrán aprendido adecuadamente cómo funciona una arquitectura basada en componentes y cómo hacer un buen diseño de una de éstas. Una forma de medir la calidad de los componentes creados es ver su reutilización y su flexibilidad por lo que, si los componentes creados en una iteración han sido reutilizados para implementar nuevas entidades en posteriores iteraciones podemos inferir que son buenos componentes. Del mismo modo, si hay código replicado en diferentes componentes podremos asumir que la distribución no es suficientemente buena, ya que hay un buen candidato a ser un componente independiente. Podemos también asumir que si los errores cometidos en las primeras iteraciones fueron solventados en las últimas, el estudiante habrá identificado los fallos y aprendido de ellos.

La primera parte del experimento, aquella en la que se usaba *Unity*, fue más complicada para los estudiantes ya que ninguno de ellos fue capaz de finalizar todas las iteraciones de ese primer juego (Figura 7.5). De hecho muchos de ellos no pasaron siquiera de la segunda. El 31 % de ellos sólo finalizaron la primera iteración mientras que el 42 % solo alcanzó la segunda, el 21 % la tercera y sólo un 5 % de ellos terminó la cuarta iteración. Lo peor de todo fue el hecho de que las distribuciones de funcionalidad en componentes no fueron muy buenas en general y dedicaron mucho tiempo a diseñar los componentes. La buena noticia

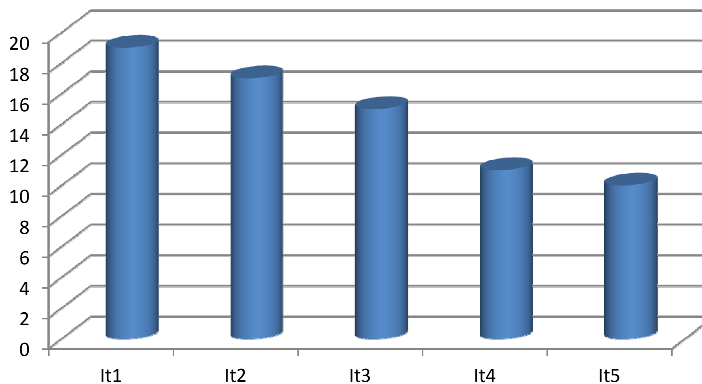

Figura 7.6: Número de estudiantes que finalizaron las diferentes iteraciones del segundo juego

es que algunos de ellos mejoraron sus distribuciones en medio de una iteración o cuando una nueva funcionalidad era añadida (pero apenas el 30 % de ellos). Lo más difícil para ellos fue encontrar similitudes entre habilidades de diferentes entidades para reutilizar componentes. Casi todos los estudiantes crearon varios componentes que incluían código y funcionalidad muy similar. Por ejemplo, ningún estudiante se dio cuenta de que el movimiento del primer enemigo del juego podía ser modelado usando el componente que se usaba para mover las plataformas y, por tanto, todos ellos acabaron con trozos de código similares en diferentes componentes. Otro ejemplo es que solamente un estudiante identificó que el código que se usaba para "matar" al jugador cuando se caía al vacío podía ser reutilizado con los enemigos, los cuales mataban al jugador al colisionar con él.

Sin embargo, cuando los estudiantes empezaron con el segundo juego, parecía que habían aprendido más de lo que en un momento nosotros pensamos. La Figura 7.6 muestra que hubo una gran cantidad de ellos que terminaron todas las iteraciones del juego (más de un 50 %) y la distribución de componentes en el segundo juego, aunque no era perfecta, era sustancialmente mejor. Los estudiantes emplearon menos tiempo en *Rosette* distribuyendo la funcionalidad en componentes y emplearon más tiempo tratando con C++ y la arquitectura del juego.

Además de la información anterior analizamos los datos recogidos de las iteraciones con el fin de inferir la calidad de las distribuciones que hicieron los alumnos. Para este propósito analizamos en cada iteración el dominio semántico, el código fuente y las trazas generadas por *Rosette*. Echando un vistazo al dominio de una iteración se puede suponer si la distribución de componentes parece buena o no, porque podemos de-

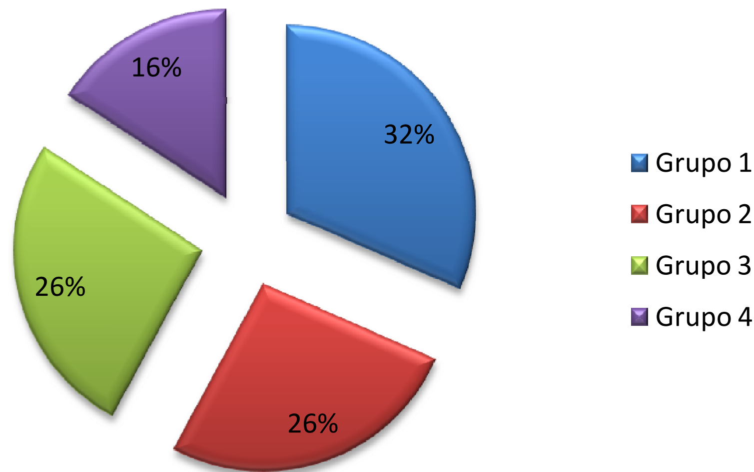

Figura 7.7: Grupos de estudiantes con diferentes progresión y calidad en sus distribuciones de componentes

tectar funcionalidad que esté duplicada desde un punto de vista semántico (por ejemplo cuando dos componentes tienen los mismos atributos y mensajes para realizar la misma funcionalidad en diferentes entidades). Sin embargo, aunque la vista que ofrece *Rosette* es buena para detectar malas distribuciones, no asegura que en caso contrario sea una buena distribución de componentes. Por tanto, tenemos que además mirar el código fuente de los componentes buscando trozos de código duplicado que revele posibles candidatos para crear nuevos componentes. Finalmente, en las trazas de *Rosette* podemos ver distribuciones en medio de una iteración y ver si los estudiantes refactorizaron el dominio a mitad.

De acuerdo con nuestro análisis de datos, hemos clasificado a los estudiantes en cuatro grupos (Figura 7.7) en función de su progresión durante el desarrollo y la calidad de las distribuciones de componentes que hicieron durante las iteraciones. Los estudiantes de cada grupo se caracterizan por:

- **Grupo 1:** Fueron capaces de identificar desde el principio los componentes reutilizables que el diseño del juego debería haber usado. Estos estudiantes no tenían errores partiendo la funcionalidad en componentes y, durante las iteraciones, simplemente necesitaban realizar muy pequeñas mejoras, añadiendo parámetros para ajustar el comportamiento del componente.

- **Grupo 2:** Estos estudiantes no crearon una distribución perfecta en un primer intento pero, según las iteraciones avanzaban, refac-

torizaban la distribución, acabando con una igual de buena que la alcanzada por el primer grupo.

- **Grupo 3:** Los estudiantes de este tercer grupo no fueron capaces de clasificar claramente todas las características del juego; identificaron algunos buenos componentes que fueron luego reutilizados en iteraciones posteriores pero también les faltó por detectar componentes importantes como el que maneja el movimiento, encargado de establecer la posición de la entidad en función de su velocidad (este componente podría reutilizarse en el jugador, nave de los enemigos y los asteroides).

- **Grupo 4:** La distribución de este pequeño grupo de alumnos no pasaba el mínimo exigible para ser considerada una distribución aceptable, debido a que usaba componentes grandes y no reutilizables o debido a que no habían entendido para nada como funciona la arquitectura basada en componentes.

En la Figura 7.7 podemos ver que casi el 60 % de estudiantes acabaron con una distribución muy buena, ya la diseñasen bien desde el principio o acabasen con ella tras alguna refactorización. Además, más del 80 % de los alumnos acabaron con una distribución aceptable, aunque las de alguno de ellos no era perfecta.

7.1.5. Conclusiones

En esta sección hemos presentado un método que hemos usado para enseñar la arquitectura basada en componentes para desarrollo de videojuegos en un máster para estudiantes licenciados en informática. Los estudiantes se sienten cómodos usando un diseño orientado a objetos, usando jerarquías, pero necesitan adaptar sus hábitos a la nueva filosofía impuesta por la arquitectura.

Nuestro método consiste en enfrentar a los estudiantes con el desarrollo de un par de pequeños juegos que deben implementar por si mismos, en lugar de hacerlo paso a paso de una manera guiada. Aunque los juegos propuestos son suficientemente simples, no era realista esperar que los estudiantes fuesen capaces de experimentar con diferentes distribuciones de componentes y, al mismo tiempo, escribir todo el código en un pequeño periodo de tiempo.

La introducción de *Rosette* sirve para simplificar el problema permitiendo trabajar desde el punto de vista semántico y acelerando el ciclo de desarrollo del videojuego, algo muy útil durante las primeras fases

del aprendizaje de la arquitectura basada en componentes, donde la refactorización de componentes es bastante común. Los estudiantes estaban motivados durante la sesión ya que tenían la oportunidad no solo de diseñar un buen sistema basado en componentes sino también probar el juego según evolucionaba.

Aunque los estudiantes obviamente no se pueden convertir en expertos en 12 o 15 horas de clase, los resultados prueban que han adquirido los fundamentos de la arquitectura basada en componentes e interiorizado sus mecanismos de una mejor manera que hacían los estudiantes de anteriores ediciones del máster (que no usaban *Rosette* ni este método). De hecho, en ediciones posteriores del máster se ha seguido manteniendo esta metodología de enseñanza debido a su buena aceptación. Por otro lado, *Rosette* ha mostrado por sí mismo que es una herramienta útil no solo para enseñar cómo desarrollar videojuegos sino para desarrollar juegos en general; alguno de los estudiantes siguieron usando *Rosette* para el desarrollo de sus juegos de fin de máster, incluso meses después de que acabasen las clases de introducción a la arquitectura de componentes.

7.2. Evaluación empírica de la identificación automática de componentes software

7.2.1. Introducción

En esta sección queremos responder la pregunta: "¿Ayuda nuestra técnica de sugerencia de componentes a crear mejores distribuciones de componentes que mejoren la calidad del software?" y para lograr ese objetivo hemos llevado a cabo un experimento usando *Rosette*.

Aunque *Rosette* y la técnica de sugerencia de componentes se ha concebido para desarrollar juegos de un tamaño medio o incluso para grandes desarrollos, no podemos permitirnos realizar un experimento de dicha magnitud. Por tanto, se llevó a cabo un experimento menos ambicioso pero que trataba de simular un desarrollo completo de un videojuego en un entorno más definido y controlado.

Para averiguar la validez de la sugerencia de componentes se contó nuevamente con estudiantes del máster en desarrollo de videojuegos de la Universidad Complutense de Madrid, los cuales se dividieron en dos grupos, los que usaban la técnica y los que no. Ambos grupos tenían la ayuda de *Rosette* para poder realizar el juego en el menor tiempo posible, lo que nos permitió recoger datos del desarrollo en las diferentes iteraciones.

Para cuantificar la calidad de las distribuciones de componentes durante las diferentes iteraciones se han usado distintas medidas que comparan las distribuciones realizadas por los dos grupos de personas en las diferentes iteraciones.

7.2.2. Metodología

El experimento llevado a cabo trata de simular las fases iniciales del desarrollo de un videojuego. El diseño del experimento refleja la naturaleza volátil del proceso de desarrollo: durante cinco iteraciones diferentes, los sujetos del experimento se enfrentan al desarrollo de un pequeño juego donde la complejidad de las entidades va creciendo. Si los participantes del experimento fallan en el diseño de un buen sistema de componentes en las primeras iteraciones, se verán forzados a refactorizar la distribución en futuras iteraciones o duplicar y escribir una gran cantidad de código terminando con una mala distribución de componentes. Este escenario nos permite medir si la sugerencia de componentes ayuda o no ayuda en el proceso.

El experimento consistía en el desarrollo de un juego de acción en dos dimensiones de desplazamiento horizontal donde el jugador maneja una nave y dispara a los enemigos mientras evita colisionar con los asteroides. Sin embargo, como se ha dicho antes, en vez de tener el diseño completo desde el principio, la especificación del juego va evolucionando durante cinco iteraciones. El juego que se desarrolla es el mismo que se desarrolló en el experimento anterior, más concretamente el descrito en la Sección 7.1.3.2, donde la descripción de las iteraciones y el resultado esperado en términos de componentes son los siguientes:

1. La primera versión del juego debía mostrar una nave volando de izquierda a derecha con la cámara moviéndose a una velocidad constante en la misma dirección. También aparece algún asteroide estático. Un diseño de componentes aceptable incluye (1) un componente gráfico, común a la nave y los asteroides, para pintarse a sí mismos, (2) un componente controlador, que captura los eventos de movimiento y mueve la nave acorde a ellos, (3) un componente que controla la lógica de la cámara actualizando su posición en el motor gráfico y (4) un componente de movimiento constante que mueve la entidad a la que pertenece y que tanto la nave del jugador como la cámara usan.

2. La siguiente iteración consiste en la creación de un movimiento aleatorio que se aplica a ciertos asteroides. Eso implica la creación

de un nuevo componente que, en cierto sentido, juega el rol de una simple inteligencia artificial de los asteroides que los mueve por el espacio de manera aleatoria.

3. La tercera iteración añade enemigos en forma de naves, que aparecen por la parte derecha de la pantalla, y la habilidad para las naves enemigas y del jugador de disparar balas. Desde el punto de vista del desarrollo esto fuerza a (1) la creación de nuevos tipos de entidades para los enemigos y para las balas, (2) desarrollar un componente que gestione la aparición de las balas, usado tanto por los enemigos como por el jugador, (3) reutilizar el componente con la inteligencia artificial de los asteroides en las naves enemigas, (4) reutilizar el componente que mueve la entidad a una velocidad constante en las balas y (5) modificar los componentes de la inteligencia artificial y el del control de movimiento para que emitan mensajes de disparo cuando se necesiten.

4. La cuarta iteración introduce una mínima detección de colisiones con el fin de que las naves reaccionen ante las balas cuando colisionan con ellas. La solución esperada consiste en la adición de un componente de física/colisión y otro de vida que reacciona ante los mensajes de colisión generados por el primer componente.

5. La última iteración se añade un objeto al escenario. El jugador empieza la partida sin poder disparar y sólo cuando coge dicho objeto obtiene balas para disparar. El desarrollo de esta última mecánica de juego es fácil si previamente se construyó una colección de componentes bien pensada. En particular no son necesarios nuevos componentes, simplemente la modificación de los existentes y la adición de algún nuevo mensaje.

Para el experimento reunimos a un total de 18 alumnos del máster en desarrollo de videojuegos en la Universidad Complutense de Madrid. En el momento en el que el experimento tuvo lugar, todos los alumnos habían recibido clases acerca de la arquitectura de juegos, de los pros y contras del uso de la herencia cuando se desarrolla usando una jerarquía de entidades y los conceptos básicos de la arquitectura basada en componentes. Además los alumnos habían desarrollado pequeños prototipos de manera guiada y un pequeño juego por su cuenta (todo ello usando componentes).

El juego tenía que ser creado en C++ y tenían a su disposición un esqueleto de juego que incorporaba una mínima aplicación que gestionaba el bucle principal del juego, un servidor gráfico que encapsulaba

las labores de renderizado,y una pequeña capa lógica que gestionaba entidades basadas en componentes. La capa lógica era capaz de cargar descripciones de entidades que se construían en tiempo de ejecución y mapas de los distintos niveles donde se describían las entidades. Ambas descripciones se encontraban en ficheros externos. Sin embargo no había ningún componente por lo que los estudiantes tenían total libertad para diseñar las habilidades de cada componente desde cero. El uso de *Rosette* ayudaba en la creación de los componentes no solo desde el punto de visa conceptual sino también generando código C++ que se adaptaba perfectamente a la capa lógica proporcionada a los estudiantes.

Para evaluar la técnica de sugerencia de componentes de *Rosette*, separamos a los estudiantes en dos grupos donde cada estudiante desarrollaba el juego de manera individual. El *grupo de control* usó una versión restringida de *Rosette*, donde el módulo de sugerencia de componentes estaba deshabilitado, mientras que el *grupo experimental* usaba *Rosette* tal y como es, con todos sus módulos habilitados. Además usamos de referencia una solución del juego realizada por un experto.

7.2.3. Evaluación

Como se ha comentado, el objetivo de este estudio es analizar el impacto de nuestra técnica de sugerencia de componentes y si ayuda o no a crear mejores distribuciones de los mismos. Con la finalidad de comparar las distribuciones de los integrantes del grupo de control con los del grupo experimental necesitamos una manera imparcial de medir la calidad de las distribuciones.

Una *buena distribución de componentes* se caracteriza por la reusabilidad de sus componentes, su flexibilidad ante los cambios en el dominio y la mantenibilidad y extensibilidad del software resultante. La reusabilidad y la mantenibilidad se incrementan cuando la *cohesión* de cada componente se maximiza y, al mismo tiempo, el acoplamiento entre los componentes (y con otras clases) se minimiza. La duplicación del código es perjudicial para la mantenibilidad, y la extensibilidad es inversamente proporcional al número de cambios necesarios para añadir una nueva característica.

Para medir todos esos aspectos en nuestro experimento hemos usado la siguiente información:

- **Código y contenido de juego generado:** Hemos recopilado los dominios de *Rosette* y el código y demás contenido de juego generado en cada iteración. Esta información la usamos para ver la

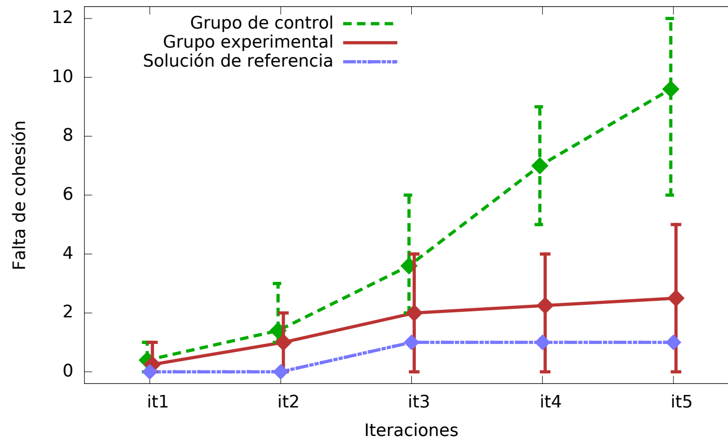

Figura 7.8: Falta de cohesión de las distribuciones por iteración

cohesión y el *acoplamiento*, que denotan deficiencias en la flexibilidad, y también buscamos en ella duplicaciones de código que daña la mantenibilidad de las implementaciones de los estudiantes.

- **Trazas:** También hemos recopilado trazas de *Rosette*, que reflejan cuál fue el uso que hicieron los estudiantes de la herramienta en las diferentes iteraciones. Estas trazas han sido usadas para analizar el esfuerzo necesitado para modificar o extender el software creado por los estudiantes.

- **Cuestionarios:** Al final del experimento dimos a los alumnos unos cuestionarios para recoger su opinión acerca de *Rosette* y su utilidad.

En los siguiente apartados describimos los resultados obtenidos en cada uno de los aspectos analizados.

7.2.3.1. Cohesión

Desde el punto de vista de programación la *cohesión* hace referencia al grado con el que los elementos de un módulo (clase o componente) se encuentran ligados (Yourdon y Constantine, 1979) o, dicho de otra manera, hace referencia a la forma en que agrupamos unidades de software (módulos, subrutinas...) en una unidad mayor. Con el fin de evaluar la *falta* de cohesión del software generado por los participantes del experimento, hemos usado la medida *Lack of Cohesion in Methods (LCOM)*, descrita en Hitz y Montazeri (1995); Chidamber y Kemerer (1994) como “el número de pares de métodos que operan sobre diferentes conjuntos

de variables, reducido por el número de pares de métodos que actúan sobre al menos una instancia de variable en común”. LCOM se aplica a la clases individualmente y su valor es mayor cuanto mayor es la falta de cohesión en dicha clase.

Aquí podemos ver un ejemplo simplificado extraído del código de uno de los participantes:

```
class Controller : IComponent {
private:
   float _speed;
   Entity* _bullet;
public:
   void VerticalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(_speed*secs,0);
      setPosition(pos);
   }
   void HorizontalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(0,_speed*secs);
      setPosition(pos);
   }
   void Shoot() {
      EntityFactory f = EntityFactory::getInstance();
      Entity e = f->instanciate(bullet, getPosition());
      f->deferredDeleteEntity(e, 2.0f);
   }
}
```

En este código, el componente *Controller* proporciona dos funcionalidades diferentes: las habilidades de moverse y disparar. Esto constituye una *falta* de cohesión, como se demuestra cuando se calcula el coeficiente LCOM de este componente: aunque ambos métodos (*VerticalMove* y *HorizontalMove*) usan el atributo *_speed*, los pares (*VerticalMove*, *Shoot*) y (*HorizontalMove,Shoot*) no comparten ningún atributo. Por tanto LCOM tiene un valor de 1 $(= 2 - 1)$.

Hemos calculado el coeficiente LCOM de cada componente del software generado por los estudiantes en las cinco iteraciones y luego hemos sumado los coeficientes de todos esos componentes para mostrar la evolución de la cohesión durante el desarrollo del juego. La Figura 7.8 presenta la media de los resultados de los dos grupos de participantes, donde podemos apreciar que en la primera iteración la cohesión es si-

milar entre los dos grupos pero, cuando el desarrollo evoluciona, el software generado por los estudiantes del grupo de control, sin la ayuda de la técnica de sugerencia de componentes, muestra mucha menos cohesión. Esos alumnos solían crear componentes más grandes y, como sus funcionalidades estaban menos unidas, eran mucho menos reutilizables, lo que significa que los componentes eran específicos sólo para el juego que estaba en desarrollo (e incluso sólo para la iteración en curso).

Con la finalidad de comprobar la significación estadística de los resultados, calculamos el valor P para los dos grupos en las tres últimas iteraciones. El valor P es la probabilidad, bajo la hipótesis vacía, de obtener un resultado al menos tan extremo como el que realmente se observa (Goodman, 1999). Cuando el valor P es menor que 0.05 (Stigler, 2008) indica que el resultado sería improbable bajo la hipótesis vacía. El valor P para las iteraciones 3, 4 y 5 fueron 0.076675, 0.0014788 y 0.00099428, lo que muestra que la probabilidad de que los valores de los dos grupos pertenezcan a la misma población decrece con el desarrollo del juego y en las dos últimas iteraciones son diferentes desde el punto de vista de la significación estadística (valor $P < 0.05$).

La Figura 7.8 también muestra el coeficiente LCOM de la solución que fue usada como referencia de una buena distribución. La figura muestra que el grupo experimental, ayudado por la técnica de sugerencia de componentes, terminó con una distribución similar a la de referencia (en cuanto a la cohesión se refiere).

7.2.3.2. Acoplamiento

El acoplamiento es el grado en el que cada módulo de programa depende de cada uno de los otros módulos y es habitualmente contrastado con la cohesión. Hitz y Montazeri (1995) define que "cada evidencia de un método de un objeto usando métodos o instancias de variables de otro objeto constituye acoplamiento". Además proponen una manera de categorizar el acoplamiento entre clases y objetos considerando ciertas reglas; desafortunadamente dichas reglas no son válidas para una arquitectura basada en componentes.

La definición dada por Lee et al. (2001) encaja mejor: "Cuando un mensaje se transmite entre objetos, se dice que esos objetos están acoplados. Las clases se acoplan cuando los métodos declarados en una clase usan los métodos o atributos de otras clases". Además, el artículo propone una *medida de acoplamiento* similar a la que usamos para la cohesión, en lugar de una categorización como se proponía en Hitz y Montazeri (1995). Lee et al. (2001) define el *valor de acoplamiento*

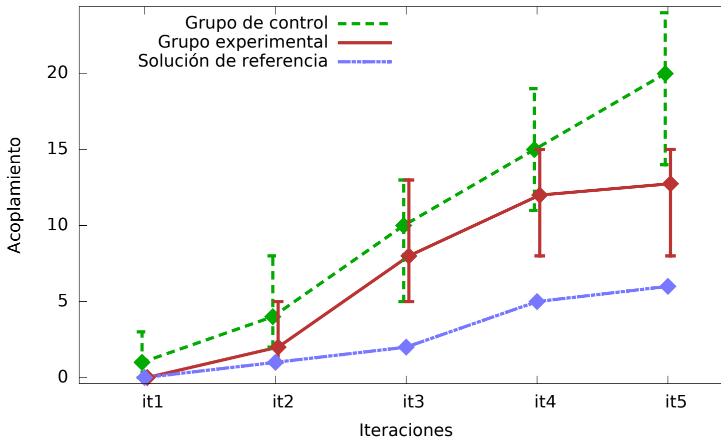

Figura 7.9: Acoplamiento de las distribuciones por iteración

(Pi) como el *número de métodos invocados* (que es diferente al número de *invocaciones de métodos*) de una clase a otra y la *interacción del acoplamiento* se define como:

$$interaccin_del_acoplamiento = \sum_{TodosLosCasosDeUso} Pi$$

Donde cada caso de uso define interacciones entre un componente y el sistema para alcanzar un objetivo extraído de los diagramas de secuencia.

Lee et al. (2001) también define el *acoplamiento estático*, el cual es causado por asociaciones de clases, composición o herencia. Para el cálculo asignan valores a diferentes relaciones: 1 cuando dos clases comparten la misma clase padre, 2 cuando una clase hereda de otra, 3 cuando hay una asociación direccional entre dos clases, 4 cuando la asociación entre dos clases es bidireccional y 5 cuando una clase es una agregación de otra.

Las arquitecturas basada en componentes añaden una nueva forma de acoplamiento de clases debido a las relaciones dinámicas. El paso de un mensaje entre componentes de la misma entidad constituye un acoplamiento entre clases hermanas por lo que, teniendo en cuenta los valores asignados por Lee et al. (2001), el valor del *acoplamiento estático* es 1. Por otro lado, cuando un componente accede a la entidad a la que pertenece está explícitamente usando agregación entre ellos, por lo que el valore del *acoplamiento estático* es 5.

Una vez que se ha definido el acoplamiento de *interacción* y el *estático*, Lee et al. (2001) definen el *acoplamiento entre dos clases* como:

$$acoplamiento_entre_dos_clases = interaccin_del_acoplamiento * acoplamiento_estatico$$

Usando el siguiente código extraído de una las las soluciones de los participantes mostramos un ejemplo de como se calcula el *acoplamiento entre dos clases*:

```
class AI : IComponent {
public:
   void tick(unsigned int msecs) {
      float _hMove,_vMove;
      if(!_entity->getType().compare("MovingAsteroid")) {
         _hMove = rand() % 10 - 5;
         _vMove = rand() % 10 - 5;
      }
      else if(!_entity->getType().compare("Enemy")) {
         _hMove = rand() % 20 - 10;
         _vMove = rand() % 20 - 10;
      }
      IMessage m = new HorizontalMove(_hMove);
      sendMessage(m);
      m = new VerticalMove(_vMove);
      sendMessage(m);
   }
}
```

La *interacción del acoplamiento* entre las clases *Entity* y *AI* tiene el valor 1 ya que *AI* invoca el método *getType()* y hay simplemente un caso de uso que involucra a las dos clases.

Por otro lado, el componente *AI* accede a la entidad a la que pertenece (el atributo heredado *_entity*) usando el conocimiento acerca de la agregación entre las entidades y componentes, lo que constituye un *acoplamiento estático* de 5. Estos dos valores provocan que el valor del *acoplamiento entre dos clases* sea 5 (= 1 * 5).

El componente *AI* envía los mensajes *HorizontalMove* y *VerticalMove* que son procesados por el componente *Controller* presentado anteriormente. Esto muestra un *acoplamiento estático* entre los dos componentes (*AI* y *Controller*) de valor 1. Ambos interaccionan en un solo caso de uso, pero intercambian esos dos tipos de mensajes, por lo que tienen un *acoplamiento de interacción* de 2, lo que conlleva a que el *acoplamiento entre dos clases* tenga el mismo valor (= 2 * 1).

Usando este procedimiento, hemos calculado minuciosamente para cada participante el *acoplamiento entre dos clases* para cada componente con otras clases y hemos sumado todos los valores para obtener el *acoplamiento global* de la distribución de componentes. La Figura 7.9 presenta los resultados de la media de este *acoplamiento global* en los dos grupos durante las cinco iteraciones. Podemos apreciar que la evolución del acoplamiento es similar en los dos grupos y, desde el punto de vista de la significación estadística, para las iteraciones 3, 4 y 5 obtenemos un valor P de 0.34346, 0.12407 y 0.0015096. Estos valores muestran una tendencia creciente aunque sólo en la última iteración los valores entre los dos grupos son significativamente diferentes. Sin embargo, mirando detenidamente el código observamos que las razones para el acoplamiento obtenido son diferentes. El acoplamiento del software generado por el grupo experimental es debido a que la alta cohesión de sus componentes previene la duplicación de funcionalidad pero provoca que los componentes deban comunicarse entre sí. De hecho el acoplamiento es habitualmente contrastado con la cohesión. Por ejemplo, el componente *Controller* del ejemplo anterior es usado por el componente *AI* pero también por el jugador, el cual manda mensajes cuando se producen eventos de teclado. En este sentido, la reusabilidad del componente *Controller* es buena en ese aspecto aunque penalice ligeramente el acoplamiento entre los componentes debido al paso de mensajes.

Sería de esperar que los componentes realizados por el grupo de control, al ser componentes más grandes y con mayor funcionalidad (menos cohesión), tuviesen menor acoplamiento. Sin embargo, tras el análisis de dichos componentes concluimos que, aunque grandes, no eran suficientemente autocontenidos por lo que al reunir demasiada funcionalidad se debía acceder a otros elementos o componentes para tomar diferentes decisiones. Por ejemplo, algunos estudiantes implementaron un componente que mezclaba la funcionalidad de los componentes *AI* y *Controller* de los ejemplos anteriores. Esto podría parecer que mejoraría el acoplamiento entre componentes pero estas implementaciones requieren conocer el *tipo de entidad* concreto (*Jugador*, *Enemigo* o *Asteroide*) para decidir como se comporta el componente, lo que introduce un acoplamiento adicional innecesario.

Finalmente, sería de esperar que nuestra solución de referencia tuviese un mayor acoplamiento ya que era la solución con mayor cohesión. Sin embargo, las otras soluciones estaban muy penalizadas por el *acoplamiento estático* ya que se invoca varias veces a métodos de la entidad (generalmente para consultar el tipo de entidad) y esto multiplica

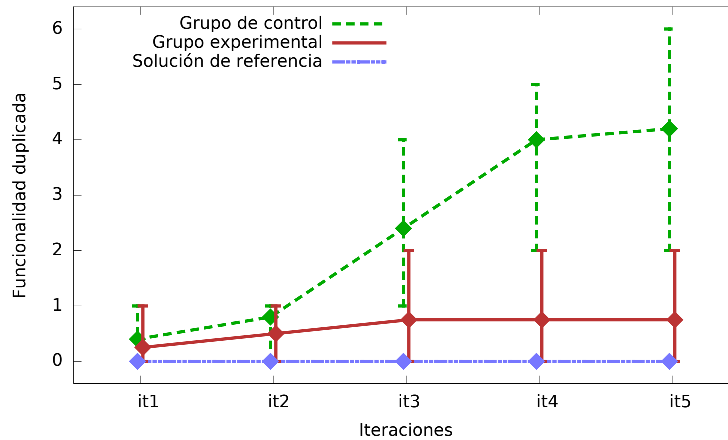

Figura 7.10: Funcionalidad duplicada en las diferentes iteraciones

por 5 el *acoplamiento de interacción*. Por tanto, aunque el número de mensajes intercambiados pos los componentes de nuestra solución son mayores, el resultado del acoplamiento es menor debido a que la medida propuesta por Lee et al. (2001) considera mucho menos peligroso para la reusabilidad este paso de mensajes que el acceso al tipo de entidad.

7.2.3.3. Duplicación

La duplicación de funcionalidad en el código fuente penaliza la mantenibilidad. Hemos analizado los componentes creados por los participantes en el experimento en busca de código duplicado y, cuando una funcionalidad (por ejemplo un método) se encuentra replicado en dos o más sitios, penalizamos el dominio. En este sentido, la medida de duplicación revela el número de trozos de código que han sido duplicados y podrían haber sido encapsulados y reutilizados por diferentes métodos y componentes.

La Figura 7.10 presenta la evolución de la funcionalidad duplicada durante el desarrollo del juego. Ilustra que los participantes del grupo experimental mantienen muy baja duplicación de código durante todo el desarrollo, mientras que los participantes que se encontraban en el grupo de control, duplicó más cantidad de código en las diferentes iteraciones. Esta figura confirma lo que revelaban las medidas de cohesión y de acoplamiento acerca de la calidad de las distribuciones de componentes y prueba que una mala distribución de componentes lleva a una mala reutilización de la funcionalidad y duplicación de código, lo cual no solo denota una pobre reusabilidad sino que también provoca una mala

mantenibilidad. El valor P entre los dos grupos para las iteraciones 3, 4 y 5 fueron 0.021433, 0.0017129 y 0.0012074, lo que confirma que el resultado no es coincidencia.

Entrando en un análisis más profundo de las iteraciones, hemos podido observar que algunos de los alumnos del grupo experimental empezaron con unas distribuciones no demasiado buenas, ya que en general algunos de los componentes que creaban tenían poca cohesión, acumulando funcionalidades de diferente índole en un mismo componente. Esto se explica debido a que *Rosette* solo puede sugerir en función de las entidades que hay en ese momento en el modelo y hace una agrupación simplemente por los datos que comparten las entidades modeladas. Aun así, para el resultado final no fue un gran problema ya que cuando se fueron añadiendo nuevas entidades en las siguientes iteraciones, *Rosette* permitió una rápida refactorización dividiendo componentes grandes en componentes más pequeños pero con mayor cohesión.

El gran incremento producido en la tercera iteración se debe principalmente a dos razones: (1) algunos participantes no se dieron cuenta de que los asteroides y las naves enemigas se movían de la misma manera y, en lugar de reutilizar la funcionalidad creada en la segunda iteración (para el asteroide), duplicaron el código; (2) alguno de ellos también duplicaron código relacionado con la habilidad de disparar (que tienen tanto el jugador como el enemigo) en lugar de parametrizarla y ponerla en un nuevo componente.

Durante la cuarta iteración el código duplicado en el grupo de control también incrementó de manera significativa porque varios estudiantes no agruparon la funcionalidad relacionada con las colisiones y duplicaron código en diferentes componentes en lugar de crear uno nuevo.

7.2.3.4. Modificabilidad y extensibilidad

El último factor que se debe tener en cuenta para evaluar la calidad de la distribución de componentes es la modificabilidad y la extensibilidad. Una buena medida de esos aspectos es el número de cambios que sufre la distribución de componentes cuando se añade una nueva funcionalidad. Hemos usado las trazas generadas por *Rosette* para recopilar dicha información. Esas trazas nos permiten medir los *cambios semánticos* en el dominio, en lugar de sólo mirar a los cambios en el código fuente, que son menos informativos. *Rosette* anota modificaciones como la creación de entidades, componentes, mensajes y atributos, la adición de componentes a las entidades, de atributos a componentes, supresión de elementos (por ejemplo quitar un atributo a un mensaje), etc.

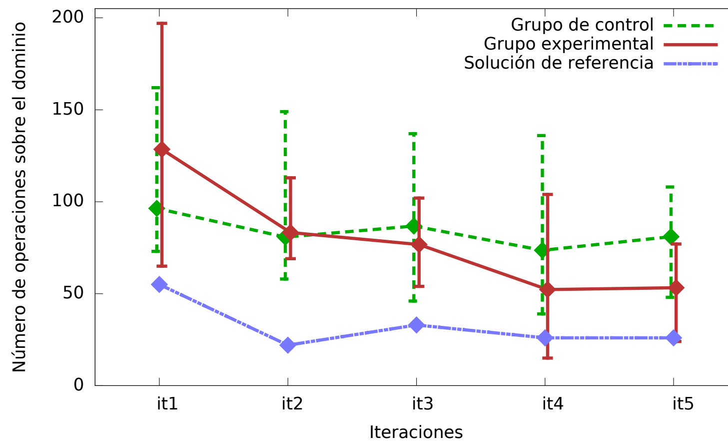

Figura 7.11: Número de operaciones hechas en cada iteración

Prestando atención al número medio de acciones requeridas para crear los dominios de ambos grupos (Figura 7.11) podemos observar que, durante las iteraciones iniciales, los participantes del grupo de control necesitaron realizar menos operaciones al dominio del juego para obtener la funcionalidad propuesta, pero en las últimas iteraciones, el número de cambios se incrementa en comparación con los participantes que se encontraban en el grupo experimental.

El mayor número de operaciones llevadas a cabo en la primera iteración puede ser explicado debido a que crear una buena distribución de componentes requiere un modelado del dominio más complejo. Aun así es importante recalcar que cuando se usa la sugerencia de componentes (grupo experimental), muchas de las operaciones del dominio *son automáticas*, por tanto un dominio más rico *no* implica mayor tiempo invertido. Los usuarios del grupo experimental obtuvieron mejores distribuciones iniciales que les permitieron tener que realizar muchos menos cambios en el dominio cuando el juego evolucionaba. Por otro lado, los usuarios que no recibieron asistencia crearon distribuciones más pobres que necesitaban un gran número de cambios (más o menos constante) en cada iteración.

En este caso, la significación estadística de los resultados no es relevante para las primeras iteraciones. Los resultados para las iteraciones 3, 4 y 5 fueron 0.35490, 0.16899 y 0.011173 lo que muestra que sólo en la última iteración los resultados de los dos grupos fueron diferentes de manera significante pero, debido a la tendencia, nos permite pensar que, si el desarrollo hubiese continuado, en las siguiente iteraciones el grupo experimental hubiese seguido requiriendo menos operaciones

que los usuarios del grupo de control.

7.2.4. Conclusiones

Nuestra intención con este experimento era probar que la técnica de sugerencia de componentes de *Rosette* ayuda a crear buenas distribuciones de funcionalidad en componentes y que la calidad del software generado es suficientemente buena en términos de flexibilidad, mantenibilidad y reusabilidad.

La información recolectada durante el experimento denota que, teniendo dos grupos de personas con la misma experiencia en desarrollo de videojuegos, aquellas personas que usaron el módulo de sugerencia de componentes de *Rosette* terminó con un software claramente mejor en cada iteración. El objetivo principal de maximizar la cohesión y minimizar el acoplamiento fue mejor llevado a cabo por ellos, viéndose además reforzado por la ausencia de duplicación de código. Sin embargo, el indicador principal de que la técnica acelera el desarrollo de videojuegos es que el número de cambios sufridos por la distribución de componentes, cuando se requieren modificaciones en el dominio, es menor cuando se usan las sugerencias propuestas por *Rosette*.

Uno de los hándicap es que la creación de una distribución de componentes prometedora al inicio parece costosa de conseguir. Aunque la sugerencia de componentes proporcionada por *Rosette* ha demostrado ser útil en este experimento y los alumnos evaluaron *Rosette* como una herramienta de uso intuitivo, alguno de los estudiantes puntualizó que el uso del módulo de recomendación de componentes no es del todo intuitivo y complica su uso en un principio. Podemos además corroborar este hecho debido a que cuatro personas del grupo experimental abandonaron el experimento, mientras que solo lo hicieron dos personas del grupo de control (estos 6 estudiantes no están contabilizados en los números ofrecidos anteriormente). Uno de los trabajos futuros es evaluar cómo podemos mejorar la curva de aprendizaje de este módulo mediante por ejemplo el uso de otro tipo de interfaz. Finalmente pensamos que debemos evaluar en el futuro la eficacia de la herramienta y de esta técnica con desarrollos más largos y que involucren grupos de personas más grandes.

Capítulo 8

Conclusiones y trabajo futuro

8.1. Contribuciones

Tras el estudio del proceso de desarrollo de videojuegos y cuál ha sido su evolución durante los últimos años detectamos que existe una carencia de mecanismos eficientes de comunicación entre los diferentes roles involucrados en el equipo de desarrollo (especialmente entre diseñadores y programadores). Este mecanismo suele ser realizado mediante documentos de diseño, donde todos los detalles del videojuego (mecánicas, jugabilidad, personajes, elementos, interacciones...) están descritos de manera textual, lo que introduce ambigüedad en muchos aspectos y provoca que la fase comunicativa sea lenta.

Por otro lado, la tecnología y metodologías usadas se han ido adaptando con el paso de los años permitiendo agilizar los desarrollos ya que, debido a la naturaleza cambiante del diseño de los videojuegos, estas tecnologías y metodologías deben ser lo más flexible posibles y permitir de manera sencilla cambios en la dirección del proyecto. Sin embargo, este proceso de flexibilización tiene un precio y la arquitectura basada en componentes, que es la elección tomada por prácticamente la mayoría de desarrollos comerciales para gestionar la lógica de juego, tiene sus problemas. Pese a ser una arquitectura mucho menos estática que sus predecesoras, conceptualmente hablando es menos intuitiva ya que mucho conocimiento semántico queda escondido tras los datos y el código y, debido a su naturaleza de composición dinámica, introduce un gran número de potenciales inconsistencias en el desarrollo.

Teniendo esto en cuenta podríamos recuperar la principal pregunta de investigación que se quiere responder con este trabajo y que fue

formulada en la introducción:

> *¿Podría un dominio formal ayudar a paliar dichos hándicaps y agilizar el desarrollo de videojuegos?*

El trabajo propuesto en esta tesis muestra como una metodología que tiene como piedra angular un dominio formal mejora el proceso en diversos aspectos:

1. **Facilidad de representación del dominio:** El uso de una herramienta específica de modelado visual facilita el trabajo tanto de diseñadores como de programadores, simplificando la edición del modelo de juego.

2. **Flexibilidad:** El dominio formal permite que la definición de las entidades de juego sean modificadas de una manera mucho más sencilla que un documento de diseño tradicional.

3. **Agilidad en la comunicación:** La comunicación entre diseñadores y programadores es más rápida donde la información puede ser visualizada en diferentes capas de definición, donde el dominio formal disminuye la ambigüedad que introducen las descripciones en lenguaje natural. Además, el uso del dominio formal por parte de los programadores les permite recuperar esa ontología conceptual que introducían los sistemas basados en herencia y que se eliminaba con la arquitectura basada en componentes (además de más tipos de información semántica).

4. **Dominio formal como contrato de mínimos:** Mediante diferentes visualizaciones del mismo dominio, que dependen del rol, los diseñadores y programadores pueden trabajar sobre el mismo dominio, de manera que éste puede verse como un acuerdo que se alcanza en común.

5. **Validación:** Al poder razonar sobre el dominio éste puede ser validado, de manera que al detectar inconsistencias se puedan subsanar posibles errores con anticipación, ahorrando trabajo de depuración en fases futuras. Esta técnica no sólo subsana las potenciales inconsistencias que se introducen con la arquitectura basada en componentes, sino que también anticipa errores durante la fase de edición de niveles.

6. **Sugerencia de distribución de funcionalidad:** A partir de las definiciones de los diseñadores (y programadores) sobre el dominio formal, se pueden identificar divisiones óptimas de funcionalidad en componentes software, acelerando así el proceso de diseño

de la arquitectura. Ésto facilita la creación de un software más flexible que permitirá un desarrollo más veloz en subsiguientes iteraciones.

7. **Generación de contenido iterativa:** La información semántica permite la generación procedimental de contenido, de manera que el desarrollo se guía por el modelo formal. Esto permite la generación de un porcentaje muy alto del código relacionado con la lógica de juego y también de contenido para las herramientas de diseño del nivel, muy útil para los diseñadores. Sin embargo, la mayor de las ventajas es la posibilidad de hacer refactorizaciones a alto nivel ya que, gracias a esta generación procedimental de código, las refactorizaciones se trasladan directamente al código de manera transparente al programador. Del mismo modo, los cambios se trasladan al resto del contenido de juego.

8. **Mejora de la calidad del software generado:** El trabajo conceptual previo ayuda a la generación de un software más reutilizable, flexible y, en definitiva, de mayor calidad. Además, la generación de código provoca que el código sea más homogéneo y de más fácil lectura. Por último, la posibilidad de hacer refactorizaciones a alto nivel incitan a las refactorizaciones si algo no fue bien concebido en lugar de "apañar" los errores de maneras poco recomendables.

8.2. Discusión de los resultados

Los experimentos presentados en el Capítulo 7 muestran en primer lugar la idoneidad de la metodología para prototipado rápido. Gracias al uso de una herramienta como *Rosette*, que soporta las ideas desarrolladas en la tesis, se pudieron llevar a cabo la construcción de pequeños juegos en iteraciones que, de otra manera, no hubiese sido posible debido a la limitación de tiempo. La representación ontológica fue gratamente valorada por los usuarios, los cuales comentaron que les ayudó a distribuir las entidades de una manera que mentalmente les resultaba más intuitiva que el simple uso de listas de componentes. Además, la separación existente entre el diseño de la arquitectura a nivel semántico y la implementación concreta de la misma influyó positivamente en la distribución de funcionalidad que hicieron los usuarios y, por tanto, en el software generado.

En uno de los experimentos realizados (Sección 7.2) se valoró cómo la inferencia automática de componentes software afectaba positivamente a la calidad de las distribuciones creadas. En dicho experimento

se comprobó de manera empírica como este método aumentaba la cohesión de los componentes, disminuía su acoplamiento y reducía la duplicación de funcionalidad pero, lo que verdaderamente demostró que la técnica era útil para el desarrollo, fue que las modificaciones que se necesitaban realizar para añadir una nueva característica al juego eran menores con el uso de la técnica.

Por otro lado, el experimento de la Sección 7.1, demuestra que la metodología propuesta simplifica la comprensión de la arquitectura basada en componentes, una técnica que al principio es rechazada por los programadores. Aunque en un principio esta metodología no fue concebida con fines pedagógicos, el dominio formal añade a la arquitectura información semántica que facilita su aprendizaje y uso. Además, gracias a la generación procedimental de contenido iterativa, los usuarios pudieron hacer refactorizaciones semánticas que les permitían probar diferentes configuraciones sin esfuerzo, ya que los cambios que hacían en las distribuciones se transmitían automáticamente al código.

Sin embargo, no hemos tenido la oportunidad de probar con diseñadores y programadores reales la idoneidad de la metodología en grandes desarrollos. Hubiese sido interesante comprobar cómo sería valorado un proceso de desarrollo de mayor tamaño en el que participasen los diferentes roles, de manera que los diseñadores realizasen un dominio que se estableciese como contrato de mínimos con los programadores, encargados de la implementación.

8.3. Trabajo futuro

Nosotros creemos que la metodología de desarrollo propuesta en esta tesis y *Rosette*, la herramienta que implementa las ideas expuestas, proporcionan una plataforma estable sobre la que seguir investigando y aportando mecanismos que, mediante el uso de representaciones formales, permitan mejorar el desarrollo de videojuegos. Para concluir esta tesis presentamos una serie de ideas o recomendaciones de por donde se podría continuar el trabajo en el futuro.

Empezando por la representación semántica, la propuesta expuesta en este trabajo se ha centrado en la descripción de las entidades de juego. Esas entidades se definen en función de sus atributos (estado) y de los mensajes que éstas interpretan (básicamente qué acciones pueden realizar y eventos a los que responde). Incluso, a nivel de componente, se puede describir qué mensajes se envía la entidad a sí misma, requiriendo funcionalidad de más bajo nivel, para poder llevar acabo acciones complejas.

Sin embargo, aunque para definir un juego es condición indispensable la descripción de sus elementos, no es condición suficiente. Las entidades de un juego suelen interaccionar entre ellas de muy diferentes maneras pero, en su forma más básica, se comunican mediante el paso de mensajes para indicar que acciones realiza una sobre otra o para activar un sensor de la entidad.

Sería interesante poder definir en el dominio formal de juego cómo *interaccionan* unas entidades con otras. Con la representación semántica que se propone en esta tesis se pueden definir cosas como que el *jugador* puede *recibir daño* y por tanto perder *vida* o que un *personaje* puede *aumentar su vida*. Si se introdujese el concepto de interacción en el dominio se podrían involucrar diferentes entidades en el mismo proceso, describiendo, por ejemplo, que un *enemigo* puede *dañar* al *jugador* o que un *botiquín* puede *aumentar la vida* de un *personaje*.

Este aumento de la descripción semántica añadiría más valor aún a la ontología y se podría usar esa información en diferentes fases, sobre todo en el diseño de niveles y comportamientos. Al definir las interacciones en el dominio, durante la edición de niveles se podría razonar sobre las entidades y proporcionar *feedback* a los diseñadores acerca de nuevas posibles inconsistencias. Además, toda esta información acerca de las interacciones sería muy útil cuando se modelan los comportamientos de las entidades, pudiéndose también valorar cómo de útil o usable es un comportamiento concreto en un nivel dado. En relación con el *feedback*, ahora mismo en *Rosette* sólo ofrecemos una descripción textual de las inconsistencias, lo que lo hace difícil de entender por los diseñadores. Deberíamos estudiar métodos más visuales y fáciles de entender, los cuales se podrían acompañar de sugerencias para corregir los errores.

La descripción de las interacciones entre las entidades sería el primer paso lógico a dar, pero no el único. En la ontología se podría seguir añadiendo más conocimiento del que se encuentra en el documento de diseño, de manera que las diferentes mecánicas, reglas de juego, relaciones entre entidades, etc. pudiesen acabar siendo modeladas semánticamente y representado de una manera visual que permitiese mejorar la transmisión de ideas entre los diferentes miembros del grupo.

Incluso, se podría ir más allá definiendo consecuencias a las acciones del dominio y, durante la edición de niveles, describiendo formalmente cuáles son los objetivos que se deben cumplir en el nivel. En ese contexto ontológico, se podría razonar sobre los diferentes niveles de juego para tratar de averiguar si esos estados objetivos son alcanzables con las entidades que se encuentran en el nivel.

Por otro lado, cuanto más se avanzase en la descripción del dominio

mayor sería el contenido que se podría generar de manera procedimental. Con un dominio suficientemente rico los niveles podrían generarse automáticamente, de manera que los diseñadores pudiesen definir *qué* es lo que quieren que tenga su nivel y *qué* objetivos se pretenden cumplir, pero no *cómo* está creado el nivel. Éste tipo de generación procedimental de contenido es muy útil para promover la rejugabilidad ya que, si se introduce un pequeño grado de aleatoriedad en la generación, se consigue que las partidas nunca sean iguales.

En esta tesis se han sentado las bases para continuar y abrir nuevas ramas. Se ha especificado una representación ontológica, una metodología que gira entorno a ella, técnicas específicas que sacan partido del conocimiento semántico y una herramienta que implementa la funcionalidad para poder probar empíricamente la validez del trabajo. Esperamos que en los próximos años se siga avanzando en este sentido y que el uso de dominios formales vaya tomando peso dentro de mundo de desarrollo de videojuegos.

Capítulo 9

Ontological methodology for game development

9.1. Introduction

The increase of the popularity of videogames has provoked an increment in the quality and, for this reason, the number of people involved in the development also increased with the passage of time. In the past, one or two programmers were in charge of the whole development process whilst nowadays, big productions, involves a big quantity of people that take different roles in the development.

Due to this team specialization, now there are different roles in a game development, were there is not just programmers in charge of everything (at least in medium or big developments). Now there is a multidisciplinary team of people that come from very different areas. Apart from the programmers, who implement the game functionality, and the artists, who generate graphic resources, we can stress the designer role. Game designers are those people responsible of the creative aspect of the development so they must conceive and develop the gameplay, the mechanics, the story with its events and dialogues, the characters, how they behave, etc. However there is not expected that they have programming skills (although a good designer is that one that has skills in every area)

Added to the increase in the demand and to the difficulty of managing a multidisciplinary team we have the fact that the design of the game mechanics need a fast development of the functionality to try whether the ideas "work". All these reasons provokes that the classic methodologies are not useful for these kind of developments and now people in the industry use more agile methodologies. These current methodolo-

gies adapt better to changes in the specification, giving priority to the development speed against the over-engineering. The game mechanic development uses to be an iterative process where the ideas that arise are documented, implemented, evaluated and, in function of this evaluation, discarded, accepted or refined, repeating all the previous phases.

Over the years, the technology used to support this kind of methodology has evolved, and today almost all the games on the industry use a component-based architecture for implement the different entities or game objects. This kind of technology is very flexible, allowing all types of changes in the game specification and promoting code reusability. However these architectures, based on composition rather than inheritance, complicate readability and code debugging since the functionality of the game entities is fragmented and divided by several software components. The distribution of component functionality allows that the creation of entities can be directed by external data files, delaying the formation of entities until the game execution. This technology provides speed when specifying new entities or modifying those that were already created. It also saves time during the compilation, but it is difficult to understand at a high level what is each entity. In the component-based architecture, entities are limited to a set of components/skills, losing the ontological meaning that a hierarchy provides and also moves many of the errors that could be detected before at compile-time to run-time, delaying and complicating the identification of problems.

The specialization in roles and the increasing number of the team members has introduced a specific problem in the development: the communication between the different members. Historically, this communication has been made ??by a *game design document* written in natural language by designers. However, if the technology had to be adapted to allow a fast and flexible development, the existing communication must also evolve to optimize the development process. Being a textual document its nature is static so changes and modifications that must be done ??on the initial specification are quite expensive to make (it is hard to describe and update a functionality) and the communication is very slow (natural language is very expressive but is considerably more expensive to read or interpret than a more visual and schematic representation).

The problems that have been highlighted affect to two of the most important roles in game development and cause major delays and bottlenecks in the implementation of agile development techniques. In the flow of the method, as it has been explained above, designers capture knowledge in a design document and programmers replicate this know-

ledge into code, what provokes that the behaviour of the entities is split and hidden between the code and the data files. This complicates code reuse and causes duplication of information, where the easy equivalence between what is documented in the design document and the final result of the deployment is lost.

9.2. Goals

The goal of this thesis is to provide a development methodology that minimizes the ambiguity of specifications, avoid duplication of information, improves the transmission of this information and makes more flexible the game domain description, making it easier to make modifications. In conclusion, a methodology that allows us to speed the development. As we shall see, the proposed methodology is based on a formal domain or a ontology so that the research question that this thesis want to answer is:

> *Could a formal domain helps to alleviate the handicaps of video game creation and accelerates its development?*

Through an ontological description we can represent a lot of semantic information that previously was simply described in a document and implemented and hidden in the code. Programmers had difficulty seeing the consistency of the final distribution of functionality, leading sometimes to code duplication. Through this kind of formal representation of knowledge, designers can easily describe (with some kind of visual editor) the game entities in a quick way. Programmers receive a representation of the domain where description of the game elements is much less ambiguous and leaves less possibilities for interpretation. They use the ontology not only to understand what the designers want but also to take profit of that knowledge (that they completed by concepts more oriented to implementation) during development. The knowledge encapsulated in the ontology can be used with a reasoner to detect inconsistencies hardly traceable in the traditional component-based architecture and also, from that knowledge, generate procedural game content, avoiding programmers redefine information in the code that had already been defined in the ontology. This way the relationship between the elements defined in the ontology and software elements of the component-based architecture is maintained, bringing to light all that knowledge that before was distributed in software components and is now in the ontology.

In order to accomplish this goal, we have analysed which is the methodology used in development teams of the industry to create a game.

The we did a review of different techniques used by other authors to mitigate similar problems that the one we try to give an answer here. Within this research about the state of the art we corroborated that other authors and designers in the industry had identified in one way or another existing problems in the transmission of information through a traditional *game design document*. Some authors have used various semantic representations to define certain aspects of games and how they procedurally generate content or code from these semantic descriptions. Finally we reviewed some industry and academia tools to create knowledge representations more easily and quickly.

Section 9.3 describes the core of this thesis, explaining what is our methodology proposal for game development, which would be the work flow and the interactions between designers and programmers, how we represent the different ontology concepts of the game and how it serves as a contract between ontology designers and programmers. Furthermore, in Sections 9.4 and 9.5 we present some specific and complementary techniques that accelerate the development and improve the software quality finally obtained, all of this thanks to having a formal game description. One of these techniques allows us to automatically identify the best distribution of software components from the functionality specified in the entities. It also shows how this distribution can be iteratively modified by adding, modifying or deleting functionality, while maintaining consistency with previous phases of development. Furthermore, we procedurally generated game content from the ontology. This content serves for programmers, producing the code of the game logic, and also for designers, creating content for use on level, maps and behaviour editors. These two techniques are closely related because, due to the relationship between the elements specified in the ontology and the generated content, they can modify the ontology with the assurance that content generation move those changes in the specification to the implementation, maintaining all previous work done by programmers and designers. Finally we explain how to use semantic knowledge, using a reasoner, to find inconsistencies. These inconsistencies include both those that are in the domain description of the game, at the entity level, and those detected in the different maps and behaviours created by designers. This will allow consistency and detect failures at design level that would otherwise be impossible to detect until the execution of the game. Furthermore, it improves the use of the above techniques as it is possible to detect inconsistencies in the domain changes, distribution of components or changes in the generated game content (i.e. level maps created by designers with specific instances of entities and behaviours).

With the idea of proving our work, we have developed a prototype called *Rosette* that is a graphical authoring tool that supports the methodology and techniques previously discussed. In Section 9.6 tells how it has been developed and how these characteristics have been implemented. *Rosette* has been used by students in the Master of Video Games of the Complutense University of Madrid[1] and, as a result, we were able to make two experiments that test the validity of our methodology. Both experiments are explained in Sections 9.7 and 9.8. We first use our methodology and, more specifically, *Rosette* to help to understand the component-based architecture. We tested whether the methodology and *Rosette* serve as a learning tool, where the semantic abstraction and the separation between specification and implementation allow to better understand the concepts. In addition, procedural content generation allows to make two small games in an acceptable time, allowing students to focus on a higher level of understanding, preventing details related to the code. In the second experiment, which was made to validate the component suggestion technique, we tested whether the suggestions offered by this technique accelerate the development and whether the distributions of functionality in software components have better quality than those generated by an expert. Analysing the results we concluded that the component suggestion technique improves the quality of the final distribution and ultimately increases the speed of the development.

Finally, in Section 9.9, we expose the conclusions of this thesis and also of the results.

9.3. Development methodology based in ontologies

Game development is nearly an art, where new ideas must be completely implemented and tested to be sure that they are fun to play. Trial and error of new game concepts is needed, what requires an elaborated software architecture that provides the needed flexibility and change tolerance. In addition, game development requires the collaboration of artists that picture the world, programmers that make the world work, and designers that create a game out of the visuals and dynamics of the virtual world. Collaboration between programmers and designers is specially important and, at the same time, specially difficult. Designers decide what need to be done and programmers decide what can be done. A fluent and constant communication between designers and

[1] `http://www.videojuegos-ucm.es/`

programmers is needed, where both know the needs and objectives of the others.

The part of the game that suffers the changes in the specification and needs flexibility is usually called the game logic. Virtually all implementation alternatives for the game logic architecture manage the state of the game through a set of *entities*, where these entities are self-contained pieces of logic (Bilas, 2002) that can perform different tasks such as render themselves, find and follow paths or make decisions. Common entity examples are enemies, playersávatars, interactive items or even props such as doors, trees and fences. Not so obvious entities are *pure logic* elements like camera sequences, waypoint markers, or *triggers* that control plot points in the story.

Practically every modern videogame implement and manage this entities by using a component-based software architecture for games (West, 2006). This specific architecture for games decompose entities in pieces of functionality in such a way that, instead of having entities implemented in a particular class which defines its exact behaviour, now every skill or functionality that the entity has is implemented by a component; so entities are just component containers.

In order to facilitate the collaboration between programmers and designers in game development, we propose a computer-aided methodology that involves both designers and programmers, and is based on two main ideas:

1. put a declarative model of the game domain as the contract between programmers and designers, and let designers formalize the elements of the game that programmers must develop, and

2. use a component-based software architecture, instantiated as an elaboration of the game domain that connects the declarative model with actual code.

9.3.1. Methodology

Game designers communicates the mechanics and other features of the game to programmers by the *game design document*. This design document is drawn up using natural language, what makes it an slow and potentially ambiguous method for communication. For that reason we propose the use of ontologies in full game developments to accelerate the process.

The presented methodology is a concrete example of the use of ontologies in software engineering (Seedorf et al., 2006), where the onto-

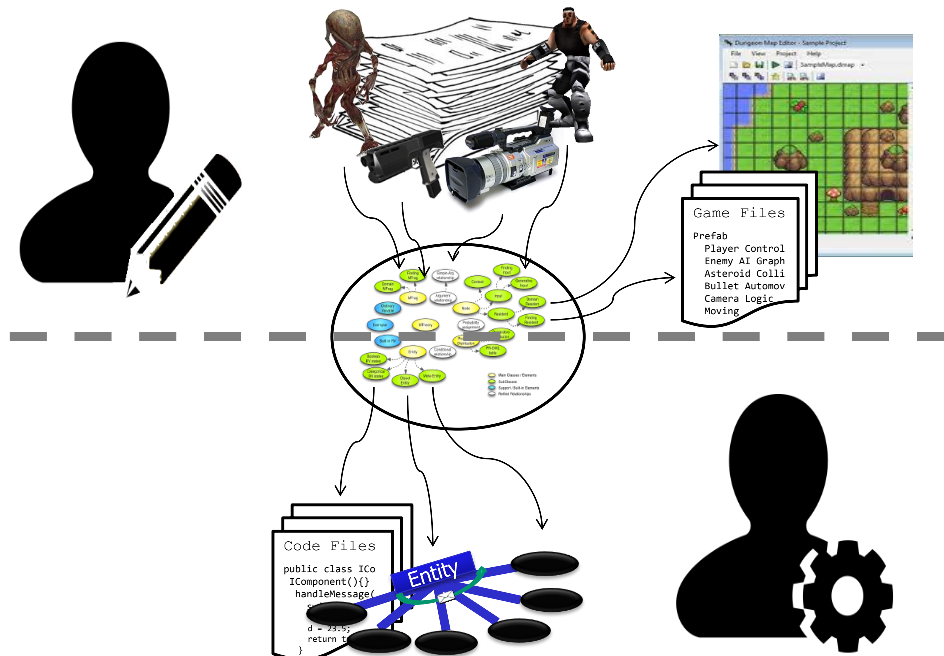

Figura 9.1: development methodology based in a formal domain

logy that drives the development can be adapted and modified during the process. This methodology proposes a development where:

1. The base of the communication is speeded up for both the emitter and the receiver of the message.

2. It is easier to introduce modifications in the game specification.

3. The ambiguity of the communication is reduced.

4. All the information used for the communication is reused in further steps of the development.

Figure 9.1 shows a conceptual view of programmers and designers responsibilities. The ontology is the link between the two roles and provides enough benefits to convince designers and programmers to maintain it up to date. Prior to enumerate the different phases of the methodology, please note that our it supports these phases being reiterated as needed. We will focus in just one iteration and, for simplicity, we will assume that the game is developed using a waterfall model. Therefore this methodology promotes agile developments where it phases are:

1. *Formal domain modelling:* Designers and programmers collaborate in the game domain description.

a) *Ontological description from a design point view:* In addition to the *game design document*, designers define or modify the entities of the game and their features using a visual authoring tool that simplifies the domain edition a visualization.

b) *Model acceptance:* Programmers analyse, reach an agreement with designers and finally validate the new game domain, which can be seen as a contract between designers and programmers.

c) *Model extension from an implementation point of view:* Programmers enrich the initial domain with technical details. Furthermore they have to identify the software components that encapsulate every action and attribute in self-contained pieces of functionality.

2. *Content generation:* Once the domain of a concrete iteration has been modelled, the semantic information of the ontology is used to create game content. Programmers and designers must then complete the content of the game.

   a) *Functionality in the code:* Programmers implement the described functionality adapted to the target platform and programming language. They only have to fill some empty methods out because the rest of the code is autogenerated.

   b) *Level design:* Designers are in charge of creating the different levels of the game, starting from the resources generated from the ontology. They should also create the behaviour of the game entities.

In the different steps of the methodology we have refereed to two different roles: programmer and designer. However, in big developments, we can be more specific detailing the people of the development team. The kind of designer that define the entity ontology, through the visual tool, must be a designer with some technical knowledge. This *technical designer* is the one that will negotiate with programmers the game requirements in terms of functionality, so he should know what is the *programmer's way of doing things*. In the same way, the kind of programmer that completes the game ontology should have a global view of the game development. This *lead programmer* of the logic module is the one that designs the software architecture and communicates with the *technical designer* to reach an agreement. Finally, there are *regular programmers* implementing the functionality in a concrete program-

ming language and the *level designers* creating maps and behaviours for the game.

So, game creation begins with the *technical designer* defining the complete game functionality. Gameplay is provided by *entities* (players, enemies, items, and so on) and the *technical designer*, starting from the *game design document*, must infer all the entity types the game will have and specify them in an ontological (hierarchical) way.

Entities have a state (position, health, armor, ...) and a set of functionalities (actions they can do and senses they perceive) that must be also specified. The *technical designer* provides all this information creating the complete *game domain* that constitutes a semantic specification of the gameplay and becomes a *contract* between designers and programmers. If any decision endangers the implementation of the game, the *lead programmer* of the logic module must report it to the *technical designer* before accept this game domain. After all, programmers will be in charge of implementing the functionality so they know the technical limitations of the game technology.

Once they come to an agreement, the *lead programmer* is in charge of breathing life into all those entities designers have envisioned. He will start using the semantic specification of the game domain, *adding* new entities, attributes and messages (actions) that, although go unnoticed from a designer point of view, will be important from the implementation perspective. This may even involve creating new entities, messages or attributes that designers do not care about, such as cameras, waypoints, triggers and other technical elements, or populate the hierarchies with internal elements that will make the development phase easier.

Entities are then split into software components that will contain subsets of the state and functionality that the starting entities had. Due to the semantic information of the game is ontologically specified into a formal domain, we can apply some mechanisms that automatically infer the best component distribution for a concrete entities description. So the component distribution can be hand-made, or created using these automatic mechanisms and then fine-tuned as desired.

It is noteworthy that the semantic model, defined in the ontology by designers and programmers, is not platform, game engine or even programming language dependant. The ontology is a game representation from a high point of view that bring to light a lot of information that is usually hidden between the code and external files. Due to a formal domain is not tie to an specific technology, the game that it describes can be implemented with any engine and even the same model could be used to implement the same game in two different platforms (i.e. PC and mo-

bile). The only difference would be the underlying technology used for the implementation and, actually, the semantic knowledge could help to create this game code and content.

Once programmers have completely modelled entities and components, the semantic model of the game domain is used to create game content and source code scaffolding for all of them, adapted to the target platform and underlying technologies. In some way, the used methodology is an Ontology Driven Architecture (Tetlow et al., 2006).

The programmers are in charge op three steps in the content generation:

- Procedural code generation.
- Implementation of the functionality.
- Procedural generation of the files with entity descriptions.

In the first step, the knowledge of the ontology is turned into code for the chosen programming language and platform, creating the game logic layer. This includes the entity, component and message management, where there is some placeholders that mark the specific points where programmers must write code to implement the components'behaviour. Therefore, they do not have to write all the game logic and component management layer, but just the code that implements the concrete functionality of the components. All the code that the target platform (say Unity3D or a in-house game engine) requires to manage the component (usually constructors, attribute declarations, and so on) are machine generated from the game domain.

Similarly, the semantic model is also used by the programmers to create files that feed the tools that game levels designers use to create maps for the game. Prefabs (Unity3D) or blueprints/archetypes information (C++ engines) are generated and become the concrete entity definitions that can be put into the game levels. They can also be used for the behaviour definition where structures such as finite state machines or behaviour trees are specified.

The work of the level designers consists on level generation where they must:

- Create the environment by using static models like walls, corridors, trees, etc.
- Add interactivity by means of entities (those defined in the ontology).

- Define character behaviours and add them to the entities.

During the whole development the knowledge-rich information of the game domain allows us to detect inconsistencies, such as when an entity is tried to make an action that it is not supposed to do according to the game domain (Sánchez-Ruiz et al., 2009a). In this way, the game semantic model constitutes a central point of the process and a *verifiable game specification*.

9.3.2. Using Ontologies for Modelling Entities

Our purpose is to provide development teams with tools that allow them to model their game in such a way that reflects the underlying component model. At the same time, we intend to solve the issues of a component model documented through blueprint files. Our model formalizes the collection of game entities in the OWL web ontology language[2]. Describing the key attributes of game entities using a language like OWL will bring to light a lot of data that is hidden in the code and knowledge that is lost in the translation to blueprints . Using easy-to-use authoring tools, designers and programmers may define the system that is later stored as OWL definition. This authoring tools may automatically adapt themselves to the changes in the game domain and even act as code generator of component and messages templates in a way that resembles the Model-Driven Architecture (Pastor y Molina, 2007). With such a strong domain model, even AI programmers may benefit using the information to increase the capabilities of the AI.

Moreover, having a model of the game entities in OWL allows to use the different tools that are able to reason over OWL representations, checking whether the defined game domain is consistent. When integrated with the authoring tools these sanity checks, executed over the OWL domain, let us provide with feedback to the users about what situations should be fixed, thus replacing compile time checking missed in a component-based architecture.

In order to create a formal specification of the entity domain, programmers must define different aspects of it. In this formal domain, these aspects are set in several ontologies, that through the is-a relation, define hierarchies of entities, components, attributes and messages. The root and leave concepts in these ontologies will typically correspond to actual classes in the code. Additionally, inner concepts can be introduced, between the root and the leaves, to structure the model at different

[2] http://www.w3.org/TR/owl2-overview/

abstraction levels:

- **Entity ontology:** Entities must be composed of components and for a fine-grained approach, they have to be populated with attributes. These attributes must be able to be filled in with default values where default values of parent entities could be overwritten in child classes. Maintaining entities in a conceptual ontology helps programmers to rapidly recognise the main purpose of them. Nevertheless this will be only a conceptual ontology and it does not have to be mistaken with a class hierarchy; every game entity will be generic and the difference between entity types is determined by the components they have.

- **Component ontology:** Every component in the ontology must be described with the messages that is able to carry out. Sometimes there are dependencies between the goal a component has (messages that it carries out) and the subgoals (other messages) that must be carried out by other components to reach the goal. In our OWL domain this turns into restrictions to carry out messages. Furthermore, a component may need that the entity, which it belongs to, has some attributes. This also have to be annotated in the component description. In this case, the ontology will typically match with the C++ representation since a component behaviour can specialise to another one.

- **Message ontology:** A message just represents a requirement of functionality. They are sent to entities and caught by components. As a refined definition, they must be described with attributes that parametrize the required functionality. As in entities, this ontology will be mostly conceptual and only the leaves of the ontology should be implemented.

- **Attribute ontology:** These attributes are the ones used in the previous ontologies. They must be described by a key (just a name) and the type of the values they can take (integer, string, etc.). In addition, these attributes could be described with stronger restrictions such as a set of unique possible values or intervals. There are also complex attributes like entity, component or message types. However, both these restrictions and the ontology will have nothing to do with the final implementation but they are useful for sanity checking.

Entity, component, message and complex attribute ontologies are represented as *class* hierarchies in OWL, whilst the rest of the attribute

ontology is represented as an OWL *data property* hierarchy.

In order to define relationships between the elements of the different hierarchies we use next *object properties*:

- **isA(?e, ?p)**: Entities are related in a conceptual hierarchy. *isA* defines that the entity *?e* is child of *?p* in the conceptual hierarchy. We do not use subclasses because child classes not necessarily inherits their parent properties.
- **hasComponent(?e, ?c)**: Indicates that *?c* is a component that belongs to the entity *?e*.
- **isComponentOf(?c, ?e)**: Is the inverse property of *hasComponent*.
- **interpretMessage(?x, ?m)**: Indicates that the entity or component *?x* can carry out the message *?m*.
- **isMessageInterpretedBy(?m, ?x)**: Is the inverse property of *interpretMessage*.
- **hasAttribute(?x, ?a)**: Indicates that the entity or component *?x* is described by the attribute*?a*.
- **isAttributeOf(?a, ?x)**: Is the inverse property of *hasAttribute*.
- **isInstantiable(?x)**: Indicates that *?x* is instantiable and will be represented in the final game.
- **isNotInstantiable(?x)**: Is the inverse property of *isInstantiable* and describes those concepts that only have sense from the conceptual point of view.

Having these properties into account, we an describe the elements of the game. An entity is described as:

```
Class: <EntityName>
  SubclassOf: Entity
    [and isA <ParentEntityName>]+
    [and (hasComponent some <ComponentName>)]∗
    [and (interpretMessage some <MessageName>)]∗
    [and (hasAttribute some <ComplexAttributeName>)]∗
    [and (<SimpleAttributeName> some <RDFDatatype>)]∗
    and (isInstantiable) | (isNotInstantiable)
```

```
(+) implies that that there must be one or more properties
like this
(*) implies that that there may be zero or more properties
like this
```

On the other hand, components are defined as:

```
Class: <ComponentName>
  SubclassOf: <ParentComponentName>
    [and (interpretMessage some <MessageName>)]*
    [and (isComponentOf only
          (hasComponent some
            (interpretMessage some <MessageName2>)))]*
    [and (hasAttribute some <ComplexAttributeName>)]*
    [and (<SimpleAttributeName> some <RDFDatatype>)]*
    and (isInstantiable) | (isNotInstantiable)
```

Messages like:

```
Class: <MessageName>
  SubclassOf: <ParentMessageName>
    [and (hasAttribute some <ComplexAttributeName>)]*
    [and (<SimpleAttributeName> some <RDFDatatype>)]*
    and (isInstantiable) | (isNotInstantiable)
```

Complex attributes:

```
Class: <ComplexAttributeName>
  SubclassOf: <ParentComplexAttributeName>
  EquivalentTo: <EntityComponentOrMessageName>
```

Lastly, attribute with simple types can be described without restriction:

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype
```

With range restrictions

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype[>= 0f, <1f]
```

Or with restrictions that determine a set of values:

```
DataProperty: <SimpleAttributeName>
  SubPropertyOf: <ParentSimpleAttributeName>
  Range: RDFDatatype and {<value,>∗}
```

Finally we create prototypical instances of the entities to store default values. We represent them as:

```
Individual: <iEntityName>
  Types: <EntityName>
  Facts: [isA <iEntityParentName>,]+
         [hasComponent <iComponentName>,]∗
         [interpretMessage <iMessageName>,]∗
         [hasAttribute <iComplexAttributeName>,]∗
         [<SimpleAttributeName> <AttrivuteValue>,]∗
```

And obviously there must be also individuals of components, messages and complex attributes.

9.4. Automatic identification of software components

Even though the use of components promote flexibility, reusability and extensibility, it does not come without costs. One of the main issues is the lack support in the mainstreams languages (mainly C++) that forces programmers to create a complex source code infrastructure that supports components and the ability of composing entities with them. This also leads to the lack of compiler support in the detection of different problems related to the lost of datatype information that a traditional class hierarchy would have exposed immediately to the programmer.

In this section, we present a new method for easing the design of a component-based architecture. Using a visual tool designers and programmers first graphically defines the entity hierarchy (data and actions), a well-known task for game programmers. Using Formal Concept Analysis (FCA) (Ganter y Wille, 1997), we automatically suggest a *component* distribution that will fit the requirements of the provided classical hierarchy (Section 9.4.2).

The candidate distribution is shown to the user through a visual interface, where she has the opportunity to modify it (Section 9.4.3).

As said before, videogames specification will surely change in the long run, and the original hierarchy (and component distribution) will need several revisions. As a result of that *agile methodologies*, such as *scrum* (Schwaber y Beedle, 2001) or *extreme programming* (Beck, 1999; Beck y Andres, 2004), are taking a big importance in the videogame developments. Agile methodologies give repeated opportunities to assess the direction of a videogame throughout the entire development lifecycle, which is split in iterations or *sprints*. The principal priority is having functioning pieces of software at the end of every sprint to re-evaluate the project goals. To this end, the technique allows an *iterative* hierarchy definition during these sprints, remembering all the changes applied to the previously proposed component distributions, and redoing them into the last one (Section 9.4.4). This process is possible using the lattices (Birkhoff, 1973) created using FCA and represented with OWL.

9.4.1. Formal Concept Analysis

Formal Concept Analysis (FCA) is a mathematical method for data analysis, knowledge representation and information management. It was proposed by Rudolf Wille (Wille, 1982) and during the last decades it has been used in numerous applications in different domains, like Psychology, Medicine, Linguistics and Computer Science among others.

FCA is applied to any collection of items (or *formal objects* according to FCA nomenclature) described by means of the set of properties (or *formal attributes*) they have. When FCA is applied over a set of objects, data is structured and grouped into formal abstractions called *formal concepts*. These formal concepts can be seen as a set of objects that share a common set of attributes. Therefore, the result of the application of FCA over a collection of items provides an internal view of the conceptual structure and allows finding patterns, regularities and exceptions among them. Moreover, formal concepts may be sorted using the subconcept-superconcept relationship, with the set of objects of subconcepts being a subset of the ones of superconcepts, and reciprocally, the set of attributes of subconcepts being a superset of the ones of superconcepts.

The items and attributes are given to the FCA in form of a *formal context*. A *formal context* is defined as a triple $\langle G, M, I \rangle$ where G is the set of objects, M the set of attributes, and $I \subseteq G$xM is a binary (incidence) relation expressing which attributes describe each object (or which objects are described using an attribute), i.e., $(g, m) \in I$ if the object g carries the attribute m, or m is a descriptor of the object g. When the sets are finite, the context can be specified by means of a cross-table,

where rows represent objects and columns attributes. A given cell is marked when the object of that row has the attribute of the column.

We can define a concept by enumerating the set of objects that belong to that concept (its *extent*) or listing the attributes shared by all those objects (its *intent*). Formally, we define the *prime* operator that when applied to a set of objects $A \subseteq G$ returns the attributes that are common to all of them:

$$A' = \{m \in M \mid (\forall g \in A)(g, m) \in I\}$$

and when applied over a set of attributes, $B \subseteq M$, its result is the set of objects that have those attributes:

$$B' = \{g \in G \mid (\forall m \in B)(g, m) \in I\}$$

With this definition, we may now define the *formal concept*: a pair (A, B) where $A \subseteq G$ and $B \subseteq M$, is said to be a *formal concept* of the context $\langle G, M, I\rangle$ if $A' = B$ and $B' = A$. A and B are called the *extent* and the *intent* of the formal concept, respectively.

The set of all the formal concepts of a context $\langle G, M, I\rangle$ is denoted by $\beta(G, M, I)$ and it is the output of the FCA. The most important structure on $\beta(G, M, I)$ is given by the subconcept - superconcept order relation denoted by $\leq$ and defined as follows $(A1, B1) \leq (A2, B2)$ where $A1$, $A2$, $B1$ and $B2$ are formal concepts and $A1 \subseteq A2$ (which is equivalent to $B2 \subseteq B1$ see Ganter y Wille (1997)). This relationship takes the form of a *concept lattice*. Nodes of the lattice represent formal concepts and the lines linking them symbolize their subconcept-superconcept relationship.

A concept lattice may be drawn for further analysis by humans (Figure 9.3). This representation uses the so called *reduced intents* and *reduced extents*. The reduced extent of a formal concept is the set of objects that belong to the extent and do not belong to any subconcept. On the other hand, the reduced intent comprises attributes of the intent that do not belong to any superconcept. In that sense to retrieve the extension of a formal concept one needs to trace all paths which lead down from the node to the bottom concept to collect the formal objects of every formal concept in the path. By contrast, tracing all concepts up to the top concept and collecting their reduced intension gives its intension.

9.4.2. Generating components through Formal Concept Analysis

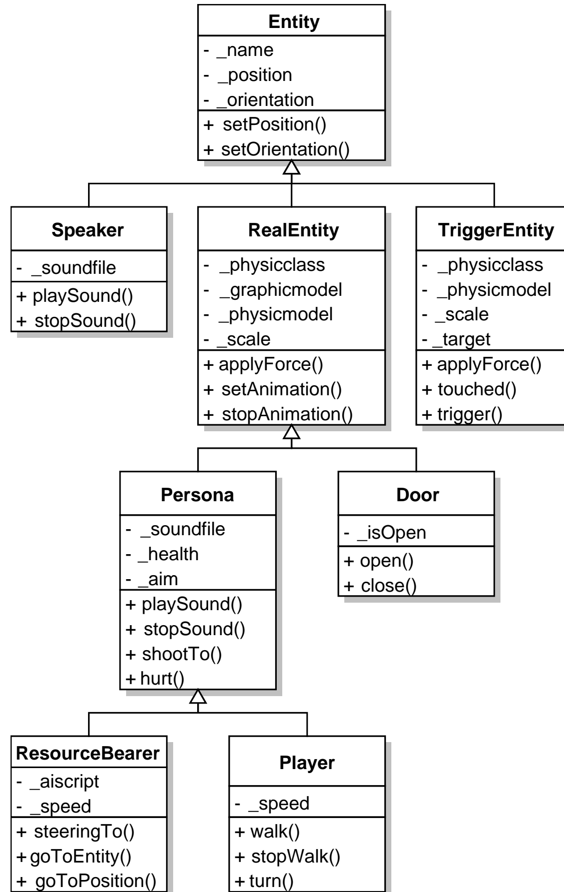

Figura 9.2: Ontological description of game entities

The first step is to graphically specify the entity hierarchy of the game. Users must provide both the entities and their features (attributes and methods). For example, they will indicate that a *Player* can *walk* and *turn* whilst a *ResourceBearer* must be able to *goToPosition* or store an *_aiScript*.

Once done, we transform the hierarchy into a *formal context* in order to apply FCA. Entity types (*Player*, *ResourceBearer*) become *formal objects* and features (*walk*, *goToPosition*, *_aiScript*) become *formal attributes*. Therefore, our formal context $\langle G, M, I \rangle$ is built in such a way that G contains every entity type and M will have every functionality and attribute. Finally, I is the binary (incidence) relation $I \subseteq G$xM, expressing which attributes of M describe each object of G or which objects are

	setPosition	setAnimation	touched	_physicclass	_aiscript	...
Entity	■					...
Trigger	■		■	■		...
Persona	■	■		■		...
ResourceBearer	■	■		■	■	...

Tabla 9.1: Partial formal context of the game domain

described using an attribute. This is filled in by going through the entity hierarchy annotating the relations between entity types and their features and considering inheritance (subclasses will include all the formal attributes of their parent classes). Table 9.1 shows a partial view of the formal context extracted from the hierarchy shown in Figure 9.2.

The application of FCA over such a formal context is done by using the Galicia project (Valtchev et al., 2003), a project usually used by end-users through their visual interface but that can be used by other applications using its API. The application of FCA gives us a set of formal concepts and their relationships, $\beta(G, M, I)$. Every formal concept represents all the entity types that form its extent and will need every functionality and attributes in its intent. Starting from the lattice (Figure 9.3) and with the goal of extrapolating the formal concepts to a programming abstraction, a naïve approach is to generate a hierarchy of classes with multiple inheritance. Unfortunately, the result is a class hierarchy that makes an extensive use of multiple inheritance, which is often considered as undesirable due to its complexity.

Therefore, though this approach of converting formal concepts into classes has been successfully used by others (Hesse y Tilley, 2005; Godin y Valtchev, 2005), it is not valid in our context. After all, our final goal is to *remove* inheritance and to identify *common entity features* in order to create an independent class (a component) for each one. Once done, we will create just a single and generic *Entity* class that will keep a list of components, as promotes the component-based architecture. The addition of new features is done by adding new components to the entity, and these components are independent among themselves. The consequence of this independence is that, now, sharing the implementation of the same feature in different entities is not hard-wired in the code, but dynamically chosen in execution, avoiding the common problems in the

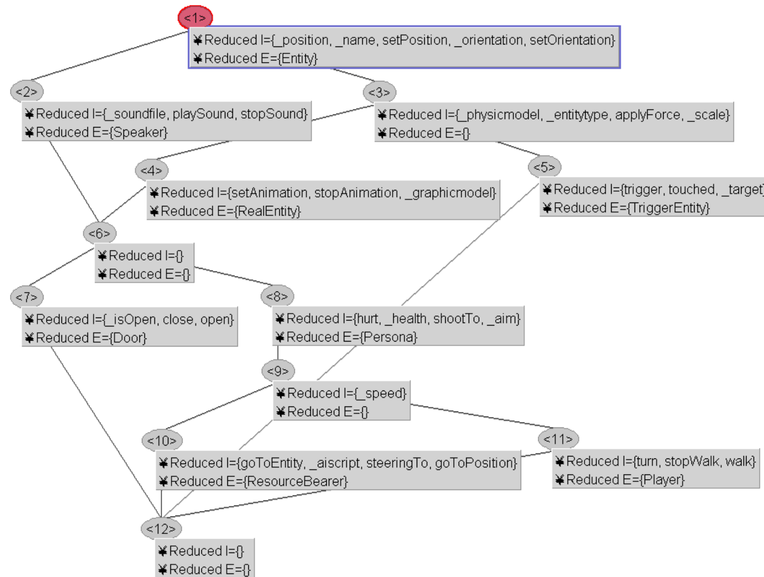

Figura 9.3: Concept lattice

long run of the rigid class hierarchies.

Using FCA, we reach this goal focusing not on the objects (reduced extents of the formal concepts), but *in the attributes* (reduced intents). The idea is based on the fact that when a formal concept has a non-empty *reduced intent* this means that the concept contributes with some attributes and/or functionality that did not have appeared so far (when traversing the lattice top-bottom). The immediate result is that the reduced extent of objects differs from the objects in the superconcepts in those properties and it should be consider to build a component with them. At the same time, we will know that all the instances of the entities in the reduced extent of the formal concept will include the new created component.

For example, when analysing the formal concept labelled *11* in Figure 9.3, our technique will extract a new component containing the features *turn*, *stopWalk* and *walk*, and will annotate that all entities (generic instances) of the concept *Player* will need to include one of those components (and any other components extracted from the formal concepts over it).

The general process performed with the hierarchy H created by the user is:

```
G = entity_types(H)
M = attributes_and_messages(H)
I = buildRelation(H)
L = β(G,M,I)
P = empty component list
for each formal concept C in L
  if C == ⊤ then continue
  B_r = reducedIntent(C)
  if B_r is empty then continue
  add(P, component( B_r ) )
end for
```

All the lines are self-explicative except that with the *add*. The *component* function receives the reduced intent of the formal concept and builds the component representation that has its attributes and functionalities.

In some cases, the top concept ($\top$) has a non-empty intent, so it would also generate a component with all its features (*name*, *position* and *orientation* in our example of Figure 9.3). That component would be added in *all entities* so, instead of keeping ourselves in a pure component-based architecture with an empty generic *Entity* class, we can move all those top features to it. Figure 9.4 shows the components extracted using the lattice from Figure 9.3. The components have been automatically named concatenating each attribute name of the component or, when no one is available, by concatenating all the message names that the component is able to carry out. For example, let us say that the original name of the *FightComp* component was *C_health_aim*.

Summarizing all the process, when analysing a concept lattice, every formal concept that provides a new feature (having no empty reduced intent) does not represent a new entity type but a new component. The only exception is the formal concept in the top of the lattice that represents the generic *entity* class, which has data and functionality shared by all the entity types. Both the generic entity and every new component have the ability of carrying out actions in the reduced intent of the formal concept and they are populated with corresponding attributes.

This way, we have easily obtained the candidate generic entity class and components, but we still have to describe the entity types. Starting from every concept which their reduced extents contain an entity type, the technique uses the superconcept relation and goes up until reaching the concept in the top of the lattice. For example, the *Persona* entity type (Figure 9.3) would have components represented by formal concepts

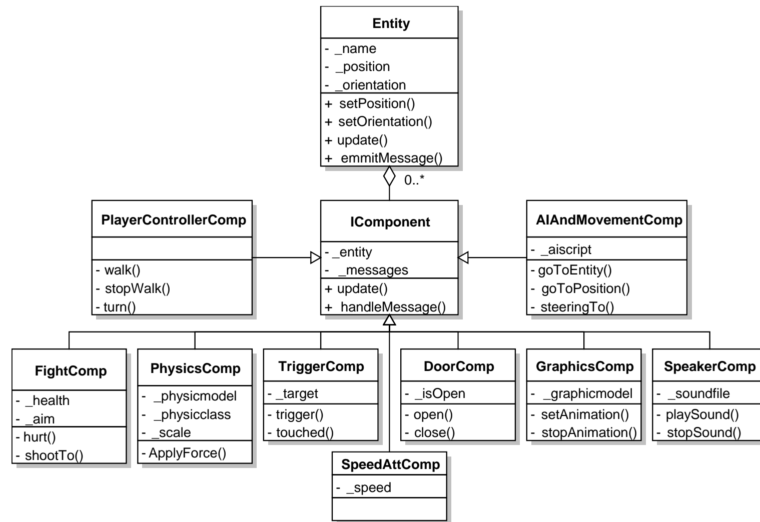

Figura 9.4: The candidate components proposed

number *8*, *4*, *3* and *2* (the number *6* has an empty *reduced intent* so it does not represent a component) whilst the *ResourceBearer* entity type would have the same components but also the number *10* and *9*. Obviously, components of every entity type are stored in the generic entity container represented by the formal concept number *1*.

Keep in mind that the final component distribution does not include information about what components are needed for each entity. This knowledge is not thrown away: we stores all the information in the original lattice using OWL, which provides a knowledge-rich representation that will let it provide some extra functionalities described in the next sections.

9.4.3. Expert Tuning

The automatic process detailed above ends up with a collection of proposed components with a generated name, and the *Entity* base class that may have some common functionality. This result is presented to developers, who will be able to modify it using their prior experience. Some of the changes will affect to the underlying formal lattice (that is *never* shown to the users) in such a way that the relationship between it and the initial formal context extracted from the class hierarchy will be broken. At this stage of the process this does not represent an issue, because we will not use FCA anymore over it. On the other hand, changes could be so dramatic that the lattice could even become an invalid one. Fortunately, we use OWL as the underlying representation, that can

be used to represent richer structures than mere partially ordered sets. In any case, for simplicity, in the rest of the section we will keep talking about lattices although internally our tool will not be using them directly.

Users will be able to perform the next four operators over the proposed component distribution:

1. **Rename**: proposed components are automatically named according to their attribute names. The first operator users may perform is to rename them in order to clarify its purpose.

2. **Split**: in some cases, two functionalities not related to each other may end up in the same component due to the entity type definitions (FCA will group two functionalities when both of them appears together in every entity type created in the formal hierarchy). In that case, we gives developers the chance of splitting them in two different components. The expert will then decide which features remain in the original component and which ones are moved to the new one (which is manually named). Formally speaking, this operator would modify the underlying concept lattice creating two concepts $(A1, B1)$ and $(A2, B2)$ that will have the same subconcepts and superconcepts than the original formal concept (A, B) where $A \equiv A1 \equiv A2$ and $B \equiv B1 \cup B2$. The original concept is removed. Although this is not correct mathematically speaking, since with this operation we do not have concepts anymore, we still use the term in this and in the other operators for simplicity.

3. **Move features**: this is the opposite operator. Sometimes some features lie in different components but the expert considers that they must belong to the same component. In this context, features of one component (some elements of the reduced intent) can be transferred to a different component. In the lattice, this means that some attributes are moved from a node to another one. When this movement goes up-down (for example from node *9* to node *10*), we will detect the possible inconsistency (entities extracted from node *11* would end with missed features) and warns the user to *clone* the feature also in the component generated from node *11*.

   If the developer moves all the features of a component the result is an useless and empty component that is therefore removed from the system. The application of this operator also modifies the lattice. Being F the features to be moved and (A, B) a formal concept where $F \subset B$, every direct subconcept $(A2, B2)$ of (A, B) is

turned into $(A2, B2 \cup F)$ and the (A, B) concept is transformed in $(A, B - (B \cap F))$.

4. **Add features**: some times features must be copied from one component to another one when FCA detects relationships that will not be valid in the long run. In our example, the dependency between node *3* and *4* indicates that all entities with a graphic model (*4, GraphicsComp*) will have physics (*3, PhysicsComp*), something valid in the initial hierarchy but that is likely to change afterwards. With the initial distribution, all graphical entities will have an *_scale* thanks to the physic component, but experts could envision that this should be a native feature of the *GraphicsComp* too. This operator let them to add those "missing" features to any component to avoid dependencies with other ones.

The expert interaction is totally necessary, first of all because she has to name the components but also because the system ignores some semantic knowledge and information based in the developer experience. However, the bigger the example is, with more entity types, the more alike is the proposed and the final set of components, just because the system has more knowledge to distribute responsibilities.

While using operators, coherence is granted because of the knowledge-rich OWL representation that contains semantic information about entities, components, and features (attributes and actions). This knowledge is useful while users tune the component distribution, but also to check errors in the domain and in future steps of the game development (as creating AIs that reason over the domain).

9.4.3.1. Example

Figure 9.4 showed the resultant candidate of components proposed for the hierarchy of Figure 9.11, that can now be manipulated by the expert to tune some aspects. The first performed changes are component rename (*rename* operator) that is, in fact, applied in the figure.

A hand-made component distribution of the original hierarchy would have ended with that one shown in Figure 9.2, that is quite similar to the distribution provided by the technique. When using a richer hierarchy, both distributions are even more similar.

With the purpose of demonstrating how the expert would use the available operators to transform the proposed set of components, we apply some modifications to the automatically proposed distribution in order to turn it into the other one.

First of all, we can consider the *SpeedAttComp* that has the *_speed* attribute but no functionalities. In designing terms this is acceptable, but rarely has sense from the implementation point of view. *Speed* is used separately by *PlayerControllerComp* and *AIAndMovementComp* to adjust the movement, so we will apply the *move features* operator moving (and cloning) the *_speed* feature to both components, and removing *SpeedAttComp* completely. This operator is coherent with the lattice (Figure 9.3): we are moving the intent of the node labelled *9* to both subconcepts (*10* and *11*).

After that, another application of the *move features* operator results in the movement of the *touched* message interpretation from the *TriggerComp* to the *PhysicsComp*. This is done for technical reasons in order to maintain all physic information in the same component.

Then, the *split* operator, which split components, is applied over the *AIAndMovementComp* component twice. Due to the lack of entity types in the example, some features resides in the same component though in the real implementation are divided. In the first application of the *split* operator, the *goToEntity* and the *goToPosition* message interpretations are moved to a new component, which is named *GoToComp*. The second application results in the new *SteeringToComp* component with the *steeringTo* message interpretation and the *_speed* attribute. The original component is renamed as *AIComp* by the *rename* operator and keeps the *_aiscript* attribute.

Finally, although the *Entity* class has received some generic features (from the top concept, $\top$), they are especially important in other components. Instead of just use those features from the entity, programmers would prefer to maintain them also in those other components. For this reason, we have to apply the *add features* operator over the *GraphicsComp*, *PhysicsComp* and *SpeakerComp* components in order to add the *setPosition* and the *setOrientation* functionalities to them.

9.4.4. Iterative Software Development with FCA

In the previous section we have presented a semi-automatic technique for moving from class hierarchies to components. The target purpose is helping programmers facing up to this kind of distributed system, which is widely used in computer game developments. Through the use of FCA, this technique splits entity behaviours in candidate components but also provides experts with mechanisms for modifying these component candidates. These mechanisms are the operators defined in Section 9.4.3, which execution in the domain alter somehow the underlying

formal lattice generated during the FCA process.

Attentive readers will have realized that the previous technique is valid for the first step of the development but not for further development steps. Due to computer game requirements change throughout the game development, the entity distribution is always changing. When the experts face up to this situation, they may decide to change the entity hierarchy in order to use the technique for generating a new set of components. The application of FCA results in a new lattice that probably does not change a lot from the previous one. However, the experts usually would have performed some modifications in the proposed component distribution using our operators. As the process is now repeated, these changes would be lost every time the expert request a new candidate set of components.

Our intention in this section is to extend the previous technique in order to allow an iterative software design. In this new approach, the modifications applied over one lattice can be extrapolated to other lattices in future iterations. Keep in mind that the domain operators (Section 9.4.3) are applied over components that has been created from a formal concept. So, these operators could be applied on similar formal concepts, of another domain, in case that both domains share the part of the lattice affected by the operators.

From a high-level point of view, in order to preserve changes applied over the previous component suggestions, the system compares the new formal lattice, obtained through FCA, with the previous one. The methodology identifies the part of the lattice that does not significantly change between the two FCA applications. This way the tuning operators executed in concepts of this part of the lattice could be reapplied in the new lattice.

The identification of the target part of the lattice is a semi-automatic process, where formal concepts are related in pairs. It automatically identifies the *constant* part of the lattice, which for our purpose is the set of pairs of formal concepts that have the same reduced intent. We do not care about the extent in our approach since the component suggestion lays its foundations in the reduced intent. The components extrated from the formal concepts that have not been matched up are presented to the expert. Then she can provide matches between old components and new ones to the considered *constant* part of the lattice.

It is worth mentioning that some of the operators could not be executed in the new domains due to component distribution may vary a lot after various domain iterations but it is just because these operators become obsoleted.

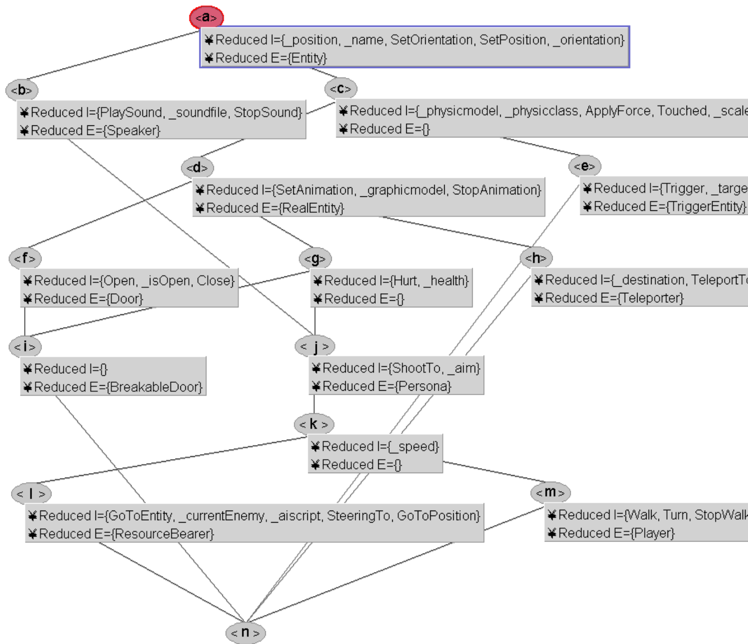

Figura 9.5: New concept lattice

9.4.4.1. Example

In Section 9.4.3.1 FCA is applied to a hierarchy and the automatic part of the proposed methodology leads us to the set of components in Figure 9.4. The resultant domain was modified by the expert, by using the tuning operators, and the component-based system developed ends up with the components in Figure 9.2.

Now, let us recover the example and suppose that the game design has new requirements. The game designers propose the addition of two new entity types: the *BreakableDoor*, which is a door that can be broken using weapons, and a *Teleporter*, which moves entities that enter in them to a far target place. Designers also require the modification of the *ResourceBearer* entity, which must have a *currentEnemy* attribute for the artificial intelligence. The expert captures these domain changes by modifying the entity hierarchy and uses the component suggestion module to distribute responsibilities. The application of FCA to the current domain results in the lattice in Figure 9.5, where formal concepts are tagged with letters from *a* to *n*.

Comparing the new lattice with the lattice of the previous FCA application (Figure 9.3), this technique determines that the pairs of formal concepts *<1,a>*, *<2,b>*, *<4,d>*, *<7,f>*, *<9,k>* and *<11,m>* remain from the previous to the current iteration. When it finishes this auto-

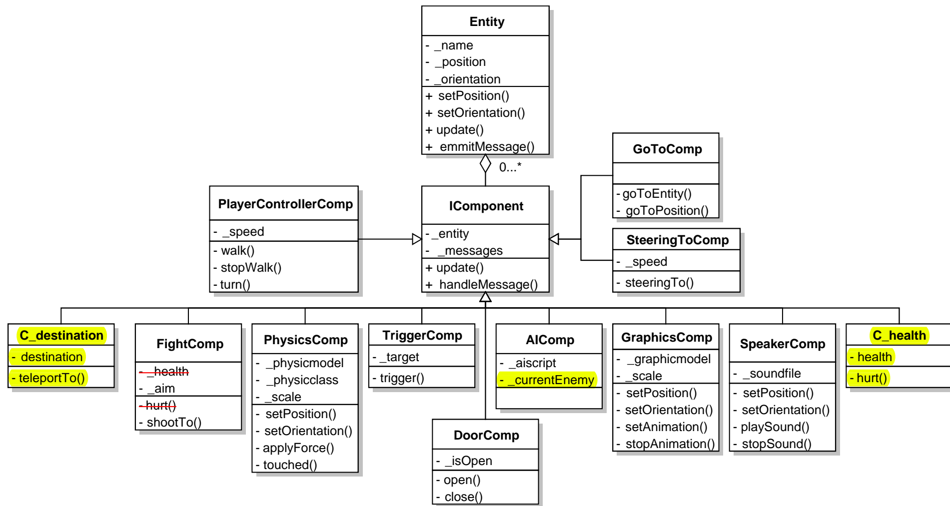

Figura 9.6: The new candidate components proposed by the technique

matic match, the formal concepts that were not put into pairs and with no empty reduced intent are presented in the screen. In this moment, the expert put the formal concepts <*3,c*>, <*5,e*>, <*8,j*> and <*10,l*> into pairs, based on their experience and in the fact that these concepts are very similar (only some attributes changes). Just the *g* and *h* formal concepts have no pairs and will become new components.

So, in these steps, the part of the lattice that does not significantly change has been identified and the modifications applied in the previous lattice are extrapolated to the new one. After applying the operators to the new domain, the new set of candidate components are finally given to the expert. Figure 9.6 shows these components, where we can compare the result with the components in Figure 9.4. The general structure is maintained but some actions and attributes has been moved between components. Furthermore two new components have arisen. The stressed features denote new elements (or moved ones) whilst the crossed out features mean that they do not belong to this component anymore (*FightComp*. At this point the expert could continue with the iteration by applying new operators to this set of components (i.e change the autogenerated names of the new components).

9.5. Consistency checking

It is clear that the component-based architecture promotes reusability and extensibility and therefore it is a good solution from the software

engineering point of view. However it is not without some drawbacks. In the next subsections we present three main issues that this approach to fight against the static class hierarchy solution manifests.

With a data-driven architecture where is so easy to create new entity types, things may go really wrong, creating inconsistent entities that cannot work at execution time. Before going down into details, we can make an analogy with the use of scripting languages to create behaviours. This languages extract the behaviour out of the game engine which therefore becomes much more independent of the game. However, a buggy script may prevent the game to execute correctly and even stall the application. In that sense an error in a input file provokes a wrong behaviour at execution time. When using the component-based approach, more data is left outside the application and therefore more chances exists to fail the execution because of external aspects. In both cases the core reason is that developers loose the feedback about whether things are correct or not because the game it is always able to start its execution, losing the ideal “if the compiler does not beep, my program should work” (Sweeney, 2006).

9.5.1. Entity inconsistencies at a component level

Regarding components, a source of errors come in the form of component dependencies. In a class hierarchy, derived classes depend on ancestor classes to work properly; for example, the *goToEntity* method of the *WanderingCharacter* (Figure 9.2) use the *steeringTo* of its parent class. In a componentized world all these capabilities have been split in different components. However in some sense there are higher level components that require other components to work properly. Notice that this dependency does not require a particular type of component, but any component that can fulfill a given goal, i.e. can process a given type of message.

Following with the example, for executing a *GoToEntity* message, a *WalkTo* componentwill need the existence of another component that executes *SteeringTo* messages (i.e a *SteeringBehaviour* component), while, we can imagine a *TeletransportTo* component that can process *GoToEntity* messages without requiring the collaboration of a companion component processing steering behaviour messages: it just moves the entity to its destination.

The key point here is that when a programmer is implementing a component that depends on others, nobody can guarantee to him that every entity that eventually will own this component, is going to also

have components that can process its messages.

When working with a traditional inheritance-based architecture, these inconsistencies are checked easier, since the feedback is provided by the compiler. When an entity has not declared a method, which is invoked by another method of the same entity, an error arises at compilation-time. However, in a component-based architecture, when a component sends a message to the entity that it belongs to, and no sibling component can process this message, no error is fired at compilation-time. Even no error is produced at run-time, simply the functionality is never executed. Due to the lack of an explicit error, the task of debugging is harder and it is difficult to know where the reason of this failure is.

9.5.2. Inconsistencies due to attribute definitions

In a fine-grained approach, entities are populated with attributes as in hierarchies. During the creation of an entity, a list of key-value pairs is given to the entity in order to initialise their attributes. This list results from combining default values defined in a blueprint and the values for a particular instance of the entity, as specified in a map file. During this initialization of attributes, the entity assumes that these key-value pairs are consistent: values correspond with an expected data-type and is within a valid interval. Although the same problem exists in inheritance-based systems this can be a big source of inconsistencies.

Moreover, what is new in a component-based architecture is the fact that components need that the entity they belong to owns and initialises every attribute that they need. We are talking again about dependencies that are not checked during compilation-time: at run-time components will require attributes that the entity could not have.

9.5.3. Entity inconsistencies due to a level configuration

While defining a new level map, designers are in charge of three main tasks:

- Create the *environment* of the map, using static meshes such as walls, corridors, terrains, trees, buildings or whatever the game needs.

- Add interactivity by means of *entities*. Programmers would have generated *prefabs* using the domain model with the accepted entities, containing those software components that programmers have approved. Some of the components would be *scripts* that could

be not developed yet, but programmers will be eventually fill them in.

- Define entities behaviours using Finite State Machines (FSM) or any other friendly AI technology, and incorporate them into the entities using a special component designed to run them.

It is important to note that the formal model contains information about entities referring their abilities (actions they must perform such as "*Attack*" or "*MoveTo*"), senses (perception capacities such as "*Enemy-Hidden*" or "*SufferDamage*") and state ("*health*" or "*armor*"). Nevertheless, the formal domain *does not* contains *when* an entity attacks or what it would do when *SufferDamage* because that would depend on the concrete map and difficulty level.

Entities become therefore the game level *building blocks*, and designers must create their concrete behaviours using a FSM editor. In this phase we *validate* the FSMs testing if all the actions used in the FSM states are available for the entity, and whether all the events specified in the transitions can be "sensed". With this purpose we translate the FSM in a fictitious OWL component:

```
Class: FSMName
  SubclassOf: Component
    and (interpretMessage some event/sense)
    and (isComponentOf only
          (hasComponent some
            (interpretMessage some action)))
    and (isComponentOf only
          (basic atributte some type))
    and (isComponentOf only
          (hasAttribute some complex attribute))
    ...
```

On the other hand, designers could be tempted to create their own entities joining together the *components* available instead of just using the prebuilt (and approved) entities (*prefabs*). Playing around with new entities is not necessarily bad, and provides a way for designers to experiment and test new ideas without affect the formal domain. Unfortunately, the component dependencies could be unattended and the resulting entities could suffer of inconsistencies because, after all, some of the components are artifacts useful for programmers that designers usually ignore. In this moment we can use the formal domain to look for inconsistencies in the map analysing the new entity types in search

of problems. It should be notice that once the new entities have been tested and designers consider that they are useful, they should be incorporated back into the formal domain so they will become *prefabs* in the next game iteration. Inclusion of new entities could result in code refactorization in order to reuse or split previously defined components, so entities created manually should be approved by both designers and programmers.

In fact, although the assistance provided is useful from the very beginning, where our tool really shines is in subsequent iterations. When the game domain is changed, we are able to track the differences between the old and new versions and refactor all the code preserving that one added by programmers to the generated templates, and moving it around to their new locations when needed. This constitutes an unvaluable tool for the agile development cycle because the error-prone task of changing code is avoided.

On the other hand, designers also loose the fear to changes because the old entities (prefabs) are automatically updated with the new ones, and parameters specified in the Unity IDE (such as the initial energy of an entity) is automatically updated even if that value is moved from a component to another one. FSMs are also reevaluated, so all the tests that they suffer while being created are executed within the new domain in order to discover inconsistencies such as an action that has vanished and a FSM is still using.

All this consistence checking (similar in spirit to *test-driven development*) for both programmers and designers makes viable an agile development cycle and is in some sense immune to changes and additions in the game design.

The OWL domain representation that is shared in all the levels of the game and it is considered the *static model*. However, when creating a level, designers create new entities, change the default attribute values and create some AIs. All these information is also integrated in our OWL representation to check the *level consistency* and constitutes the *dynamic model*.

9.5.4. Example

In order to validate and show our method we have developed a small game example using Unity. This example consists of a simple 3D action game where the player controls an avatar and the system controls several non-player character (NPC), which can do different things such as protect an area, explore the environment or look for the player. De-

signers are responsible for defining the game in depth, determining the details of the playability, aesthetic aspects, history, etc. According to our methodology, this means that they are in charge of creating a hierarchical distribution of entity/game object prototypes. These entity prototypes are defined with the state and functionality in terms of attributes they have and actions they are able to carry out. Designers can also set values to the attributes that populated the entity prototypes although these values will be editable later in a per instance basis.

This domain created by the designers represents a definition of the game entities that makes sense from a semantic point of view but, before implementing the game, this domain must be accepted by the programmers. When the programmers accept the given domain they become responsible of adding all the extra information needed for the implementation of the game. They create new entities to represent abstract things such as the camera o triggers, new attributes without semantic interest and possibly new actions and senses that, from the programming point of view, correspond to messages that entities send and accept in the component-based architecture (West, 2006). After completing the population of the domain, programmers distribute the attributes and the functionality among components obtaining components like *Movement* that allows a entity to walk through the environment at a set speed or the *Combat* component that gives the entity the ability to attack in a hand to hand combat.

Figure 9.7 shows the distribution finally obtained by designers and programmers. Entities shadow in grey (Character, Level or Tree) are only important from the semantic point of view since they will not exist as such in the final game; their main purpose is to enhance the ontology knowledge, but only black entities (NPC, Player, etc.) will be instantiated in the game. The figure also shows some components, actions and attributes in their hierarchies and in the inspector we can see an example of a entity definition with components, actions that is able to carry out, and attributes, which are not visible in the figure.

When programmers decide they have finished the technical design, programmers generate the content needed for the creation of the game. Unity scripts are created with the majority of the code and programmers only have to fill some blanks with the concrete functionality. These scripts correspond with the components that are in the ontology. Prefabs, defined as entity prototypes, are also created in Unity and are used by the designers to populate the level with entities. Figure 9.8 shows the level created by designers using the prefabs or entities created on the fly by adding components. The figure also shows an NPC in the inspector

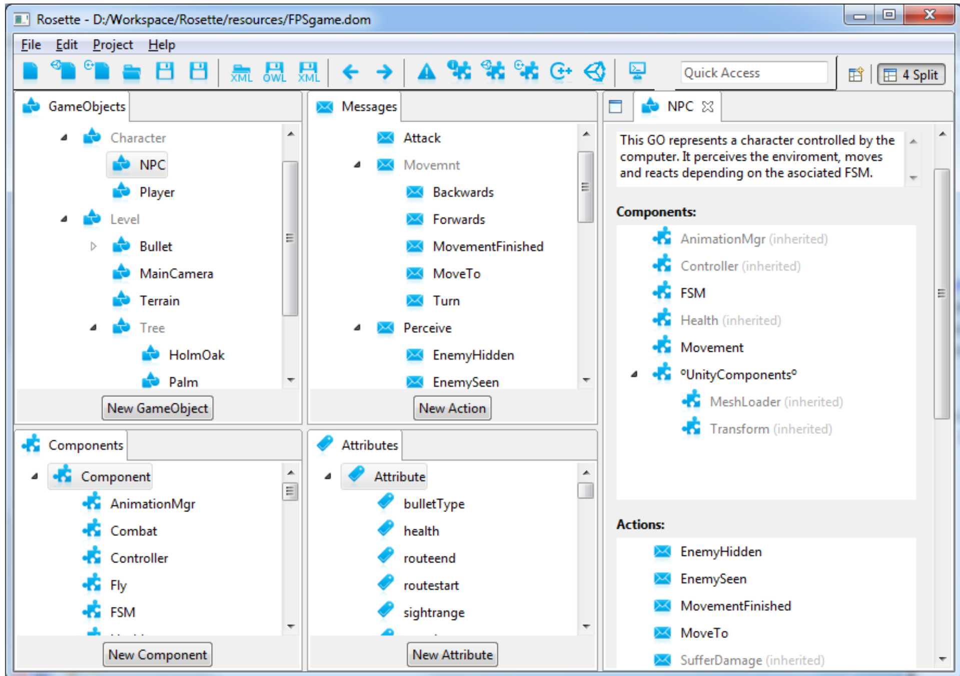

Figura 9.7: Example distribution

with the components/scripts that programmers created.

For this NPC the designers have created a simple FSM (Figure 9.9) in an external editor to determine how this entity should act during the game execution. The created FSM keeps the NPC moving by the scene and, when the NPC receives an attack or it sees an enemy, it changes its behaviour and attacks the enemy. Then, it returns to its normal behaviour when it loses sight of the enemy. On the other hand, in the level editor (Unity), the designer altered some default values, changing for example the speed value from 5 to 20, to personalize the specific entity and to adapt it to the associated FSM.

So, the level design comprises several task where designers must express their talent putting different building bocks together. As they combine very different pieces of functionality, the level edition usually causes a great number of inconsistencies:

- During the domain design, designers and programmers creates attributes that will be added to entities and components. An attribute is defined as a basic o complex type and could have some constraints like an attribute accepting only a set of values or a range of them. In our example, it was set that the *speed* attribute can only adopt values from 0.0 to 10.0.

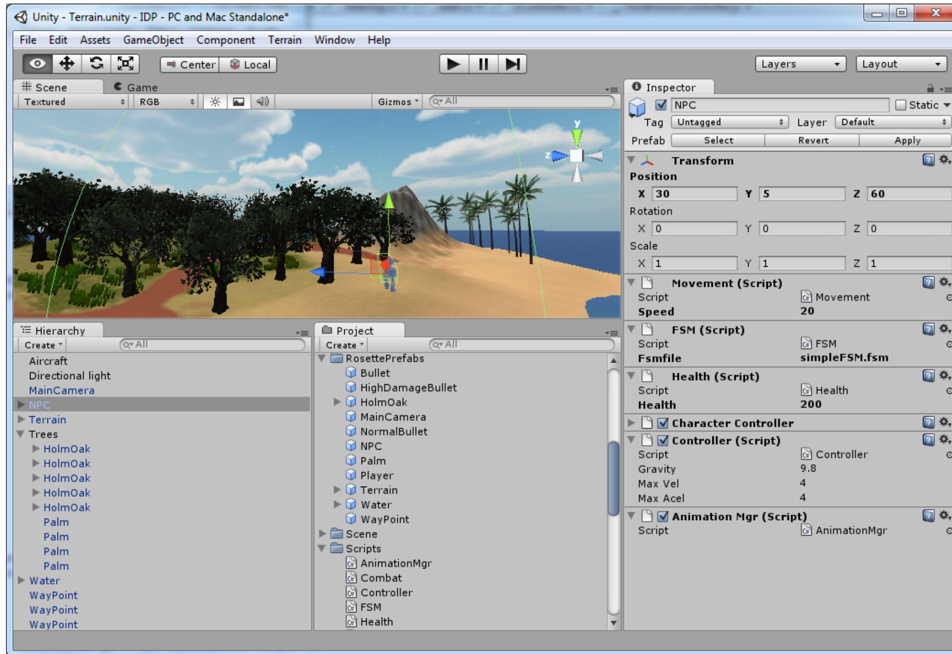

Figura 9.8: Unity example

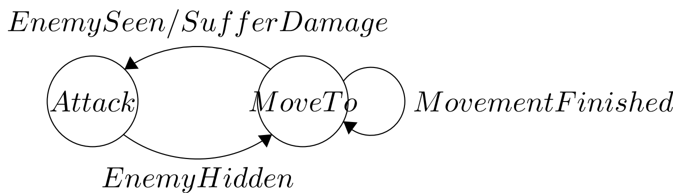

Figura 9.9: Simple FSM

- Assembling entities on-the-fly, by putting components together, can cause unpredictable failures during the game execution because components sometimes require other components to work properly. For example, the *Movement* component, in order to carry out a *MoveTo* action, need to set the *walk* animation so the entity needs to have another component that is able of setting animations (i.e. *AnimationMgr* component).

- Designers create FSMs and associate them to entities what can cause inconsistencies and errors similar to the previous point. A FSM usually needs to execute actions and a designer can associate a FSM to a entity that is not able to execute some of those actions. The FSM of the Figure 9.9 need to execute *MoveTo* and *Attack*

a) NPC entity definition:

```
Character
and (hasComponent some Movement)
and (hasComponent some FSM)
and (speed some float)
and (fsmfile some string)
```

b) Movement component definition:

```
Component
and (interpretMessage some
      MoveTo)
and (speed some float)
```

e) iNPC individual:

```
Object property assertions:
  hasComponent iAnimationMgr
  hasComponent iController
  hasComponent iHealth
  hasComponent iMeshLoader
  hasComponent iMovement
  hasComponent iFSM
  hasComponent iTransform
Data property assertions:
  fsmfile ""
  sightrange 10f
  health "100"^^unsignedInt
  speed 20f
  meshFile "Lerpz"
```

c) speed attribute range:

```
float
and minExclusive 0f
and maxInclusive 10f
```

d) FSM “component” definition:

```
Component
and (interpretMessage some
      EnemyHidden)
and (interpretMessage some
      EnemySeen)
and (interpretMessage some
      MovementFinished)
and (interpretMessage some
      SufferDamage)
and (isComponentOf only
      (hasComponent some
        (interpretMessage some
          MoveTo)))
and (isComponentOf only
      (hasComponent some
        (interpretMessage some
          Attack)))
and (isComponentOf only
      (fsmfile some string))
```

f) Attack message definition:

```
Message
and (target some string)
```

Figura 9.10: OWL definitions for entities, components, messages and attributes

actions thus at least one component of the entity should be able to carry out those actions.

These inconsistencies constitute terrible headaches for designers because its difficult to find out where the problem is. As problems happen during the execution of the game, the feedback about the problem is non-existent and sometimes the problem is because of technical reasons that designers should not be aware. Because of this, we provide desig-

ners with the possibility of checking whether their domain is consistent or not with the formal domain they specified before. In the previous section we explained that part of the OWL domain is static (this part related to the attributes, messages and components) and some part is dynamic and created depending of the concrete level (this part related with the FSMs and the entities). Figures 9.10b, 9.10c, 9.10f show examples of the static part of the domain extracted from the game.

Related to the dynamic part of the OWL domain, Figure 9.10d is an example of the FSM created by the designer (Figure 9.9). The FSM is represented as a component that define the events that trigger transitions in the FSM as messages that can be carried out by the fictitious component. In this case the FSM reacts to *EnemySeen*, *EnemyHidden* and *SufferDamage* but, when these events happen, the FSM has to execute an action that should be executed by other component so the entity must contain components that can execute *Attack* and *MoveTo* actions.

As we allow designers to create their own entities with the existing components and they also create FSMs, which in OWL are just new components, we must also generate entity definitions dynamically. Figure 9.10a shows the NPC with its components and attributes while the (Figure 9.10e) represents a concrete individual where the attribute values are defined as data property assertions. With this semantic information automatically generated when the designer checks the level that he designed, we are able to reason and determine whether the domain violates a property or, on the contrary, the level is consistent with the domain definition. In the example presented during this section designers create a level with some inconsistencies. Figure 9.10 reveals them:

- The *speed* attribute is set with a not allowed value due to the range was set between 0.0 and 10.0 (Figure 9.10c) by the game designer but, however, the current value has been set to 20.0 (Figure 9.10e). In this case the level designer is informed about the error and he must set a valid value.

- The FSM associated to the NPC will not be able to be executed due to this NPC must have other components with the ability of executing the actions that the FSM has in its states. Although the entity is able to carry out the *MoveTo* action, through the *Movement* component, there is no component in this entity able of carrying out an *Attack* action. In this case the user is just informed that he must add a component able of executing an *Attack* and will recommend to use the *Combat* component, the only one of this component collection that is able to do it.

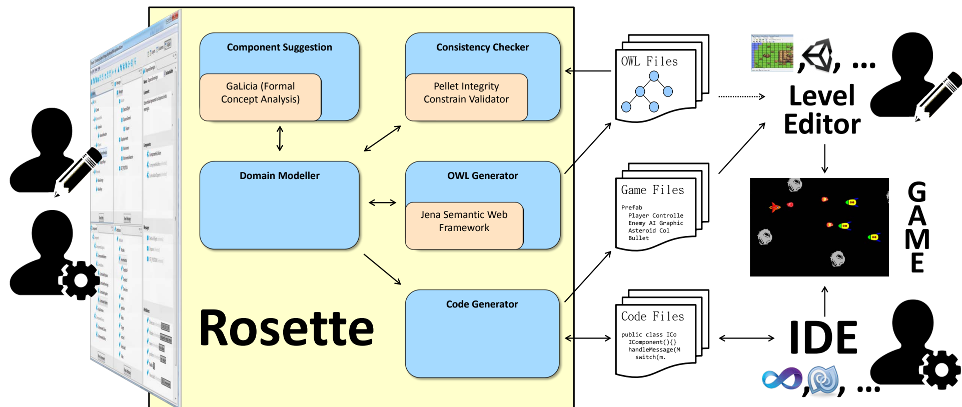

Figura 9.11: General view of the *Rosette* architecture

This feedback is very useful to the designer not only when creating a new level but also to check if an old designed level is still consistent when a code and design refactorization is done. This way we promote agile methodologies simplifying the problems between iterations.

9.6. Rosette

Although our methodology is valuable as-is, its main benefit comes when it is supported by tools that help in the construction of the game model, code generation, and consistence checking.

In that sense, we have developed a visual authoring tool called *Rosette*, an easy to use visual tool that fills this gap. *Rosette* acts as a central point where designers and programmers come to an agreement (see figure 9.11). The tool is aware of these two different roles and presents different information and options depending on the role of the current user.

Though the current state of the application allows an iterative development, in the description that follows we will assume that we are using a waterfall model, just as we did in the previous section.

At the beginning of the development process, *Rosette* is used by technical designers. There they define a kind of conceptual hierarchy of the entities that comprise the game. *Rosette* allows adding to every entity a comment that acts as documentation and a set of other properties. In particular, technical designers may specify a set of attributes (*name*, *health* or *maxSpeed* to mention just a few) and a set of actions that the entity can perform (such as *shoot* or *move*) . As the entities are organized hierarchically, the inheritance model applies and therefore the

set of attributes and actions that an entity has available also contains the attributes and actions of all its predecessors in the hierarchy. As an analogy, we can say that the process is similar to defining a UML class diagram but without taking into account the peculiarities of programming languages but the game domain itself, and therefore the final result is closer to an *ontology* than a class diagram (users at this point are technical designers but not programmers).

The final result of this step may be seen as an specification of the game or, put it in other way, the contract between designers and programmers, because it specifies the set of entities that designers assure they will need to build the game levels and, therefore, the functionality programmers must develop.

The complete game domain of *Rosette* (referring both designers'gameplay aspects and programmers implementation needs) is stored using OWL, a knowledge-rich representation that allows inferences and consistence checking. Entity verification allows to detect errors as soon as possible, instead of let them go down until the final source code and game execution, when it would be too expensive to solve them.

In the next step, programmers come to scene. Within *Rosette*, they have a more technical view of the domain and available options to enrich the previous information with other knowledge that may be needed prior coding.

After that, programmers may take a step further using *Rosette*: split entities into components. They, as experts, can manually specify the components that the game will have and then modelling the set of entities on the game domain according to them. The process is supervised by *Rosette*, and so, it checks whether the final result is complete, in the sense of assuring that all the attributes and actions an entity has are covered by the chosen components for that entity.

Moreover, *Rosette* may aid in the process of generating components Instead of having developers creating the set of components from scratch, *Rosette* may give them a initial proposal to begin with. As the fully-fledged domain model allows us to use reasoning engines, it can be used together with Formal Concept Analysis (FCA) for extracting an initial set of components. FCA is a mathematical method for deriving a concept hierarchy from a collection of objects and their properties (Ganter y Wille, 1997). This may be useless in our case, because we do not have a collection of objects but a hand-made conceptual hierarchy and therefore it may be argued that FCA cannot help in the process. However this is not entirely true. In short, *Rosette* converts the game model (a conceptual hierarchy by itself) in a set of objects and feed them into FCA. The

final result is not the original ontology but a (formal) concept hierarchy where each (formal) concept represents not one but maybe more than one original concept sharing a set of properties. Since a component is also a set of properties (actions and attributes) that may be used by several objects (entities), *Rosette* uses this (formal) concept hierarchy to automatically identify a component distribution that will fit the requirements of the provided entity ontology.

The candidate distribution is shown to the user through the visual interface, where he has the opportunity of modifying it by merging or splitting components, adding or removing extra functionality, changing names, etc.

At this point, the modelling phase is over and both level designers and programmers start working independently on their respective tools: level editors and IDEs. *Rosette* has the capability of integration with two different target platforms: the well known Unity3D game engine and an in-house engine developed by the authors in C++ that is being extensively used in our master degree of videogames in the Complutense University of Madrid and supported by *Rosette* to use it in our classes. So, once designers and programmers agree, the target platform is selected in *Rosette*, and it generates the input files for their respective tools. In case of Unity3D, the project is populated with a prefab per entity and a C# file for every component. Prefabs are preconfigured with the set of components specified in the domain model and the initial values of their attributes (if any). On the other hand, the source files act as skeletons with hooks where programmers will add the code that performs the specific actions assigned to the components. By contrast, if the selected target platform is our in-house game engine, a plethora of C++ files (both header and implementation files) is created and some XML files are generated containing the list components of every entity (*blueprints*) and the default values of their attributes (*archetypes*). *Rosette* has been designed to be extensible in the sense that many other target platforms may be easily added by means of template files.

Figure 9.11 summarizes the process: both designers and programmers use *Rosette* as a domain modeller tool. The game model is then used for consistency checking using OWL and for component suggestion using FCA. When the process is complete, *Rosette* code generator creates game files meant to be read by the used level editor and source files that will be enriched by programmers through IDEs.

Summarizing, the main advantages of *Rosette* for supporting the methodology are:

- *Join point between designers and programmers*: instead of having designers writing game design documents that are hardly read by programmers, *Rosette* acts as a union point between them because the model created by designers is the starting point for programmers work.

- *Easy authoring of the game domain*: designers are comfortable with hierarchical (not component-based) domains. Using *Rosette*, they are able to create the entity hierarchy enriched with state (attributes) and operations (actions).

- *Semantic knowledge stored in a formal domain model*: the game domain is transparently stored in a knowledge-rich representation using OWL. Therefore, *Rosette* is a working example of using ontologies for software engineering (Seedorf et al., 2006) and introduces an Ontology Driven Architecture methodology (Tetlow et al., 2006) in the game development cycle.

- *Automatic component extraction*: using Formal Concept Analysis (FCA), *Rosette* may be used for suggesting the best component distribution for the entity hierarchy specified by designers and enriched by programmers and allows manually fine-tune that components.

- *Consistency checking of the game domain*: the semantic knowledge allows validation and automated consistency checking (Zhu y Jin, 2005). As an example, *Rosette* guarantees that manual changes done by programmers to the component distribution or changes done by designers in the levels are coherent with the game domain.

- *Code sketches generation*: as in Model Driven Architectures (MDA), *Rosette* boosts the game development creating big code fragments of the components. This saves programmers of the boring and error-prone task of creating a lot of component scaffolding code. *Rosette* is platform independent in the sense that its code generator module is easily extended to incorporate other target platforms different than the two currently supported, Unity3D and plain C++.

9.7. Methodology to teach the component-based architecture

Component-based software architecture deconstructs object-oriented mechanisms in order to promote reusability by generating game entities through the composition of fine-grained components. Although the main ideas of this approach are usually easily grasped by a Computer Science graduate, it usually takes time and a number of errors to developed the skills required to come up with a good component-based design in terms of reusability, extensibility and lack of functionality duplication.

For that reason, we designed a new teaching methodology that allows students to very rapidly iterate through increasingly complex versions of simple games, allowing them to try and evaluate a large number of component distributions, thus accelerating the learning process. This approach is made possible thanks to *Rosette* because it maintain an ontological view of the entities in a game and connecting it to aggregations of components in the source code.

In this section, we have developed an iterative method to assess the use of *Rosette* in students'comprehension of component-based architecture design. The method involves the development of several iterations of two simple games, the first one developed in Unity and the second one programmed in C++.

9.7.1. Methodology

As teachers in a master degree of videogames in the Complutense University of Madrid, we have faced some difficulties while teaching component-based architectures in the last few years. Thus, we adopted a method that allows us to introduce *Rosette* and use it as a pedagogical tool to allow students to assimilate component-based design and methodology.

We used to put into practice a methodology where students implemented a 3D game using a component-based architecture in a guided way. After some basic explanation about the theoretical aspects of the architecture, they were told what components they should implement for developing the small test-bed game, and why. We provided them with sample components, code sketches and many placeholders where students had to write specific code for implementing the game.

Although students seemed to assimilate the inner workings of component-base architecture, when they were confronted with their master project where they must implement their own videogame nearly from scratch,

they suffered a lot of difficulties trying to distribute the functionality among components. The first prototypes used to have both very big components with low cohesion or very specific non configurable components. None of them are reusable, so each new feature, kind of enemy or change in the game design supposed a new component, usually very similar to an existing one, causing duplicity of code (and bugs).

Our conclusion was that, although component-base architecture fundamentals are easy to understand, get proficiency in identifying and designing a good component distribution is not so simple. After all, component-base architecture breaks in some way the object oriented programming paradigm, so students must change their minds when designing components.

In order to overcome the previous difficulties, we have changed our teaching methodology. We confront students with two small videogames where they must not just fill in the gaps, but completely design the entity layer, identifying entities and distributing their functionality into components.

To emulate a complete videogame development, where game design is usually a moving target, students are provided with a partial definition of each game, that is enriched in a progressive way after some iterations. Our intention is to force the students to reevaluate their previous components distributions, trying to reuse as much component as possible or deciding when they are forced to refactor. In this way, we mimic the challenges students will afford during their projects, but in a more controlled environment.

Instead of leaving students completely on their own when developing their game, they will use *Rosette* as a help. Each iteration consists of the next stages:

- Students are provided with a *running example* of the game they must implement at the end of the current iteration. We could have used a prototypical game design document explaining the requested functionality. But those documents require some time to be analysed, and are prone to misinterpretations. A running game accelerates the comprehension phase and avoids problems. With the executable, students receive also all the graphical resources that they need to implement their own clones of the game.

- Once new requirements are assimilate, students must focus in distributing the functionality among components and defining the entities of the game. For this task the students use *Rosette* which helps them to create a collection of components in the first itera-

tion or to update this collection in subsequent iterations. In this stage the students keep their mind clean from the implementation details, what let them create a better semantic distribution.

- Finally, students use *Rosette* to (re)create the code sketches, and implement the new functionalities they added in the previous stage. As *Rosette* is an iterative development tool, they can generate the code fragments at any moment without losing the changes and additions done in the source code files of previous iterations.

We have defined the process as three stages but, during the same iteration, students could move from *Rosette* to the implementation stage more than once in the case that they did not correctly specify the domain model and component decomposition in the first try.

As teachers, our aim with those practise sessions is teaching how to distribute components, but the motivation of the students is to create videogames. In that sense, *Rosette* greatly speeds up the development of the iterations and eases the component-base architecture refactorization. This means that students can very quickly test different component distributions without getting worried about the subsequent changes in the source code, because the tedious work of rearrange it is facilitated by *Rosette*. We will shown that this possibility has become a great mechanism for learning (and teaching) component-base architecture.

9.7.2. Methodology in Practice

We have put into practice the ideas presented above in two separated experiences with the same group of people. All of them were students of our post-graduate master on game development at the Complutense University of Madrid, and most of them were graduated in computer science. 19 students were enrolled, only a few of them have prior experience developing games and none having ever created a component distribution.

Before the experiment took place, they were confronted with the creation of small games using components. Specifically, in the first weeks of our studies, game development process are introduced with the help of Unity, a component-based game development platform. Teachers guided students to build, step by step, some small games. After that, they attended theoretical lectures about game architecture in general, the old inheritance based implementations and component based architectures.

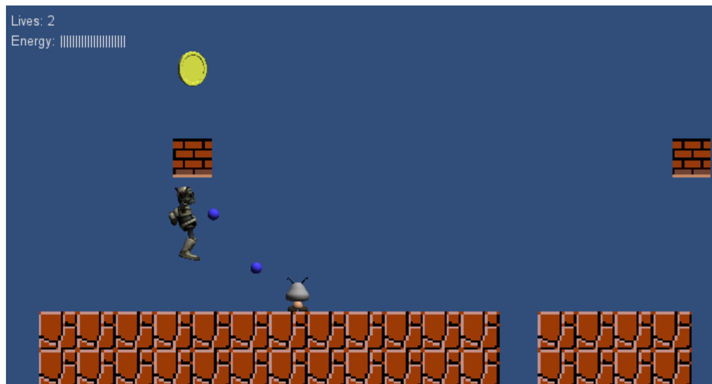

Figura 9.12: First game, created using Unity

Our experiment took place in this context, and was split in two different experiences. The first one consists on using Unity itself to build a small game. Unity has a built-in library with common components, so students had a collection of components to begin with and to extend. *Rosette*, which has an integrated semantic representation of all those components, is able to generate Unity compatible C# code. Therefore, students could use it as an external tool where model the game ontology and components before implementing the details in Unity.

After that, we switch to a game development from scratch, where they had to code the entiry game using C++.

9.7.2.1. First game: using Unity

the goal of our study is to teach students how to create a video game from scratch using a component-based architecture. Thus our research question is: do students learn how to effectively use component based architectures when developing games from scratch if we introduce it through our methodology and the use of *Rosette*?.

At the end of the course, students have a good understanding of the inner software pieces that compound a traditional videogame. However, to alleviate the stepping into this world, the first game that the students must create will be developed using Unity. Unity is a cross-platform game engine and IDE that makes the development of videogames easier supplying a visual game editor, subsystems management, scripting languages to create the game content, and a great amount of other features. It has a well-designed entity manager based on the component-based architeture and has a build-in component library that resolves the technical aspects of the videogame, such as realistic physics, collisions

and rendering.

All of these features make Unity an ideal tool for our first experiment. Students will be faced to the creation of a videogame and they will have to worry just about designing the components that implement the specific funcionality of that game, having other components, a professional level editor and a high-level language such as C# available, much more easier than C++.

The game students had to implement was a 2D side-scrolling platform game. However, the first iteration was no more than some static platforms, one platform that moved itself between two points, a player that can move left and right (using the keyboard) and falls if there is no platform below him, and the camera logic that scrolls to always show the player on the screen (except when it falls). This is a very basic and simple functionality for a game but enough to let students think in a first distribution of the functionality in components. The idea is that if they create a good starting distribution the next steps will be easier.

In the second step of the game development, students had to continue adding some features to the game. The player had to be able to fly during some seconds, using its rechargeable batteries, and it had to die when it fell into the void. When the player died it lost a life and starts again and, if it lost all the lives the game ended. This second iteration is similar to the first one because students only had to divide the functionality among the previous components or new ones. However in the subsequent iterations a good distribution helps a lot. For example, in the third iteration we introduced the first enemy, that moves between two points and, if the player came into contact with it, the player died. If they had done a good component distribution, this iteration was trivial because the enemy moved like the moving platform and killed the player as when it fell into the void. However, if they did not create a good component distribution they could identify and solve it at this step, learning from their own mistakes.

The next iterations followed a similar idea. They introduced new features where some of this functionality should be reused from previous iterations. The iterations fourth, fifth and sixth added typical blocks of these games that, when the player hit them with the head, they drop some kind of power-up. Specifically, the fourth iteration includes the first block of the game, which dropped a coin that moves up, then down and finally disappears. In the fifth one, the dropped item was a star that moved to the right and when touched the player the star killed him. The last dropped item, in the sixth iteration, is a ball that, when it came into contact with the player, provides him with the ability of shooting small

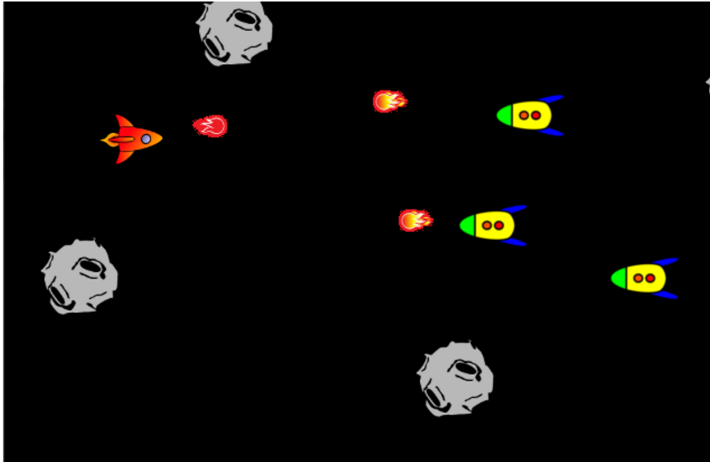

Figura 9.13: Second game, build in C++ from scratch

rocks that disappear after a couple of seconds. The last iteration made the enemies to die (disappear) when a rock hits them.

All those iterations shares many functionality between them and with previous iterations, so students can realized by themselves about their mistakes in the component distribution. For example they could have created a specific no flexible component for the block that drops coins (instantiate them when the player collided it) but, the fifth and sixth iterations would make them (in some cases) parameterize this component to let them create any object passed as parameter when the collision happens. Other examples are that they can reuse previous features for moving the dropped object, killing the player, make elements disappear, etc. In conclusion, the students learn from their bad distributions and can correct them during the iterations.

We think that this was a good starting point because *Unity* eases the development, provide some built-in solutions in form of components, and because using *Rosette* the students can concentrate their efforts in the semantic description of the entities, components, etc. Last, but not least, when they had to refactor the component distribution, *Rosette* alleviated the tedious implementation task.

9.7.2.2. Second game: C++ from scratch

As discussed earlier, the main purpose of our studies is that the students learn all the pieces of a videogame architecture. Therefore, all over the year students create a 3D videogame using C++ where the content and functionality is created in an incremental way. Students keep

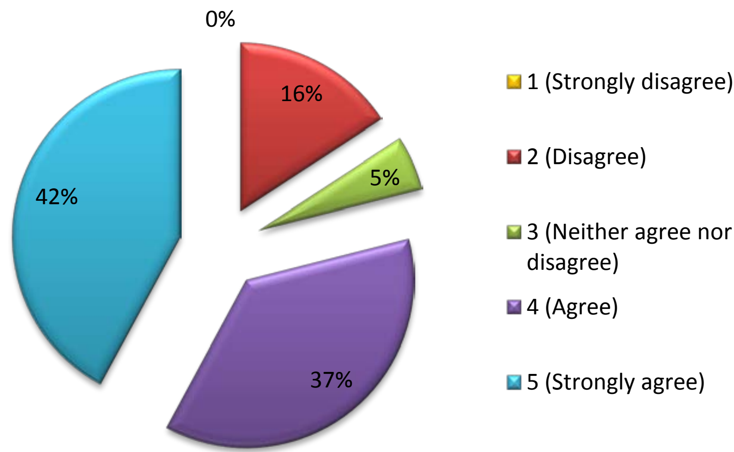

Figura 9.14: Statement: *Rosette* and the methodology, to teach (or learn) what a component-based architecture is, is very useful

adding different software pieces into the final application when they attend the theoretical classes that explain them.

Using C++ technology developed nearly from scratch means that the students lose all the facilities that Unity provides to them such as the level editor, the scripting languages or the entities management. However, the main difference refering our teaching purpose is that they started with an empty logic layer where there were no built-in components. Students need to design and develop those more technical components (that Unity provides) and cope with the different subsystems (i.e. the graphic system responsible of the rendering).

For this second part of the experiment, the students had to create a side scrolling shoot-em-up arcade 2D game where the player controls a spacecraft shooting enemies whilst doing their attacks. The genre of the two games are similar and they share many features such as that they are 2D games, they use side scrolling to advance in the level or in both of them the user controls just one character that can move and shoot. That way, students were able to use the knowledge learnt in the previous stage of the experiment and apply it. The game, though, is different enough to present the challenge of thinking about what is the best component distribution because the previous one cannot be used as-is for this one.

This stage of the experiments works as the first one: they would have to create this videogame in several iterations. In that case the entire process took up to five iterations. In the first one the students had to create a first distribution that included some static asteroids, a spacecraft that automatically advanced through the level but that was able also to move

forward, backwards, up and down with the keyboard and the logic that made the camera advance through the level at the same speed than the player. This initial iteration seems similar to the first iteration of the previous game but with the already mentioned difficulty that they have to create all the components (even those related with the graphics and the rendering).

The second and the third iterations almost did not modify any of the previous entities but added new ones. Firstly, the students had to add a new kind of asteroid that randomly moved forward, backwards, up and down and, secondly, they had to add spacecrafts as enemies that move like these asteroids but also shot randomly, feature that is also added to the player spacecraft when the user press the space key. The fourth iteration introduce collisions because until this iteration there were spacecrafts, asteroids and shots but nothing collided but everything went through the others. These collisions implied that spacecrafts and shots were destroyed when collided with any other visual entity with the exception of the player that was able to collide three times before it died and the game ended. Finally, in the fifth iteration we added a power-up, the player started without bullets and when it collided with the power-up, it got one hundred bullets.

As in the Unity game, the idea was that the students iteratively identified similarities between different entities in order to build components with the maximum cohesion and minimum coupling to enhance reused. In this game there were a lot of features that belongs to more than one entity. Examples are the constant velocity that the player, camera and shots have, the ability to shoot of enemy and player spacecrafts, the rendering or collisions abilities of almost every entity, the ability to react against collision been destroyed, taking a power-up, etc.

The key here is that students were able to identify good candidates of components in different moments of the development. An example of this is that if they had encapsulated the movement of the player in a component that received move messages, they could have reused it in the moving asteroids and the enemies but, if they did not have done it in the first iteration, the could have still done it in the second or third one. This means that they could do everything correctly from the beginning and finish the game in a short time, but as they were not experts but students they were able to learn how to distribute the features from bad distributions and correct them in future iterations. Moreover, they had to think in a high level view with *Rosette*, so it should have been easier to identify similarities (as features were not obfuscated in the code) and also to refactorize and try different component distribution (as *Rosette*

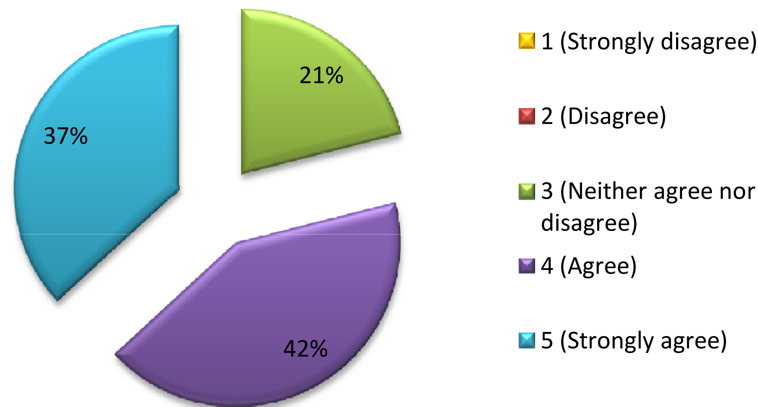

Figura 9.15: Statement: The iterative code generation of *Rosette* is very useful when the domain is modified because it preserves all the code implementation

deals with the code for them).

9.7.3. Evaluation

Before teaching these classes, our students knew the theory of the videogame component-based architecture and they have worked with Unity creating some prototypes in a guided way. However when they faced with the first videogame they had serious difficulties to divide the features among components what means that the previous theoretical and practical classes were not enough. In this section, we will show learning results once students are used *Rosette* in the games described in the previous section.

First of all, we must confess that one of the things that positively affected the teaching was the high motivation of the students. The motivation was high due to the fact that students were invested in developing their own games, as opposed to previous years were they had to develop a game based on other people's work based on a given assignment We really think that this is one of the keys and if we had made students to work only in a semantic way, without implementing the game, the result would have not been the same.

To evaluate our teaching approach, we wanted to measure the progression of the students in dividing responsibilities among components. To do this we adopted several methods:

1. We gave students a questionnaire at the end of the classes to gauge the usefulness of the tool as they perceive it.

2. We also collected data from each iteration, including code, the semantic domain modeled in *Rosette*, traces from *Rosette*, etc.

The questionnaire was composed of several quantitative and qualitative questions (in a on a likert scale). The Figure 9.14 shows that almost the 80 % of the students consider that the methodology of the classes was useful, whilst only the 16 % of them considered that to work in a high level point of view (using *Rosette*) does not provide anything to them and they preferred to directly work with Unity scripts or C++ (they told us so in the gualitative question). Figure 9.15 reflect that the majority of them (again the 80 %) finds useful to have the possibility to modify or refatorize the component distribution from a semantic point of view without paying a price in re-coding the functionality previously programmed that is moved to a new component. In this case, no student considered this feature as something undesirable.

On the other hand, we have then used the collected data to observe the results of our methodology focusing in different aspects. Our premise is that if the component distribution created by students along the iterations are good enough, they have properly learnt what the component-based architecture works. A form to measure the quality of the created components is seeing their reusability and flexibility so, if the components created in one iteration have been reused to implement new entities in subsequent iterations we can infer that they are good components. In the same way, if there are code replicated in different components we will assume that the distribution is not good enough due to there is a good candidate to be a new component. We can also understand that if the mistakes in early iterations are solved in later iterations the student has identified the mistakes and has learnt about them.

The first part of the experiment using Unity was hard for the students, nobody was able to finish all the iterations of the first game (Figure 9.16). In fact, a lot of them did not pass the second one. 31 % of the students only finished the first iteration whilst 42 % only reached the second one, 21 % the third and only 5 % of them finished the fourth iteration. Even worse is the fact that the component distributions were not very good in general and they dedicated a lot of time to design the components. The little good news was that some of them improved their distributions in the middle of an iteration or when a new feature was added (but barely the 30 %). The most difficult thing to them was to find similarities between abilities of different entities to reuse components. Almost every student created several components that includes very similar code and functionality. For example, no student was aware that the movement of first enemy of the game could be modelled using the

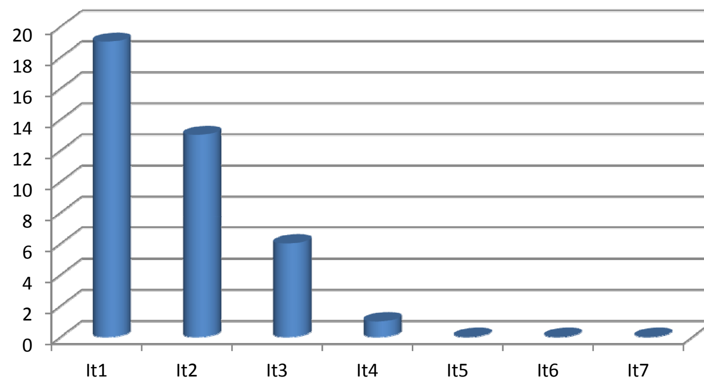

Figura 9.16: Number of students that finished the different iterations of the first game

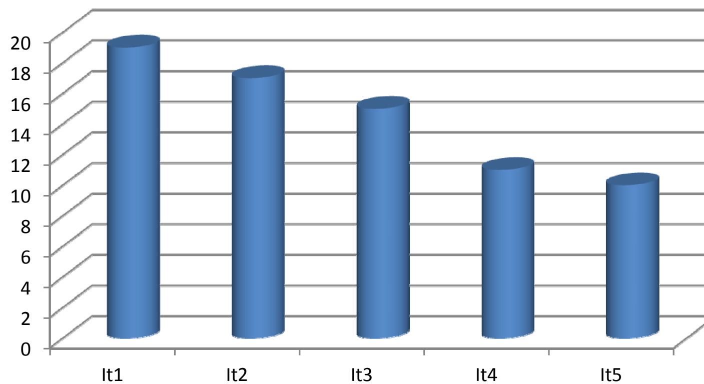

Figura 9.17: Number of students that finished the different iterations of the second game

component used by the moving platform, and therefore all of them ended up with similar pieces of code in different components. As another example, just one student identified that the code used to manage the dead of the player when falling into the void could be reused when it is killed by enemies.

However, when students started with the second game it seemed that they had learnt more than we initially thought. Figure 9.17 shows that there were a lot of them that finished all the game iterations (more than the 50 %) and the component distribution in this second game, although no perfect, were substantially better. Students spent less time in *Rosette* distributing the components and more time dealing with C++ and the game architecture.

Besides the previous data, we have analyzed the data collected from

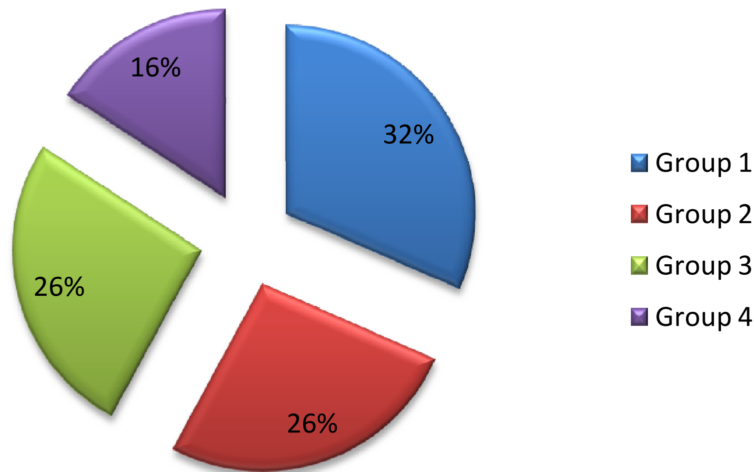

Figura 9.18: Groups of students with different progression and quality of their component distributions

the iterations in order to infer the quality of the component distributions the students did. To this end, we have analyzed in each iteration the semantic domain, the source code and the traces of *Rosette*. Having a glance at the domain of an iteration we can suppose if the entity and component distributions seems to be good or not because we can detect functionality that is duplicated from a semantic point of view (i.e. two components have the same attributes and messages to do the same functionality in different entity). However, although the view of *Rosette* is enough to detect bad distributions, it is not to assure that we have a good component collection. Consequently, we have also to inspect the source code of the components looking for duplicated pieces of code that reveal possible candidates to be new components. Finally, in the traces of *Rosette* we can see distributions in the middle of an iteration and we can see if the student refactorized its domain.

According with our data analysis, we distributed the students in four groups (Figure 9.18) depending on their progressions in the development and the quality of the component distributions they did in the iterations. The students of each group are characterized themselves by:

1. They were able to identify from the beginning the reusable components the game design should used. These students had no mistakes splitting the functionality in components and, over the iterations, they just needed to make very little improvements, adding parameters to adjust the component behaviour.

2. They did not create a perfect component distribution in its first at-

tempt but as the iterations advanced, they refactor them, finishing with a component distribution as good as the one reached by the first group.

3. They were not able to clearly classify all the features of the game; they identified some good components that were later reused in subsequent iterations, but they also missed important ones such as the movement component that includes the code to set the position of the entity according to its speed (a component reusable in player, enemy ship and asteroid).

4. The distribution of these small group of students did not pass the minimum threshold to be considered a good distribution because of the use of big non-reusable components or because of not having understood the component based architecture at all.

9.7.4. Conclusions and Future Work

In this section we have present the tool and the method we have used for teaching component-based architectures (component-base architecture) for game development in a Computer Science post-graduate master. Students are usually fluent with object-oriented design, using hierarchies, but they need to adapt their habits to the new architecture philosophy.

Our method consists on confront students with a couple of small games that they must implemented on their own, instead of in a step-by-step way. Although the proposed games are simple enough, it is unrealistic to expect students will be able to experiment with different component distributions and, at the same time, to write all the code in a short time.

To solve this problem, we use *Rosette*, a graphical tool that let the user define the semantic game domain, and provides iterative consistence checking and code generation. It boosts the game development cycle, something very useful for the first stages of component-base architecture learning where component refactor is quite common. Students are also motivated during the practice sessions, because they have the opportunity not just of learning component-base architecture, but also playing their evolving game.

Although students do not, obviously, become component-base architecture experts in just 12 or 15 hours, results have proven that they get the fundamentals of component-base architecture and internalize their mechanisms in a better way that previous students (who did not

employ *Rosette*) used to do it. On the other hand, *Rosette* has shown itself as a useful tool not just for teaching but for game development in general; some of the students are still using it for their capstone projects, even when component-base architecture introduction lectures ended more than a month ago.

9.8. Empirical evaluation of the automatic generation of component-based architecture

In this section we present the evaluation of the technique that automatically suggest a component distribution to implement a given declarative model of the entities in the game. Given a model of the entities, described in terms of their attributes and messages, the user may choose to manually identify a number of components to distribute such data and functionality or ask *Rosette* to do the work.

The goal of the work presented here is to empirically compare the quality of the component distributions designed by programmers using *Rosette* with those others automatically obtained by *Rosette* itself. Understanding game development as an iterative process, we measure quality of a component distribution in terms of coupling, cohesion, code duplication and modifiability and extensibility, as it evolves across several iterations of game development.

9.8.1. Methodology

The goal of the experiment that we carry out with *Rosette* was to answer the question "Does our component suggestion technique help to create better component distributions that improve the quality of the software?".

The experiment simulates the initial stages of a game development. Its design reflects the volatile nature of the development process: over five different stages, the subjects of the experiment are faced to the development of a small game where the complexity of its entities increases. If they fail to design a good component system in the first iterations, they will be forced to rewrite a great amount of code in the refactorization process, or they will end up with a messy set of components. This scenario allows us to measure whether the component suggestion does or does not aid in the process.

The experiment consists in the development of a side scrolling shootémup 2D arcade where the player controls a spacecraft, shooting enemies while avoiding collision of asteroids. However, as previously told, instead

of having the entire design from the very beginning, the game specification evolves over five iterations. The description and their expected result in terms of components follow:

1. The first version of the game had to show a spaceship flying from left to right with the camera moving at constant speed in the same direction. Some static asteroids also appear. A reasonable component design includes (1) a graphics component common to spaceship and asteroids for drawing themselves entities, (2) a controller component that captures movement events and moves the player accordingly, (3) a camera component that, given the entity position, updates the camera in the graphics engine and (4) a movement component that moves the owner entity at constant speed and that both player and camera use.

2. Next iteration consists in the creation of a random movement in the asteroids. That implies the development of a new component that, in some sense, plays the role of the simple artificial intelligence of asteroids that moves them randomly throughout the space.

3. Third iteration adds enemies in the form of other spaceships that appear on the right side of the screen and the ability of both player and enemies to shoot bullets. From the development point of view this forces to (1) the creation of new kinds of entities for enemies and bullets, (2) developing a component that manages the spawn of the bullets, which is used in both player and enemy entities, (3) reusing the simple AI component for the spaceships movement, (4) reusing the component that moves the owner entity at constant speed for the bullets and (5) modifying the *Controller* and *AI* component to emit the shoot messages when needed.

4. The fourth iteration asks for minimal collision detection in order for spaceships to react when bullets or asteroids collide against them. The expected solution is the addition of a physic/collision component and a life component that respond to the collision events.

5. In the last iteration an item is added to the scenario. The player is not able to shoot bullets until he picks this item. The development of this last gameplay mechanic is easy if the previously built components were well-thought. In particular, no other components are required, but only some modification to the existing ones.

For the experiment we gathered a group of 18 post-graduate students from our Master on Game Development at Complutense University of Madrid. At the moment the experiment took place, all of them

had taken master classes about game architecture, the pros and cons of the use of inheritance when developing the hierarchy of entities and component based architecture concepts.

The game had to be created in C++ and they had available a game skeleton that incorporates a minimal application framework that manages the main loop of the game, and a graphic server that encapsulates the inner workings of the Ogre render API[3]. It also had a small logic layer that manages component-based entities. The logic layer was able to load the description of the entities and the level map from external files. It, however, had no components, so students had the freedom to design the capabilities of every component from scratch. *Rosette* could aid in the creation of those components because it is able to generate the C++ code that is smoothly integrated within the logic layer provided to the students.

In order to evaluate the component suggestion technique of *Rosette*, we separated the students in two groups where every student developed the game individually. The *control group* used a restricted version of *Rosette* where the *Component Suggestion* module was not available and the *experimental group* was able to use *Rosette* unabridged.

9.8.2. Evaluation

As discussed earlier, the goal of this study is to analyze the impact of our component suggestion technique and whether it helps to create better component distributions. In order to compare the distributions of the control and experimental group we need an unbiased way of measuring their quality.

A *good component distribution* is characterized by the reusability of their components, its flexibility against changes, and the maintainability and extensibility of the resulting software. Reusability and flexibility increase when the *cohesion* of each component is maximized and, at the same time, the coupling between components (and with other classes) is minimized. Code duplication is detrimental to maintainability, and extensibility is inversely proportional to the number of changes needed to add a new feature.

In order to measure all these aspects in our experiment, we have used these tools:

- **Code and generated software:** we collected the *Rosette* domains and the code of each iteration. We used it for searching *cohesion*

[3] `http://www.ogre3d.org/`

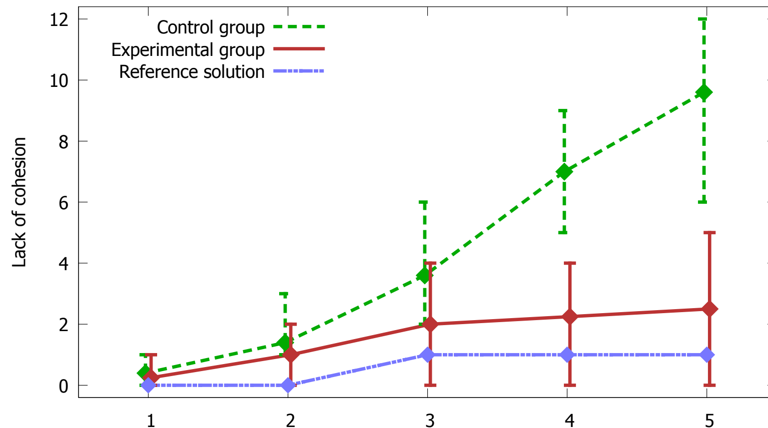

Figura 9.19: Lack of cohesion of the distributions per iteration

and *coupling* issues that state flexibility deficiencies, and also to look for code duplicities that damage maintainability in the studentsímplementations.

- **Traces:** we also collected traces from *Rosette* that reflects the use of the tool during the different iterations. We have used this information to analyze the effort needed to modify or extend the software created by the students.

- **Questionnaire:** we gave students a questionnaire at the end of the experiment to gauge the usefulness of the tool as they perceive it.

In the next subsections we will detail our findings about each one of these aspects while using *Rosette*.

9.8.2.1. Cohesion

From the programming point of view, cohesion refers to the degree to which the elements of a module (class or component) belong together (Yourdon y Constantine, 1979), for example we can measure the cohesion of the methods of a class. In order to evaluate the *lack* of cohesion of the software generated by the participants of the experiment we have used the *Lack of Cohesion in Methods (LCOM) measure*, described in Hitz y Montazeri (1995); Chidamber y Kemerer (1994) as "the number of pairs of methods operating on disjoint sets of instance variables, reduced by the number of method pairs acting on at least one shared instance variable". LCOM is applied to individual classes, and is higher when they lack cohesion.

Here we have a simplified example extracted from the code of one of the participants:

```
class Controller : IComponent {
private:
   float _speed;
   Entity* _bullet;
public:
   void VerticalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(_speed*secs,0);
      setPosition(pos);
   }
   void HorizontalMove(float secs) {
      Vector2 pos = getPosition();
      pos += Vector2(0,_speed*secs);
      setPosition(pos);
   }
   void Shoot() {
      EntityFactory f = EntityFactory::getInstance();
      Entity e = f->instanciate(bullet,getPosition());
      f->deferredDeleteEntity(e, 2.0f);
   }
}
```

In this code, the `Controller` component provides two different functionalities: movement and shooting ability. This constitutes a lack of cohesion, as comes into light when LCOM is calculated: although both methods (`VerticalMove`, `HorizontalMove`) use the `_speed` attribute, the pairs (`VerticalMove`, `Shoot`) and (`HorizontalMove`, `Shoot`) do not share anything at all. Thus, LCOM has a value of 1 $(= 2 - 1)$.

We have calculated the LCOM of every component of the software generated by the students in the five iterations and then we have added them up to illustrate the evolution of the cohesion during the game development. Figure 9.19 presents the average results of the two groups of participants where we can appreciate that in the first iterations the cohesion is quite similar between the two groups but, when the development evolves, the software generated by the students in the control group, without the help of *Rosette*, has much less cohesion. They usually have bigger components and, as their functionality is less united, they are less reusable, what means that these components are specific just for the game in current development (and even only for the current iteration). In order to test the statistical significance of the results, we calculated the P-value for the two groups in the last three iterations. The P-value is the probability, under the null hypothesis, of obtaining a result at least as extreme as the one that was actually observed (Good-

man, 1999). When the P-value is less than 0.05 (Stigler, 2008) indicates that the result would be unlikely under the null hypothesis. The P-value for the iterations 3, 4 and 5 were 0.076675, 0.0014788 and 0.00099428 what shows that the probability that the values of the two groups belongs to the same population decreases with the game development and in the last two iterations are significantly different (P-val < 0.05). Figure 9.19 also shows the LCOM of the solution, described above in the Experiment Section, that was used as reference of a good distribution. The Figure reveals that the experimental group, aided by *Rosette*, ended with a distribution with similar cohesion to that in the reference one.

9.8.2.2. Coupling

Coupling is the degree to which each program module relies on other modules and is usually contrasted with cohesion. Hitz y Montazeri (1995) define that "any evidence of a method of one object using methods or instance variables of another object constitutes coupling". They also propose a way to categorize the coupling between classes and object considering some rules; unfortunately they are not suitable for component-based architectures.

The definition given by Lee et al. (2001) fits better: "When a message is passed between objects, the objects are said to be coupled. Classes are coupled when methods declared in one class use methods or attributes of the other classes". The paper also proposes a *coupling measure* similar to that one we have used for the cohesion. They define the *coupling value* (Pi) as the *number of methods invoked* (different to the number of *method invocations*) from one class to another where *interaction coupling* is defined as:

$$Interaction_Coupling = \sum_{AllUseCase} Pi$$

Where each use case defines interactions between a component and the system to achieve a goal and they are extracted from the sequence diagrams.

Lee et al. also define *static coupling* which is caused by class association, composition or inheritance. They assign values to different relationships: 1 when two classes shares the same parent class, 2 when one class inherits from another, 3 when there is a directional association between two classes, 4 when the association is bidirectional and 5 when one class is an aggregation of another.

Component-based architectures add a new way of class coupling due to dynamic relationships. We consider that message passing between

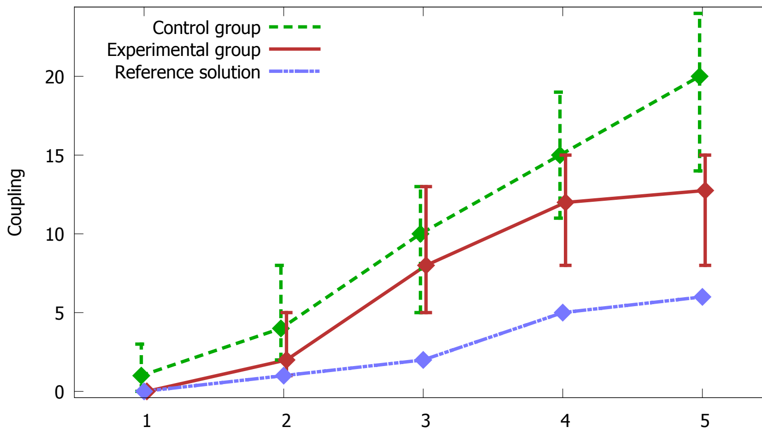

Figura 9.20: Coupling of the distributions per iteration

components constitutes a coupling of sibling classes, so are valued with 1. On the other hand, when a component accesses to the entity it belongs to is explicitly using the aggregation between them, so according with Lee et al. (2001) the coupling is 5.

Once *interaction* and *static coupling* are defined, Lee et al. create the *two classes coupling* as:

$$Two_Classes_Coupling = Interaction_Coupling * Static_Coupling$$

Using the next code extracted from one of the participants'solutions we will show an example of how $Two_Classes_Coupling$ is calculated:

```
class AI : IComponent {
public:
   void tick(unsigned int msecs) {
      float _hMove,_vMove;
      if(!_entity->getType().compare("MovingAsteroid")) {
         _hMove = rand() % 10 - 5;
         _vMove = rand() % 10 - 5;
      }
      else if(!_entity->getType().compare("Enemy")) {
         _hMove = rand() % 20 - 10;
         _vMove = rand() % 20 - 10;
      }
      IMessage m = new HorizontalMove(_hMove);
      sendMessage(m);
      m = new VerticalMove(_vMove);
      sendMessage(m);
   }
}
```

$Interaction_Coupling$ between `Entity` and `AI` classes has a value

of 1, because `AI` only invokes the `getType()` method and there is just one use case that involves both classes.

On the other hand, the `AI` component accesses the entity it belongs to (inherited `_entity` attribute), using the knowledge about the aggregation between entities and components, which constitutes a static coupling of 5. These two values become a $Two_Classes_Coupling$ of 5 (= 1 * 5).

`AI` sends `HorizontalMove` and `VerticalMove` messages that are processed by the `Controller` component previously presented. That shows a *static coupling* between both components (`AI` and `Controller`) valued as 1. They both interact in just one use case, but they exchange those two types of messages, so they have an $Interaction_Coupling$ of 2 that turns into the same value for $Two_Classes_Coupling$.

Using this procedure, we have carefully calculated for each participant the $Two_Classes_Coupling$ of each component with other classes, and we have added them up to obtain the *global coupling* of the component distribution. Figure 9.20 presents the results of the average of this global coupling in both groups during the five iterations. We can appreciate that the evolution of the coupling is similar in both groups and, from the statistical significance point of view, the iterations 3, 4 and 5 we have P-values of 0.34346, 0.12407 and 0.0015096. This shows a positive tendency but only in the last iteration the values between the two groups are significantly different. However, a closer look at the code has shown us that the reasons for this coupling are different. The coupling of the software generated by the experimental group is because the higher cohesion of their components prevents duplication in the functionality but makes the components communicate with each other. In fact coupling is usually contrasted with cohesion. For example, the `Controller` component of the previous example is used by the `AI` component but also by the player, which sends messages in the presence of keyboard events. In this way, the reusability of the `Controller` is good in this aspect although this slightly penalizes component coupling due to the message passing.

As components created by students in the control group were bigger and had more functionality (low cohesion) it would be expected that they had less coupling. However, those components, although big, were not usually self-contained. For example, some students implemented a component that mixed functionality of the `AI` and `Controller` components. This seems better from the coupling point of view but these implementations needed the *concrete entity type* (`Player`, `Enemy` or `Asteroid`) to decide how to behave, which creates unnecessary coupling.

gs to an *Asteroid*). In the same way,

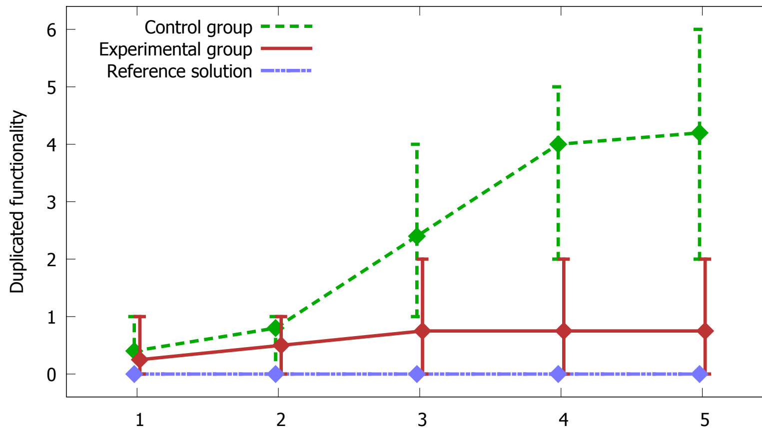

Figura 9.21: Duplicated functionality in the different iterations

Finally it was to be expected that our reference solution should have a bigger coupling since it was the solution with less cohesion. However, the other solutions have been very penalized by the *static coupling* because they invoke several times methods of the entity (generally to consult the entity type) and this multiplies by 5 the *interaction coupling*. So, although the number of messages exchanged by the components of our solution is bigger, the resulting coupling is smaller because it is considered less dangerous for reusability than accessing the entity type.

9.8.2.3. Duplication

Duplication of the functionality in the source code penalizes its maintainability. We have analysed the components created by the participants in the experiment looking for duplicate code and, when a functionality (i.e. a method) was replicated in two or more places, we penalized that domain. In this sense, the duplication measure reveals the number of pieces of code that have been duplicated and could have been encapsulated and reused by different methods.

Figure 9.21 presents the evolution of the duplicated functionality during the game development. It illustrates that participants, using the component suggestion of *Rosette*, maintained low code duplication during the whole development while people that did not use it duplicated more code each iteration. This figure confirms what cohesion and coupling measures revealed about the quality of the component distributions and proves that a bad distribution of the components leads to a bad reutilization of the functionality and to code duplication, which not only proves a poor reusability but also provokes a worse maintainability.

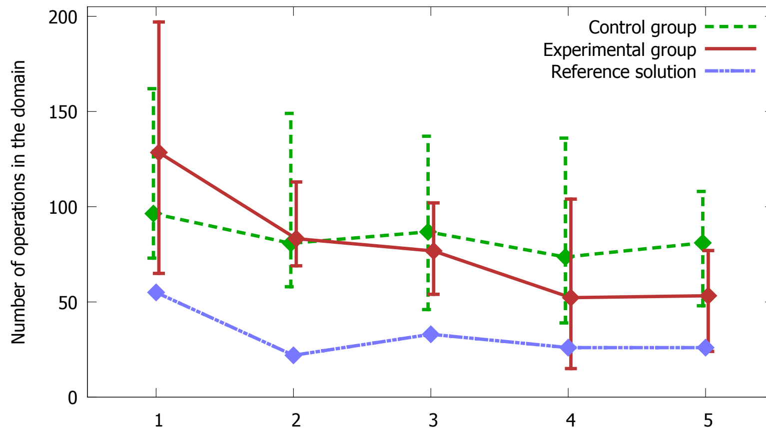

Figura 9.22: Number of operations done per iteration

The P-values in the iterations 3, 4 and 5 were 0.021433, 0.0017129 and 0.0012074 what prove that this results are not coincidence.

The big increment in the third iteration is mainly due to two reasons: (1) some participants were not aware of asteroids and enemy spaceships moving in the same way so, instead of reusing the functionality created in the second iteration (asteroids) they duplicated this code; (2) some of them also duplicate some code related with the shoot (both player and enemy) instead of parametrizing it in a new component.

During the fourth iteration duplicated code in the control group also increases significantly because many students did not group together the functionality related to collisions so they duplicated code in different components.

9.8.2.4. Modifiability and Extensibility

The last factor that should be taken into account to evaluate the quality of a component distribution is the modifiability and extensibility. A good measure of those aspects is the number of changes the component distribution suffers when a new functionality is added. We have used the traces generated by *Rosette* to collect this information. Those traces let us measure *semantic changes* in the domain, not just source code modifications that are less informative. *Rosette* tracked domain modifications such as components, attribute or messages creation, adding components to entities, attributes to components, elements removal, etc.

Paying attention to the average number of actions required to create the domains of the participants of both groups (Figure 9.22) we can observe that during the initial iterations, participants in the control group

needed to apply less operations to the game domain to obtain the proposed functionality, but in the last iterations, the number of changes increased contrasted with the people in the experimental group.

The bigger number of operation carried out in the first iteration can be explained because creating a better component distribution needs a more complex domain model. But it is important to note that when using the component suggestion of *Rosette* (experimental group), many of the domain operations *are automatic*, so a richer domain does *not* imply more human time. Users using component suggestions got a better initial distribution that let them to require fewer and fewer changes into the domain when the game evolved. On the other hand, users that did not receive assistance created a poorer distribution that needed a nearly constant number of changes in the next iterations.

In this case, the statistical significance of the results is not relevant for the first iterations. Results for iterations 3, 4 and 5 were 0.35490, 0.16899 and 0.011173 what shows that only in the last iteration the results of the two groups were significantly different but let us think that, if the development had continued, in further iterations the experimental group would require fewer operations than the control group.

9.8.3. Conclusions

Our intention with this experiment was proving that the component suggestion technique of *Rosette* helps to create a good distribution of the functionality in components and that the quality of the software generated is good enough in terms of flexibility, maintainability and reusability.

The data collected during the experiment reflects that, having two groups of people with the same skills in the development of videogames, those people that used the component suggestion module of *Rosette* ended with unbiased better software in every iteration. The key principle of maximizing the cohesion and minimizing the coupling is better accomplished by them and is also reinforced with the lack of code duplication. However, the main indication that this technique may improve and accelerate game development is that the number of changes the component distribution suffers when the requirements change is lower when following *Rosette*'s suggestions.

Nevertheless, the creation of an initial promising component distribution is expensive. Although the component suggestion of *Rosette* lighten this work, some students pointed out that this module is not intuitive, becoming hard to use. We could corroborate this fact because up to four

people in the experimental group left the experiment, when only two of the control group abandoned. We, then, plan to study how to flatten the *Rosette* learning curve. In addition, we need to further evaluate the effectiveness of the tool and techniques with bigger games and larger user groups.

9.9. Conclusions and future work

After studying the process of game development and how has it evolved over the years, we found that there is a lack of efficient communication mechanisms between the different roles involved in the development team (especially between designers and programmers). This mechanism is usually done by design documents, where all the details of the game (mechanics, gameplay, characters, items, interactions...) are described textually, which introduces ambiguity in many aspects and makes the communicative phase slow.

On the other hand, the used technology and methodologies have been adapted over the years allowing faster developments since, due to the changing nature of the design of video games, these technologies and methodologies should be as flexible as possible and allow changes in the project direction. However, this process of flexibility has a price and the component-based architecture, which is practically the choice taken by most commercial developments to manage the game logic, has its own problems. Despite it is a less static architecture than its predecessors, it is conceptually less intuitive since much semantic knowledge is hidden behind the data and the code. Moreover, due to its nature of dynamic composition, it introduces a large number of potential inconsistencies in the development process.

With this in mind we could recover the principal research question we want to answer with this work, which was formulated in the introduction:

> *Could a formal domain helps to alleviate these handicaps and improves the game development?*

The work proposed in this thesis shows a methodology that has as its cornerstone a formal domain that improves the process in several ways:

1. **Ease of domain representation:** The use of a specific visual modelling tool facilitates the work of both designers and programmers, simplifying the edition of the game model.

2. **Flexibility:** The formal domain allows that the definition of the game entities are modified in a simpler way than with the traditional *game design document*.

3. **Agility in communication:** Communication between designers and programmers is faster where information can be displayed in different layers of definition. The formal domain reduces the ambiguity introduced in natural language descriptions. Furthermore, the use of the formal domain by programmers allows them to recover the concept ontology introduced by inheritance based systems, and that is eliminated with the component-based architecture.

4. **Formal domain as contract of minimums:** Using different views of the same domain, depending on the role, designers and programmers can work on the same domain, so that domain can be seen as an agreement that is reached in common.

5. **Validation:** As we can reason reason over the domain it can be validated, so we can detect inconsistencies and correct errors in advance, saving debugging work in future phases. This technique not only overcomes the potential inconsistencies that are introduced with the component-based architecture, it also intercepts errors during the level edition.

6. **Suggested distribution of functionality:** From the definitions of the designers (and programmers) in a formal domain, we can identify optimal divisions of functionality in software components, accelerating the process of architectural design. This simplifies the creation of a more flexible software that will allow faster development in subsequent iterations.

7. **Iterative content generation:** The semantic information enables procedural content generation, so that development is guided by the formal model. This allows the generation of a very high percentage of the related game logic code and content for level design tools, very useful for designers. However, the biggest advantage is the possibility of high-level refactorings because, thanks to this procedural code generation, high-level refactoring are transferred directly to the code in a transparent manner to the programmer. Similarly, the changes are transferred to the rest of the game content.

8. **Improves the quality of the generated software:** The previous conceptual work helps to generate a reusable, flexible and, ultimately, higher quality software. In addition, the content generation

causes a more consistent and easier to read code. Finally, the possibility of doing high-level refactorings incite to do them if something was not well designed instead of *botching* the software.

9.9.1. Discussion of the results

The experiments presented in Sections 9.7 and 9.8 show first the suitability of the methodology for rapid prototyping. By using a tool such as *Rosette*, which supports the ideas developed in the thesis, it was possible to carry out the construction of small games iterations that would otherwise not have been possible due to time constraints. The ontological representation was gratefully appreciated by the users, who commented that it helped them to distributed entities in a way that was more mentally intuitive than simply using component lists. Moreover, the gap between architecture design and the concrete implementation had a positive influence on the distribution of functionality and therefore in the generated software.

In one of the experiments (Section 9.8) we measured how the automatic inference of software components positively affect to the quality of the distributions. In this experiment we empirically check whether this method increased the cohesion of the components, decreased the coupling and reduced duplication of functionality, but what really showed that the technique was useful for the development was that the modifications needed to make for adding a new feature to the game were lower with the use of the technique.

Furthermore, the experiment of Section 9.7, shows that the proposed methodology simplifies the understanding of the component-based architecture, a technique that at first is rejected by programmers. Although initially this methodology was not designed for educational purposes, a formal domain adds semantic information to the architecture that facilitates its learning and use. Moreover, thanks to the iteratively and procedural content generation, users were able to do semantic refactorings that allowed them to try out different configurations without effort, since the changes made ??in the distributions are automatically transmitted to the code.

However, we have not had the opportunity to test the suitability of the methodology in large developments with real designers and programmers. It would be interesting to see how it works with a larger development process with different roles, in a way that designers would make some domain to be established as a contract with programmers, those responsible of implementing the functionality.

9.9.2. Future work

We believe that the development methodology proposed in this thesis and *Rosette*, the tool that implements the ideas, provide a stable platform for further research that will improve the game development. To conclude this thesis we present a series of ideas or recommendations to continue this work in the future.

Starting from the semantic representation, the proposal presented in this work has focused on the description of the game entities. These entities are defined in terms of their attributes (state) and the messages they interprete (basically what actions they can perform and events that they catch). Even at the component level, where we can describe what messages are sent to the entity itself, requiring lower-level functionality, to carry out complex actions.

However, although it is essential to describe the elements of the game, it is not sufficient. Entities often interact between them in very different ways, but in its most basic form, they communicate by message passing.

It would be interesting to define in the formal domain how entities *interact* with other entities. With the semantic representation proposed in this thesis can define things like the *player* can *take damage* and therefore lose *life* or a *character* can *increase its health*. If the concept of interaction is introduced into the domain may involve different entities in the same process, describing, for example, that an *enemy* can *damage* a *player* or a *kit* can *increase* the *health* of a *character*.

The description of interactions between entities would be the first logical step to do, but not the only one. In the ontology we could keep adding more knowledge than now is in the *game design document*, modelling the different mechanical, rules, relationships between entities, etc.. which would allow better transmission of ideas between different members of the group.

This thesis has laid the basis for continue and for starting new branches in this field. We specified an ontological representation, a methodology that turns around it, specific techniques that take advantage of semantic knowledge and a tool that implements the functionality to empirically test the validity of the work. We hope a further progress in this direction in the coming years and that the use of formal domain go taking weight within the world of game development.

Bibliografía

Adams, E. *Fundamentals of Game Design*. New Riders Publishing, Thousand Oaks, CA, USA, 2nd edición, 2009. ISBN 0321643372, 9780321643377.

Araújo, M. y Roque, L. Modeling Games with Petri Nets. En *Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference* (editado por B. Atkins, H. Kennedy y T. Krzywinska). Brunel University, London, 2009.

Beck, K. Embracing change with extreme programming. *Computer*, vol. 32, páginas 70–77, 1999. ISSN 0018-9162.

Beck, K. y Andres, C. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN 0321278658.

Bilas, S. A data-driven game object system. En *Game Developer Conference*. 2002.

for Biomedical Informatics Research, S. C. Protégé. 2013. Disponible en `http://protege.stanford.edu/overview/` (último acceso, Noviembre, 2013).

Birkhoff, G. *Lattice Theory, third editon*. American Math. Society Coll. Publ. 25, Providence, R.I, 1973.

Buchanan, W. *Game Programming Gems 5*, capítulo A Generic Component Library. Charles River Media, 2005. ISBN 1-584-50352-1.

Chady, M. Theory and practice of game object component architecture. En *Game Developers Conference*. 2009.

Chandler, H. *The Game Production Handbook Second Edition (Computer Science Series)*. Infinity Science Press LLC 2009, Hingham, MA, USA, 2009. ISBN 978-1-934015-40-7.

Chidamber, S. R. y Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, vol. 20(6), páginas 476–493, 1994. ISSN 0098-5589.

Cook, D. The chemistry of game design. gamasutra. 2007. Disponible en `http://www.gamasutra.com/view/feature/1524/` (último acceso, Septiembre, 2013).

Deline, G. *Ontology-driven Methodology for Constructing Software Systems*. Phd, Athabasca University, 2006.

Dormans, J. *Engineering Emergence: Applied Theory for Game Design*. Universiteit van Amsterdam [Host], 2012. ISBN 9789461907523.

Flórez-Puga, G., Llansó, D., Gómez-Martín, M. A., Gómez-Martín, P. P., Díaz-Agudo, B. y González-Calero, P. A. Empowering designers with libraries of self-validated query-enabled behaviourtrees. En *Artificial Intelligence for Computer Games* (editado por P. A. González-Calero y M. A. Gómez-Martín), páginas 55–82. Springer, 2011.

Gaildrat, V. Declarative modelling of virtual environments, overview of issues and applications. Informe técnico, ToulouseIII University, IRIT, VORTEX Team, 2009.

Gamma, E., Helm, R., Johnson, R. E. y Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Massachusetts, USA, 1995.

Ganter, B. y Wille, R. Formal concept analysis. *Mathematical Foundations*, 1997.

Gao, Z., Ren, L., Qu, Y. y Ishida, T. Virtual space ontologies for scripting agents. En *Massively Multi-Agent Systems I* (editado por T. Ishida, L. Gasser y H. Nakashima), vol. 3446 de *Lecture Notes in Computer Science*, páginas 70–85. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-26974-8.

Garcés, S. *AI Game Programming Wisdom III*, capítulo Flexible Object-Composition Architecture. Charles River Media, 2006. ISBN 1-584-50457-9.

Ghallab, M., Nau, D. y Traverso, P. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004. ISBN 978-1558608566.

Godin, R. y Valtchev, P. *Formal Concept Analysis*, capítulo Formal Concept Analysis-Based Class Hierarchy Design in Object-Oriented Software Development, páginas 304–323. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-27891-7.

Gómez-Martín, P. P., Gómez-Martín, M. A., González-Calero, P. A. y Palmier-Campos, P. Using metaphors in game-based education. En *Technologies for E-Learning and Digital Entertainment. Second International Conference of E-Learning and Games (Edutainment'07)* (editado por K. chuen Hui, Z. Pan, R. C. kit Chung, C. C. Wang, X. Jin, S. Göbel y E. C.-L. Li), vol. 4469 de *Lecture Notes in Computer Science*, páginas 477–488. Springer Verlag, 2007. ISBN 3-540-73010-9.

Goodman, S. N. Toward evidence-based medical statistics. 1: The p value fallacy. *Annals of Internal Medicine*, vol. 130(12), páginas 995–1004, 1999.

Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition*, vol. 5, páginas 199–220, 1993. ISSN 1042-8143.

Hesse, W. y Tilley, T. A. *Formal Concept Analysis used for Software Analysis and Modelling*, vol. 3626 de *LNAI*, páginas 288–303. Springer, 2005.

Hitz, M. y Montazeri, B. Measuring coupling and cohesion in object-oriented systems. En *Proc. Intl. Sym. on Applied Corporate Computing*. 1995.

Horridge, M. y Patel-schneider, P. F. P.f.: Manchester syntax for owl 1.1. En *In: OWLED 2008, 4th international workshop OWL: Experiences and Directions (2008) Live Extraction 1223*. 2008.

Ibáñez, J. y Delgado-Mata, C. A basic semantic common level for virtual environments. *IJVR*, vol. 5(3), páginas 25–32, 2006.

Ibáñez-Martínez, J. y Mata, D. Virtual environments and semantics. *European Journal for the Informatics Professional*, vol. 7(2), página 16, 2006.

Katharine, N. Game design tools: Time to evaluate. En *Proceedings of 2012 DiGRA Nordic*. University of Tampere, 2012.

Keith, C. *Agile Game Development with Scrum*. Addison-Wesley Professional, 1st edición, 2010. ISBN 0321618521, 9780321618528.

Lakos, J. *Large Scale C++ Software Design*. Addison Wesley, 1996. ISBN 0-201-63362-0.

Lee, J. K., Seung, S. J., Kim, S. D., Hyun, W. y Han, D. H. Component identification method with coupling and cohesion. En *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, APSEC '01, páginas 79–. IEEE Computer Society, Washington, DC, USA, 2001.

Librande, S. One-page designs. presentation at the game developers conference, san francisco ca. 2010. Disponible en `http://stonetronix.com/gdc-2010/` (último acceso, Septiembre, 2013).

Lifschitz, V. What is answer set programming?. En *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. 2008.

Llansó, D., Gómez-Martín, M. A., Gómez-Martín, P. P. y González-Calero, P. A. Explicit domain modelling in video games. En *International Conference on the Foundations of Digital Games (FDG)*. ACM, Bordeaux, France, 2011a.

Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. *Reflective components for designing behaviour in video games*. Proyecto Fin de Carrera, Universidad Complutense de Madrid, Madrid, Spain, 2009a.

Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. Self-validated behaviour trees through reflective components. En *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (editado por C. Darken y G. M. Youngblood). The AAAI Press, Stanford, California, USA, 2009b. ISBN 978-1-57735-431-4.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A. y González-Calero, P. A. Iterative software design of computer games through FCA. En *Procs. of the 8th International Conference on Concept Lattices and their Applications (CLA)*. INRIA Nancy, Nancy, France, 2011b.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A. y González-Calero, P. A. Knowledge guided development of videogames. En *Papers from the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process (IDP)*. AIII Press, Palo Alto, California, USA, 2011c.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A. y González-Calero, P. A. A declarative domain model can serve as design document. En *Artificial Intelligence in the Game Design Process 2 (IDP)*, páginas 9–15. AIII Press, 2013a. ISBN 978-1-57735-635-6.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A. y González-Calero, P. A. Domain modeling as a contract between game designers and programmers. En *Actas del Primer Simposio Español de Entretenimiento Digital* (editado por P. A. González-Calero y M. A. Gómez-Martín), páginas 13–24. 2013b. ISBN 978-84-695-8350-0.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A. y González-Calero, P. A. Empirical evaluation of the automatic generation of a component-based software architecture for games. En *Proceedings, The Ninth AAAI Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, páginas 37 – 43. AIII Press, 2013c. ISBN 978-1-57735-607-3.

Llansó, D., Gómez-Martín, P. P., Gómez-Martín, M. A., González-Calero, P. A. y El-Nas, M. S. Tool-supported iterative learning of component-based software architecture for games. En *International Conference on the Foundations of Digital Games (FDG)*, páginas 376–379. 2013d.

Long, S. Online product development management: Methods and madness. presentation at the game developers conference, san francisco ca. 2003.

Machado, A., Amaral, F. N. y Clua, E. W. G. A trivial study case of the application of ontologies in electronic games. En *Actas Videojogos 2009: Conferência de Ciências e Artes dos Videojogos*. 2009.

Nelson, M. J. y Mateas, M. A requirements analysis for videogame design support tools. En *FDG* (editado por J. Whitehead y R. M. Young), páginas 137–144. ACM, 2009. ISBN 978-1-60558-437-9.

Pastor, O. y Molina, J. C. *Model-Driven Architecture in Practice. A Software Production Environment Based on Conceptual Modeling*. Springer Verlag, 2007.

Peters, C., Dobbyn, S., MacNamee, B. y O'Sullivan, C. Smart objects for attentive agents. En *WSCG*. 2003.

Rabin, S. *Introduction to Game Development*. Charles River Media/-Course Technology, Cengage Learning, 2010. ISBN 9781584507093.

Rene, B. *Game Programming Gems 5*, capítulo Component Based Object Management. Charles River Media, 2005. ISBN 1-584-50352-1.

Rogers, S. *Level Up!: The Guide to Great Video Game Design*. Wiley, 2010. ISBN 9780470688670.

Ryan, T. The anatomy of a design document. gamasutra. 1999. Disponible en http://www.gamasutra.com/view/feature/3384/the_anatomy_of_a_design_document_.php (último acceso, Septiembre, 2013).

Salen, K. y Zimmerman, E. *Rules of Play: Game Design Fundamentals*. The MIT Press, 2003. ISBN 0262240459, 9780262240451.

Sánchez-Ruiz, A. A., Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. Authoring behaviour for characters in games reusing abstracted plan traces. En *Intelligent Virtual Agents* (editado por Z. Ruttkay, M. Kipp, A. Nijholt y H. Vilhjálmsson), vol. 5773 de *Lecture Notes in Computer Science*, páginas 56–62. Springer Berlin Heidelberg, 2009a. ISBN 978-3-642-04379-6.

Sánchez-Ruiz, A. A., Llansó, D., Gómez-Martín, M. A. y González-Calero, P. A. Authoring behaviours for game characters reusing automatically generated abstract cases. En *ICCBR Workshop* (editado por S. J. Delany), páginas 129–137. 2009b.

Schwaber, K. y Beedle, M. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edición, 2001. ISBN 0130676349.

Seedorf, S., Informatik, F. F. y Mannheim, U. Applications of ontologies in software engineering. En *In 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), held at the 5th International Semantic Web Conference (ISWC 2006)*. 2006.

Sheffield, B. Defining games: Raph koster's game grammar. gamasutra. 2007. Disponible en http://www.gamasutra.com/view/feature/130395/defining_games_raph_kosters_game_.php (último acceso, Septiembre, 2013).

Sicart, M. Defining game mechanics. *Game Studies*, vol. 8(2), 2008.

Silvano-Orita-Almeida, M. y Soares-Correa-da Silva, F. A systematic review of game design methods and tools. En *Entertainment Computing - ICEC 2013* (editado por J. Anacleto, E. Clua, F. Silva, S. Fels y H. Yang), vol. 8215 de *Lecture Notes in Computer Science*, páginas 17–29. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-41105-2.

Smelik, R. *A Declarative Approach to Procedural Modelling of Virtual Worlds*. Phd, Technische Universiteit Delft, 2011.

Smith, A. M., Andersen, E., Mateas, M. y Popovic, Z. A case study of expressively constrainable level design automation tools for a puzzle game. En *FDG* (editado por M. S. El-Nasr, M. Consalvo y S. K. Feiner), páginas 156–163. ACM, 2012. ISBN 978-1-4503-1333-9.

Smith, A. M. y Mateas, M. Answer set programming for procedural content generation: A design space approach. *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 3(3), páginas 187–200, 2011.

Stigler, S. Fisher and the 5 *CHANCE*, vol. 21(4), páginas 12–12, 2008. ISSN 0933-2480.

Stumme, G. Efficient data mining based on formal concept analysis. En *Database and Expert Systems Applications, 13th International Conference* (editado por A. Hameurlain, R. Cicchetti y R. Traunmüller), vol. 2453 de *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-44126-3.

Sweeney, T. The next mainstream programming language: a game developer's perspective. En *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'06)*, página 269. ACM, New York, NY, USA, 2006. ISBN 1-59593-027-2.

Swift, K., Wolpaw, E. y Barnett, J. Thinking with portals. *Game Developer*, páginas 7–12, 2008.

Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997. ISBN 0201178885.

Tetlow, P., Pan, J., Oberle, D., Wallace, E., Uschold, M. y Kendall, E. Ontology driven architectures and potential uses of the semantic web in software engineering. 2006. W3C, Semantic Web Best Practices and Deployment Working Group, Draft.

Tutenel, T. *Semantic Game Worlds*. Phd, Technische Universiteit Delft, 2012.

Tutenel, T., Bidarra, R., Smelik, R. M. y de Kraker, K. J. Rule-based layout solving and its application to procedural interior generation. *Proceedings of the CASA Workshop on 3D Advanced Media in Gaming and Simulation (3AMIGAS'09)*, 2009a.

Tutenel, T., Bidarra, R., Smelik, R. M. y Kraker, K. J. D. The role of semantics in games and simulations. *Computers in Entertainment (CIE)*, vol. 6(4), página 57, 2008.

Tutenel, T., Smelik, R., Bidarra, R. y Jan De Kraker, K. Using semantics to improve the design of game worlds. *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE'09)*, páginas 14–16, 2009b.

Tutenel, T., Smelik, R. M., Bidarra, R. y de Kraker, K. J. A semantic scene description language for procedural layout solving problems. En *Proceedings of AIIDE 2010 - 6th Conference on Artificial Intelligence and Interactive Digital Entertainment, 11-13 October, Stanford, CA, USA*. 2010.

Tutenel, T., Smelik, R. M., Lopes, R., de Kraker, K. J. y Bidarra, R. Generating consistent buildings: A semantic approach for integrating procedural techniques. *IEEE Trans. Comput. Intellig. and AI in Games*, vol. 3(3), páginas 274–288, 2011.

Valtchev, P., Grosser, D., Roume, C. y Hacene, M. R. Galicia: An open platform for lattices. En *In Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03*, páginas 241–254. Shaker Verlag, 2003.

W3C. Owl 2 web ontology language document overview (second edition). 2012. Disponible en `http://www.w3.org/TR/owl2-overview/` (último acceso, Noviembre, 2013).

West, M. Evolve your hiearchy. *Game Developer*, vol. 13(3), páginas 51–54, 2006.

Wille, R. *Restructuring Lattice Theory: an approach based on hierarchies of concepts*. Ordered Sets, 1982.

Youngblood, M., Heckel, F. W., Hale, D. H. y Dixit, P. N. Embedding information into game worlds to improve interactive intelligence. En *Artificial Intelligence for Computer Games* (editado por P. A. González-Calero y M. A. Gómez-Martín). Springer, 2011. ISBN 978-1-4419-8188-2.

Yourdon, E. y Constantine, L. L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edición, 1979. ISBN 0138544719.

Zagal, J. P. y Bruckman, A. The game ontology project: supporting learning while contributing authentically to game studies. En *Proceedings of the 8th international conference on International conference for the*

learning sciences - Volume 2, ICLS'08, páginas 499–506. International Society of the Learning Sciences, 2008.

Zagal, J. P., Mateas, M., Fernández-vara, C., Hochhalter, B. y Lichti, N. Towards an ontological language for game analysis. En *in Proceedings of International DiGRA Conference*, páginas 3–14. 2005.

Zhang, X., Tutenel, T., Mo, R., Bidarra, R. y Bronsvoort, W. F. A method for specifying semantics of large sets of 3d models. En *GRAPP/IVAPP*, páginas 97–106. 2012.

Zhu, X. y Jin, Z. Ontology-based inconsistency management of software requirements specifications. En *SOFSEM 2005: Theory and Practice of Computer Science* (editado por P. Vojtáŝ, M. Bieliková, B. Charron-Bost y O. Sýkora), vol. 3381 de *Lecture Notes in Computer Science*, páginas 340–349. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-30577-4_37.