DISEÑO DE EJEMPLOS DOCENTES PARA EL APRENDIZAJE EFICAZ DE METODOLOGÍAS ORIENTADAS A AGENTES

IGNACIO MOYA SEÑAS

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Sistemas Inteligentes

Curso 2012 - 2013

Convocatoria de Junio

Calificación: SOBRESALIENTE (10)

Director

JORGE JESÚS GÓMEZ SANZ

Autorización de difusión

Ignacio Moya Señas

12 de julio de 2013

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "DISEÑO DE EJEMPLOS DOCENTES PARA EL APRENDIZAJE EFICAZ DE METODOLOGÍAS ORIENTADAS A AGENTES", realizado durante el curso académico 2012-2013 bajo la dirección de Jorge Jesús Gómez Sanz en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

Los sistemas basados en agentes resultan interesantes para la construcción de sistemas distribuidos. Sin embargo, estos sistemas son difíciles de construir y por eso se concibe la Ingeniería del Software Orientada a Agentes (AOSE). De cualquier forma, las metodologías existentes para el desarrollo de los sistemas basados en agentes no contemplan las dificultades de su aprendizaje, lo cual es un impedimento para nuevos desarrolladores. Aprovechando los paralelismos existentes entre AOSE y la ingeniería del software, se propone investigar cómo se plantea la docencia en esta disciplina, con el fin de trasladarlos al dominio de AOSE. Esta investigación encuentra que los ejemplos educativos son un recurso básico para mejorar la educación en la ingeniería del software. Sin embargo, no se ha encontrado ningún trabajo que defina cómo deben de ser estos ejemplos. Durante este trabajo, se identifican algunas de las características deseables para que un ejemplo educativo sea efectivo. Posteriormente se proponen unos ejemplos para su evaluación según las características identificadas.

Palabras clave

Agentes inteligentes, Agentes software, Ingeniería del Software Orientada a Agentes, AOSE, Educación en Ingeniería del Software, INGENIAS, Metodologías orientadas a agentes.

Abstract

Agent-based systems are interesting for building distributed systems. Nevertheless, this systems are hard to build, and the methods available from Software Engineering are not suited nor effective. Instead, Agent Oriented Software Engineering (AOSE) is conceived. Anyway, existent methodologies for building agent-based systems do not address the training problem, thus they are hard to learn. Taking advantage from existing parallelism between AOSE and Software Engineering, its proposed to study existing solutions that facilitate Software Engineering adoption with the goal of transferring them to AOSE methodologies. This research finds out that tailored examples are useful for improving software engineering education. Nevertheless, no work has been found that deals with the features those examples should possess. During this research, some features are identified that may make an example more effective. Later some examples are proposed that evaluate effectiveness of the identified features.

Keywords

Intelligent agents, Software agents, Agent-Oriented Software Engineering, AOSE, Software Engineering education, INGENIAS, Agent-oriented methodologies.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El desarrollo de sistemas distribuidos es costoso. Cuando se persigue que, además, sean autogestionados o demuestren algún tipo de inteligencia, el desarrollo pasa a otro nivel de complejidad. En la investigación en agentes software, se trabaja con la hipótesis de que tales sistemas se pueden construir de forma más eficaz sobre el concepto de agente software. Esto da lugar a los Sistemas Multi-Agente, que pueden exhibir características como las antes mencionadas, si se siguen las técnicas y métodos que ha identificado la investigación. Estos intentos por construir métodos diferentes de los tradicionales para crear Sistemas Multi-Agente de forma disciplinada, engloban lo que se ha conocido como *Ingeniería del Software orientada a Agentes* (*Agent Oriented Software Engineering* o AOSE).

El desarrollo de Sistemas Multi-Agente, hasta ahora, ha planteado soluciones que divergen de los métodos clásicos, entendiendo como tales los que propone la Ingeniería del Software (IS). Irónicamente, el desarrollo de estos sistemas no ha incorporado técnicas de ingeniería del software, o *Software Engineering* (SE), porque los investigadores entendían que no iban a ser suficientes ni efectivas [Wooldridge et al., 2000]. Sin embargo, la experiencia dice que, siguiendo estos preceptos, resulta que esta clase de sistemas siguen siendo difíciles de construir [Sterling and Taveter, 2009].

Al igual que la ingeniería del software tradicional, el éxito en la aplicación de la ingeniería del software orientada a agentes está relacionada con la correcta aplicación de una metodología y un proceso de desarrollo. Esta correcta aplicación requiere que la metodología empleada se haya aprendido apropiadamente. Sin embargo, si se repasan los contenidos de la principal escuela de verano de agentes, la *European Agent Systems Summer School* [EASS, 2000, 2008, 2009, 2010, 2011, 2012] , se encuentran pocas referencias a las metodologías orientadas a agentes y a su aprendizaje. La literatura especializada no es diferente, por lo que, más que hablar de metodologías o tecnologías poco efectivas, quizás haya que hablar de fallos más fundamentales en la transmisión de conocimientos a los que deben aplicar AOSE. En algunos casos como la metodología Prometheus, se reconocen las dificultades del alumno para aprender a construir este tipo de sistemas, pero los autores se limitan a afirmar que fueron conscientes de ellas en el momento de concepción de la metodología y a apuntar que su metodología ha supuesto una mejora de la situación [Padgham and Winikoff, 2005]. Tampoco dan detalles acerca de las posibles dificultades encontradas en el aprendizaje de

su propia metodología. Este hecho puede deberse a que la ingeniería del software orientada a agentes es un paradigma relativamente joven, que aún no ha conseguido tener relevancia real dentro del mundo de la industria [Gómez-Sanz and Fuentes-Fernández, 2013].

Existe un paralelismo obvio entre los problemas de formación en AOSE y la Ingeniería del Software, por lo que parece natural estudiar qué se ha hecho en la Ingeniería del Software y aprender de ello. Por ello, se plantea primero investigar cómo se enseña ingeniería del software dentro de los currículos docentes e identificar similitudes trasladables. Esta investigación se resuelve en cuatro grandes grupos de conclusiones:

- La educación en la ingeniería del software debe dar tratar de dar solución a los problemas que puede encontrar el alumno en cualquiera de las fases de un desarrollo. Por ejemplo, el alumno que se inicia en la ingeniería del software, a menudo, encuentra serias dificultades en la captura de requisitos de un sistema al definir un diagrama de casos de uso [Halling et al., 2002; Pucher and Lehner, 2011]. Frecuentemente identificar los casos de uso relevante de un sistema no será trivial para el alumnado, y corre el riesgo de cometer errores graves, producidos en parte por la sensación de ridículo que supone preguntar aparentemente obvio.

- Aprender la práctica de la Ingenieriería del Software es fundamental. Durante la revisión de la educación en la ingeniería del software, resulta evidente la importancia de los ejemplos dentro del proceso educativo. Los ejemplos son valorados positivamente tanto por alumnos como docentes y no reciben la atención suficiente [Liu et al., 2009; Shaw, 2000]. Sin embargo, cualquier ejemplo no vale.Se debe apostar por la efectividad de estos ejemplos. De esta forma, identificar las características que debe poseer un ejemplo para ser efectivo será una tarea principal. También, y como medio para mejorar la calidad y eficacia en los ejemplos educativos, se propone la aplicación de teorías de aprendizaje. El alcance de los ejemplos es importante también. Algunos autores defienden ejemplos simples autocontenidos y muy concretos. Otros abogan por prácticas de más alcance, como proyectos enteros.

- Las teorías de aprendizaje son relevantes para aumentar la efectividad de la docencia en AOSE.Dentro de los modelos y procesos de aprendizaje se resalta la teoría de carga cognitiva (Cognitive load theory), que es el fruto de trabajo de investigación tras una mejora en la efectividad de la asimilación de información compleja [Pollock et al., 2002].

- Agent Oriented Programming vs AOSE, o la programación vs la ingeniería. Un estudio preliminar muestra una primera disyuntiva respecto de la relevancia de la implementación frente al diseño. En el caso de los agentes, esto se refiere a las técnicas de diseño versus lo que se conoce como Agent Oriented Programing (AOP). La problemática está íntimamente relacionada con la discusión que lleva a diferenciar Computer Science de Software Engineering. Por ello la educación de AOP no es indiferente de la educación en AOSE, dado que en numerosas ocasiones la implementación de sistemas y su traslación desde la especificación hasta el código, se produce de manera manual

[Nunes et al., 2011]. De hecho,se podría decir que AOP es parte de AOSE, pero se hace necesario entender mejor su lugar.

Estos aspectos llevan a concluir que formarse en AOSE, implica no sólo conocer la teoría, sino, también, cómo aplicarla. Y formarse en la correcta aplicación de los principios no es una tarea fácil, pero se ve aliviada si se disponen de ejemplos escalables coherentes y entendibles. Sin embargo, no está claro si existen ejemplos reutilizables entre las diferentes propuestas metodológicas en AOSE. La contribución de este trabajo consiste en avanzar en esta línea identificando características deseables de estos ejemplos y validando estas características con ejemplos prototípicos. Se usa como hipótesis de trabajo que poseer estas características hace un ejemplo más efectivo. La metodología que se elige como objetivo de los ejemplos es INGENIAS. Se puede avanzar que se elige INGENIAS porque ofrece generación automática de código (que permite obviar el debate antes mencionado sobre la implementación) y amplia cobertura del ciclo de vida del proyecto.

El resto del texto prosigue de la siguiente forma. Primero se revisa elestado del arte de la educación en la ingeniería del software y de aquellos elementos que son relevantes para un aprendizaje efectivo. Después, se presenta la contribución en forma de las características principales identificadas en los ejemplos educativos. Más tarde se presenta la aplicación de las características para la evaluación de ejemplos educativos. Finalmente, se presentan las conclusiones y el trabajo futuro.

Capítulo 2

Estado del arte

El estado del arte revisa cinco grandes aspectos relacionados con la docencia de la Ingeniería del Software y su relación con AOSE:

- Comenzar repasando metodologías relevantes en la Ingeniería del Software orientada a Agentes, con el fin de entender la complejidad de las metodologías de AOSE y cómo se presentan.
- Educación en Ingeniería del Software. Revisar los trabajos dedicados a especificar qué contenidos se deben incluir durante la educación en IS que se puedan utilizar para educación en AOSE.
- Estudiar cómo se aplica SE en las clases y qué aproximaciones existen, además de las metodologías contempladas, para ser utilizadas en las clases y la forma de articular estas últimas.
- Repasar teorías de aprendizaje y trabajos que indiquen espacio de mejora dentro de la ingeniería del software y en el diseño de ejemplos educativos, buscando su aplicación a AOSE.
- Programación en Ingeniería del Software. Reunir trabajos referidos al aprendizaje de programación, como un elemento que frecuentemente forma parte de la práctica de la ingeniería del software y revisar alternativas AOP para AOSE.

Este método se propone porque se puede considerar importante contar con un trasfondo de los fundamentos que caracterizan el campo y los planes de estudios que se implementan para cubrir la demanda que existe. También parece adecuado revisar el trabajo dedicado a definir cómo se debe enseñar la ingeniería del software y qué aproximaciones existen, además de las metodologías contempladas para ser utilizadas en las clases y la forma de articular estas últimas. Este tipo de trabajos pueden ser de interés para razonar acerca de cómo enseñar AOSE.

2.1. Metodologías orientadas a agentes

El término *Agent Oriented Software Engineering* (AOSE), se lleva acuñando desde finales del siglo XX. Ya por entonces, las características de los sistemas complejos llevaban a señalar a los sistemas basados en agentes como el futuro de estos sistemas, probablemente distribuidos, donde la interacción supone un rasgo característico [Jennings, 1999; Wooldridge and Ciancarini, 2001]. De esta forma, resultan necesarias técnicas específicas para analizar, diseñar y evaluar este tipo de sistemas. AOSE supone la aplicación de SE donde los conceptos y abstracciones principales son los de los agentes software.

La falta de herramientas adecuadas para AOSE ha llevado a la creación de gran variedad de metodologías. Una metodología orientada a agentes consiste en un sistema de métodos orientados a agentes. Un método en el contexto de IS son elementos que estructuran y guían las actividades concernientes a IS, con el fin de hacerlas sistemáticas [Gómez-Sanz and Fuentes-Fernández, 2013]. De este modo, un método AOSE supondrá un método IS donde la abstracción principal es el agente software.

Existen diversas metodologías AOSE que se consideran bien establecidas [Gómez-Sanz and Fuentes-Fernández, 2013], como por ejemplo: Gaia, MaSE, MESSAGE, Tropos, SODA, PASSI, Prometheus o INGENIAS.

A continuación se revisan cinco de las metodologías mencionadas. El repaso a las metodologías consistirá en describir sus distintas fases, modelos y diagramas, mencionando herramientas de soporte y referencias a la educación en caso de haberlas.

2.1.1. Gaia

Gaia [Wooldridge et al., 2000; Zambonelli et al., 2005], es una metodología que no ha evolucionado mucho a lo largo del tiempo, dado que tan sólo se han elaborado dos versiones. La versión actual, se basa en la metáfora organizacional para modelar los sistemas según las siguientes abstracciones: entorno, rol, interacción, reglas y estructuras de la organización. A partir de estas abstracciones se construyen los modelos considerados en las etapas de la metodología.

Gaia se compone de dos fases: análisis y diseño, aunque la etapa de diseño suele dividirse a su vez en dos: diseño arquitectónico y diseño detallado. En consecuencia, la metodología no cubre la especificación de requisitos ni la fase de implementación.

Durante la fase de análisis, se contemplan las posibles suborganizaciones que puedan contemplarse en la organización del sistema que pretendemos diseñar, se compone el modelo de entorno, y se elaboran los modelos preliminares de roles e interacciones, a los que se añade la definición de las reglas de la organización.

La fase de diseño, en concreto el diseño arquitectónico, comienza por definir la estructura de la organización, lo que supone estructurar los roles teniendo en cuenta los modelos anteriores.Gaia propone directivas para la elección de la topología (en lo referente a jerarquías) y el régimen de control dentro de la organización (que diseña las interacciones dentro de ella), donde las reglas de la organización resultan determinantes.

La definición de la estructura de la organización resulta muy apropiada para la aplicación de patrones, algo que se contempla de forma explícita en Gaia.

También como parte de la estructura de la organización, se considera la representación gráfica de las relaciones (control, dependencia o igualdad) entre los diferentes roles, aunque no propone una notación específica (como pudiera ser la representación mediante una ontología). A ojos de Gaia, una representación suficiente podría proponer la utilización de un grafo donde los nodos representan los roles y las aristas representan relaciones.

Como parte del diseño arquitectónico, también se considera la elaboración de los modelos completos de roles e interacciones. Completar los roles significa definir todas las actividades y responsabilidades de un rol, añadiendo al conjunto los llamados roles organizacionales, que aparecen como consecuencia de la elección de una determinada estructura para la organización. Para completar las interacciones es necesario completar la definición de sus protocolos, además de resolver la aparición de los protocolos organizacionales, que se adoptan como consecuencia de la estructura elegida.

Finalmente, la última fase de la metodología, el diseño detallado supone la elaboración del modelo de agente y de servicios. El modelo de agentes define las clases de agentes contempladas dentro de la organización, qué roles podrán jugar y qué número de instancias de esa clase podrán encontrarse en el sistema. De esta forma, en el modelo de agente podrán representarse simplemente con los roles que puede interpretar cada tipo de agente (sin restricciones respecto al estilo, pudiendo ser una tabla). La idea de agentes de Gaia corresponde a entidades software de grano grueso, considerando que un agente representará un pequeño número de roles, habitualmente sólo uno.

El modelo de servicios contempla la funcionalidad que se asigna a cada rol, definida mediante distintos servicios. La definición de un servicio supone la identificación de su entrada/salida, precondición y poscondición, información que se obtiene a partir de los modelos de entorno, rol e interacción; siendo habitual la aparición de nuevos servicios referidos al ciclo de vida del agente. Gaia no considera una notación específica ni impone restricciones en cuanto a la implementación de servicios.

La primera versión de Gaia consideraba un modelo adicional, el modelo de "acquintances". La función de este modelo era representar las comunicaciones que se consideraban entre las clases de agente. A modo de representación, se consideraba suficiente la utilización de un grafo dirigido donde los nodos son tipos de agente y las aristas representan las comunicaciones.

Se observa que los modelos de Gaia no tienen como objetivo ajustarse a ningún metamodelo específico, utilizando en su lugar una notación bastante informal, incluso dejándola a gusto del diseñador. Se puede decir que resultaría esperable la existencia de mayor rigor por parte de una metodología.

Una característica negativa de Gaia es que, en cuanto al proceso de desarrollo, la propuesta de Gaia es secuencial. Este aspecto nos permite hacer una correspondencia con metodologías de desarrollo no orientado a agentes y usados comúnmente, como el Waterfall model. Esta comparación no es algo positivo, dado que este tipo de metodologías se considera superado a los ojos de la mayoría de la comunidad de la ingeniería del software.

Otra característica negativa de Gaia es no dar ningún soporte a la implementación. Sin

embargo, esta situación no es algo casual, dado que el enfoque de la metodología es ser universal e independiente de las tecnologías destino (plataforma de agentes, framework) con el fin de ser adecuada para un número mayor de sistemas. Por ello propone un diseño conceptual con mayor libertad, aunque esta libertad corre riesgo de ser perjudicial para el producto final, dado que al carecer de herramientas de soporte y no contemplar la implementación, inevitablemente será propenso a contener errores.

Respecto a la enseñanza de Gaia, no se ha encontrado referencias.

2.1.2. Tropos

Tropos [Bresciani et al., 2004; Giorgini et al., 2005] es una metodología de desarrollo orientada a agentes con larga trayectoria. La principal característica que le identifica respecto de otras alternativas, consiste en un mayor énfasis dirigido a la identificación y análisis de requisitos.

El desarrollo siguiendo Tropos se compone de cinco fases: análisis de requisitos iniciales, análisis de requisitos tardíos, diseño arquitectónico, diseño detallado, y finalmente la fase de implementación. Respecto a la fase de implementación, Tropos inicialmente no la contemplaba como una fase de su desarrollo, limitándose a proponer una serie de directrices con el fin de facilitar la misma. Por esta razón, será habitual que en otros escritos de Tropos que se encuentran se hable de cuatro fases y no de cinco.

Durante el análisis requisitos iniciales, se busca identificar el ámbito en el cual el sistema debe funcionar, en términos de qué interesados relevantes se consideran dentro del dominio y qué relaciones existen entre ellos. Para hacer esto, Tropos considera dos diagramas: de actores y de justificación (del inglés "rationale"). El diagrama de actores, plasma a los interesados identificados (que serán los actores) con sus relaciones (que se muestran en términos de dependencias). La representación utilizada consiste en un grafo dirigido donde los nodos son actores, considerando el nodo origen como el actor dependiente y el nodo destino como el dependido. La arista toma el papel de la dependencia concreta que existe entre ellos, y según su tipo (objetivos, tareas, recursos), modifica la arista con un icono intermedio. Los diagramas de actores se complementan con los diagramas de justificación. Dentro del *Rationale diagram*se examinan los objetivos de un actor concreto (descomponiéndolos en subobjetivos) y sus dependencias con otros actores (con su influencia respecto a los objetivos). Este proceso se considera incremental, siendo habitual encontrar la identificación de nuevas dependencias. Este diagrama, posteriormente pasó a denominarse diagrama de objetivos, siendo más coherente con su función.

La siguiente fase (análisis de requisitos tardíos) repite lo visto anteriormente, considerando al sistema que se pretende construir como un nuevo actor, y utilizando los mismos diagramas. De esta forma se identifican los requisitos funcionales y no funcionales del sistema, describiendo al sistema dentro del entorno donde se pretende desplegar.

El diseño arquitectónico, configura la estructura de la organización del sistema a través de su descomposición en subsistemas relacionados mediante dependencias. Esta fase se desarrolla en tres etapas: definición de la estructura general del sistema, especificación de las capacidades disponibles para cada autor y la definición de tipos de agentes, a quienes se

les asignan las capacidades identificadas.

Para definir la estructura general del sistema, se considera el uso de diferentes estilos, que se utilizan a modo de patrones. Estos estilos se definen mediante una metaclase que, a partir de unos parámetros de diseño, presenta su asignación de objetivos y procesos. Algunos ejemplos de estos estilos son *"Structure-in-5", "Strategic Apex", "Middle Line"*.

Como su nombre sugiere, la fase de diseño detallado consiste en aumentar el nivel de detalle respecto a los componentes que se han identificado en la fase anterior. Esto incluye definir los protocolos que guiarán las comunicaciones que se han encontrado durante el diseño. Para dar soporte en esta fase, Tropos propone diferentes patrones sociales, que presentan situaciones comunes de interacción. Los patrones sociales se dividen en dos categorías, según la naturaleza de la interacción: pareja o mediación. Los patrones de pareja representan interacciones donde un actor se comunica con uno o más actores adicionales en una comunicación "de par a par". Un patrón de pareja que considera más de dos actores, tendrá un actor que tiene una misma interacción con más de un actor (como puede ser una subasta). Los patrones de mediación representan situaciones donde es necesario un intermediario, siendo el propósito de la interacción satisfacer las necesidades de los otros actores.

Durante la fase de implementación, se consideran actividades para guiar el traslado del diseño al código de la manera más natural posible, haciendo corresponder las propiedades de la plataforma de agentes objetivo con la semántica del diseño.

Tropos cuenta con una colección de herramientas de soporte muy completa en la cual destaca TAOM4E. TAOM4E aporta generación y test de código utilizando desarrollo dirigido por modelos. TAOM4E es software libre y en su versión actual se integra como *plug-in* de ECLIPSE.

Respecto a la enseñanza de Tropos, no se ha encontrado referencias.

2.1.3. PASSI

La metodología PASSI [Burrafato and Cossentino, 2002; Cossentino, 2005], integra modelos y conceptos orientados a objetos con AOSE, proponiendo una metodología "paso a paso"que cubre todas las fases del desarrollo (desde la especificación de requisitos hasta la implementación). PASSI se caracteriza por promover la adhesión a los estándares siempre que sea posible. Siguiendo esta línea, considera el uso de tecnologías bien establecidas como UML como lenguaje de modelado, la plataforma de la FIPA como destino de la implementación y XML para estructurar las comunicaciones.

PASSI considera un proceso de desarrollo compuesto de cinco fases. Cada una de estas fases guía una producción de modelos, donde el resultado de cada fase será su modelo respectivo. Estos modelos son: System Requirements Model, Agent Society Model, Agent Implementation Model, Code Model y finálmente, Deployment Model. Para la elaboración de estos modelos, se consideran diferentes etapas, por lo que el proceso en PASSI consiste en varias fases, compuestas de diferentes etapas. Este proceso es iterativo y, además, dado que la implementación se encuentra explícitamente dentro del proceso de la metodología, se dan actividades de testing entre iteraciones.

En [Cossentino, 2005] se encuentran el esquema de las fases y etapas de PASSI, que se muestra en la figura 2.1.

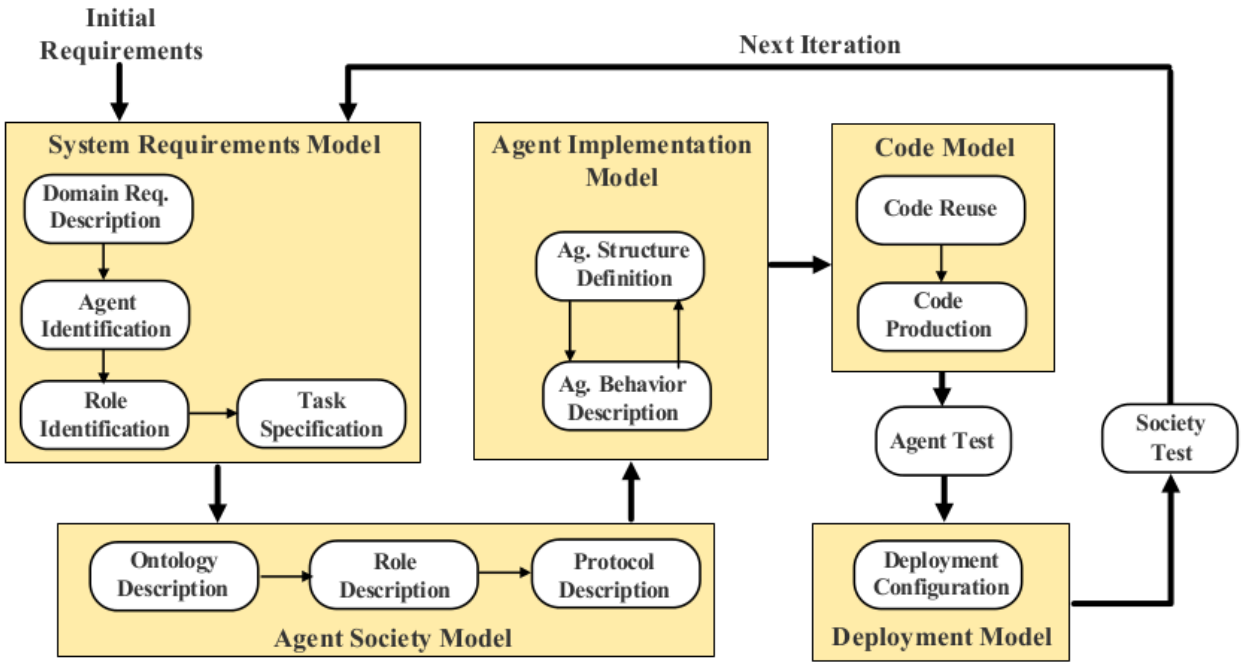

Figura 2.1: *Fases y etapas de PASSI.*

System Requirements Model reúne los requisitos del sistema en cuatro etapas. Su primera etapa, Domain Requirements Description, recoge en diagramas de casos de uso convencionales los requisitos de los escenarios posibles del sistema. Las demás etapas describen una especificación de alto nivel de los agentes, primero haciendo una identificación preliminar de los agentes y repartiendo los casos de uso, posteriormente planteando los roles de los agentes mediante diagramas de secuencia donde se presentan los escenarios principales del sistema, y finalmente se especifican las tareas asignadas a cada agente mediante diagramas de actividades.

Agent Society Model describe el conocimiento y las comunicaciones entre los agentes que pueblan el sistema y por tanto, la ontología del mismo. De esta forma, en las diferentes etapas, se compone una descripción de la ontología mediante un diagrama de clases, al que se añade otro que modela las interacciones (una clase por cada clase de agente identificada y una relación por cada comunicación). Posteriormente se asignan roles a las clases de agentes identificadas mediante otro diagrama de clases (en este caso, los agentes se representan con paquetes y los roles con clases). Finalmente, se especifican los protocolos que guían las comunicaciones entre agentes, siguiendo los contenidos en el estándar FIPA o definiendo nuevos mediante diagramas de secuencia AUML.

Agent Implementation Model comienza definiendo tanto la estructura general del sistema

como la estructura individual de los agentes. En el primer caso se hará mediante un diagrama de clases que representa las interacciones significativas con el entorno mediante actores. Por otra parte, para la estructura individual, es suficiente con un diagrama de clase corriente por cada clase de agentes.

La estructura definida será completada con la especificación del comportamiento de los agentes a nivel global mediante diagramas de actividades, donde se muestra la ejecución de las diferentes tareas y los mensajes intercambiados. Para expresar el comportamiento local de los agentes se considera suficiente una descripción informal de las tareas implementadas. Esto significa que la estructura del sistema y de los agentes condiciona el comportamiento de estos, y al contrario. De esta manera, existe una relación iterativa entre la estructura y el comportamiento del sistema.

Code Model ocupa el lugar de la fase de implementación. Esta fase se desarrollará primero generando el código a partir de los modelos mediante alguna de las herramientas de soporte, que incluyen la generación de plantillas además de la opción de reutilización de patrones.

Deployment Model describe el despliegue del sistema. Para ello, se utilizan diagramas de despliegue con una sintaxis extendida para poder expresar la movilidad de los agentes. Para finalizar la iteración se considera el testing del sistema desplegado.

Como soporte a la metodología, resulta destacable la herramienta *PASSI ToolKit* (PTK), que incluye generación de código a nivel de template. Posteriormente, se completa manualmente el código automáticamente generado previamente.

Respecto a la enseñanza de PASSI, no se ha encontrado referencias.

2.1.4. Prometheus

Prometheus [Padgham and Winikoff, 2005; Winikoff and Padgham, 2004] es una metodología orientada a agentes que nace con el propósito de ser práctica y de propósito general. Con esta premisa, se decide también a acceder a sistemas con agentes basados en objetivos y planes, y se compromete a proporcionar una serie de herramientas que faciliten el desarrollo a largo plazo. Cuando se define como práctica, quiere decir que tiene como objetivo atender las necesidades de un amplio espectro: desde las personas que se inician en los sistemas multiagente hasta las necesidades que demanda la industria.

Basándose en una definición de agente como la siguiente: "una entidad software autónoma, situada en un entorno, reactiva a los cambios en dicho entorno, proactiva en la persecución de objetivos y con habilidades sociales, siendo además flexible y robusta" [Padgham and Winikoff, 2005]; Prometheus identifica los siguientes conceptos: un agente se relaciona con su entorno mediante sensores y acciones, será proactivo si persigue objetivos, será reactivo si reacciona a eventos, tendrán planes y beliefs, y se comunicarán mediante mensajes que responden a determinados protocolos de interacción. Con estos términos, pueden definirse tanto agentes que se ajusten al modelo de Beliefs Desires Intentions (BDI) como agentes que no.

El proceso que propone Prometheus se descompone en tres fases: especificación del sistema, diseño arquitectónico y diseño detallado.

La fase de especificación del sistema define los requisitos del mismo en términos de objetivos, escenarios de casos de uso (o simplemente escenarios), funcionalidades y el conjunto de sensores y acciones. Los objetivos son buenos representantes de los requisitos pues es fácil que representen lo que se pretende del sistema. Estos objetivos pueden también formar parte de otros objetivos, es decir se consideran también subobjetivos; de hecho, nuevos subobjetivos suelen aparecer preguntándose cómo se consiguen los objetivos.

A partir del conjunto de objetivos identificados, se definen funcionalidades como fragmentos del comportamiento que mostrará el sistema, es decir, las habilidades que tendrá para poder alcanzar sus objetivos. Las funcionalidades del sistema se identifican agrupando objetivos con aspectos comunes, sensores y acciones relacionados con ellos, así como los datos que necesitan. Las funcionalidades tienen asociados descriptores que muestran la información necesaria: sus objetivos relacionados, las acciones que puede realizar y las situaciones que disparan dicha funcionalidad (triggers), así como una descripción textual.

Los escenarios son pequeñas secuencias en las que se muestra cómo se alcanza un objetivo o como se reacciona a un evento concreto. Estos escenarios a menudo contemplan variaciones, y cada paso representa una categoría, que podrá ser una acción, un percept, un objetivo, u otro escenario.

El entorno donde el sistema se situará se define también. Dando a los agentes que pueblan el sistema una interfaz en forma de sensores y acciones. Al considerar los sensores, se considera también un procesado por el cual extraer información útil.

La fase de diseño arquitectónico se centra en elegir las clases de agentes del sistema, describir cómo interactúan entre ellas y diseñar la estructura general del sistema.

Para elegir las clases de agentes presentes en el sistema, se tendrán en cuenta las funcionalidades previamente identificadas en común con los conjuntos de datos que necesitan manejar. Esta situación se representa en un *Data coupling diagram*. Partiendo de este diagrama, se puede decir que una buena elección es agrupar funcionalidades y conjuntos de datos que se relacionen entre sí de forma explícita, es decir, se encuentren firmemente relacionados. Una vez que se han elegido las clases de agentes, es útil representar la relación de estas con un diagrama llamado Agent acquaintance diagram, que muestra qué clases se comunican y en qué sentido.

A partir de estos diagramas, se elaboran *Interaction diagram* e *Interaction protocols*, que son diagramas similares a los diagramas de secuencia utilizados en UML. Igual que los escenarios, los Interaction diagram sólo describen situaciones concretas. Al desarrollar el diagrama e introducir variaciones hasta contemplar todas las situaciones posibles, logramos un Interaction protocol.

La estructura general del sistema consiste en un grafo dirigido donde los nodos pueden ser agentes, sensores, acciones, mensajes y conjuntos de datos, y las flechas indican interacción.

La fase de diseño detallado consiste en concretar los aspectos internos de los agentes, esto es, definir los agentes en función de capabilities, desarrollar process diagrams a partir de interaction protocols, y desarrollar las capabilities en términos de otras capabilities, planes, conjuntos de datos y eventos.

Como parte del proceso iterativo que supone ir encajando estas piezas, se incorpora un diagrama de tipo *Agent overview diagram*, donde se representa el agente en función de

capabilities y de la comunicación de estas (entre ellas y con el exterior). Este diagrama es similar al que muestra la estructura general del sistema, sólo que este se centra en el comportamiento local.

Para dar soporte a la metodología se ha diseñado la herramienta *Prometheus Design Tool* (PDT) que incorpora editores gráficos además de medios de control de la consistencia del sistema. Esta herramienta, además del soporte a los diagramas que forman la metodología, también puede exportarse hacia *JACK Development Environment* (JDE), que además de cubrir aspectos como los mencionados en PDT, también incorpora funcionalidad de generación automática de código. Esta generación automática en forma de templates se hace en código JACK.

Una característica negativa de Prometheus es que no considera la movilidad de los agentes. Esto es así porque a los autores no les parece un aspecto prioritario de un sistema multiagente. Además, tampoco trabaja en profundidad la identificación de requisitos, como sí hace Tropos.

Respecto a la enseñanza de Prometheus, en [Padgham and Winikoff, 2005] los autores afirman que el uso de Prometheus ha supuesto una mejora en sus clases. También describen su aplicación en el aula, como una perspectiva basada en proyecto 2.3.1 donde se propone el desarrollo de un proyecto determinado. Sin embargo, parecen dar a entender que supone una tarea individual, por lo que probablemente la experiencia con Prometheus se limite al diseño del sistema.

2.1.5. INGENIAS

La metodología INGENIAS [Pavón and Gómez-Sanz, 2003; Pavon et al., 2005], es una metodología que guía el desarrollo de sistemas multiagente desde el análisis hasta la implementación. INGENIAS sigue un proceso de desarrollo similar al proceso unificado, definiendo un conjunto de actividades que guían el proceso de elaboración.

En INGENIAS, el sistema se modela según diferentes vistas que se especifican a través de diferentes metamodelos, que se complementan con algunos diagramas UML, y en su conjunto representan la totalidad del sistema. Los metamodelos con los que cuenta INGENIAS se instancian en vistas y estos son: modelo de organización, modelo de agente, modelo de interacción, modelo de entorno, y modelo de tareas y objetivos.

- El modelo de organización, diseña la estructura del sistema, definiendo un espacio donde los agentes, sus recursos y sus tareas/objetivos conviven, además de cómo se relacionan. El modelo de organización, divide estos elementos considerando la definición de grupos y workflows, y define su funcionalidad mediante las tareas y los objetivos.

- El modelo de agente diseña el comportamiento de agentes individuales en función de sus roles, tareas, objetivos y su estado mental. Con estas herramientas se modela la funcionalidad del agente, donde además se podrán definir estados mentales determinados para una situación específica, para lo que se cuenta con diversas entidades para describir estados, como hechos y planes.

- El modelo de interacción define cómo tienen lugar las interacciones, qué actores toman parte en ellas y qué información se intercambia, además del objetivo que persiguen y el contexto al que pertenece. Las interacciones tendrán una especificación asociada, que podrá definirse a partir de diagramas de diferente procedencia (UML, AUML). Incluso se puede contar con la especificación GRASIA!, que define las interacciones a partir de unidades de interacción.

- El modelo de objetivos y tareas define la especificación de las tareas, así como su relación con los objetivos. Al especificar las tareas, se define qué información produce/consume y qué consecuencias puede tener respecto a los objetivos a los que está relacionada o al estado mental del agente.

- El modelo de entorno define la integración del sistema con otras aplicaciones u otros agentes. También define los recursos del sistema.

Los agentes modelados siguiendo INGENIAS siguen el principio de racionalidad, y la generación de código elige JADE como plataforma objetivo. Al igual que Prometheus, INGENIAS tampoco considera a los agentes móviles.

INGENIAS cuenta con la herramienta *INGENIAS Development Kit* (IDK) como soporte para el desarrollo siguiendo la metodología. Esta herramienta permite diseñar las vistas del sistema siguiendo los modelos definidos por la metodología, a la que añade soporte para validación y verificación. También se incluyen diferentes diagramas UML que contribuyen al desarrollo dirigido por modelos. Además, el IDK cuenta con generación automática de código, lo que le convierte en una herramienta muy completa. Sin embargo, esta herramienta no está completa y no permite incluir toda la semántica de la metodología, lo que supone una desventaja.

Una característica negativa de INGENIAS es que, al igual que Prometheus, no considera el modelado de agentes móviles.

Respecto a la enseñanza de INGENIAS, en internet se encuentran transparencias de clases de posgrado en las que se enseña INGENIAS. La inspección de estas transparencias parece limitarse a describir los elementos de la metodología. No se han encontrado sin embargo referencias o consejos acerca de cómo enseñar INGENIAS.

2.1.6. Agent-Oriented Programming versus Agent-Oriented Software Engineering.

La importancia del diseño aumenta conforme el sistema va creciendo. Esta es una constante de la que la ingeniería del software orientada a agentes no puede escapar. En el caso de sistemas pequeños, se puede proceder a programar sin una serie de métodos que aporten estructura al proceso, pero en estas condiciones es difícil conseguir un producto de calidad. Esta dinámica para proyectos más grandes no se puede utilizar, porque se vuelve impracticable.

Sin embargo la programación sigue jugando un papel importante. En las condiciones actuales, recorrer la distancia entre la especificación y el código implica cierta complejidad

en el caso de los sistemas multiagente. Esto es así porque supone encajar las abstracciones utilizadas para definir la especificación con las utilizadas por la plataforma objetivo [Nunes et al., 2011]. Además hay que tener en cuenta las abstracciones propias del lenguaje utilizado en la implementación.

Esta situación también dificulta la reutilización, por un lado, reutilizar fragmentos de modelado de las metodologías investigadas en las secciones 2.1.1, 2.1.2, 2.1.3, 2.1.4 o 2.1.5, puede no ser una opción porque resultan muy heterogéneas. Por otro lado, reutilizar diseños para distintas plataformas puede no ser una opción viable, al menos sin reimplementar todo el código. Esta situación de diversidad, aunque quizá apropiadas para casos concretos, resultan un problema para el caso general, dado que existe riesgo de que parte de los sistemas se diseñen e implementen de manera *ad hoc*, y en estas condiciones la posible reutilización parece muy limitada.

Por estas razones se entiende AOSE necesita a *Agent-Oriented Programming* (AOP), y sin AOSE sólo se podría aplicar AOP a pequeña escala. Esto implica que el estudio de AOSE no puede excluir al estudio en AOP.

Como lenguajes AOP, se pueden destacar Jason y 2APL.

- Jason [Bordini and Hübner, 2006; Bordini et al., 2007] consiste en un intérprete para una versión extendida de AgentSpeak, un lenguaje de programación orientado a agentes declarativo inspirado en la programación lógica. Las extensiones introducidas en Jason, resultan vitales para la implementación de sistemas multiagente con AgentSpeak. AgentSpeak modela el comportamiento de los agentes según un modelo de razonamiento específico, en este caso, el modelo BDI.

- *A Practical Agent Programming Languaje* (APAPL) es un lenguaje de programación orientada a agentes destinado tanto a la programación de agentes individuales como de sistemas multiagente [Dastani, 2008]. Los agentes individuales se diseñan siguiendo la arquitectura BDI, es decir en términos de creencias, objetivos, planes y un conjunto de reglas de razonamiento. Los agentes están situados en entornos, que pueden ser más de uno, que se modelan mediante objetos Java. Estos entornos pueden ser en principio cualquiera, siempre que cumplan con *Environment Interface Standard* (EIS), que relaciona de forma bidireccional plataformas y entornos de agentes arbitrarios.

2.1.7. Conclusiones

Las metodogías AOSE investigadas ofrecen enfoques muy diferentes (diferentes procesos de desarrollo, diferentes abstracciones y modelos, diferente cobertura del desarrollo de un proyecto AOSE).

Respecto al nivel de cobertura ofrecidos por las metodologías investigadas, todas salvo Gaia consideran un espacio suficientemente amplio del ciclo de vida del proyecto como para ser elegidas para enseñar AOSE.

Por el contenido estudiado, la única metodología que menciona el aspecto docente es Prometheus. Sin embargo, no se encuentran referencias que sugieran el método para enseñar

esta metodología. Tampoco en referencia a cómo se estudian las demás. El material docente que se ha podido reunir se limita a describir las metodologías (qué vistas del sistema incluyen o qué fases consideran).

2.2. Educando en Ingeniería del Software

Se ha visto a lo largo de la sección 2.1 que las metodologías AOSE, en general, no parecen preocuparse por aspectos docentes. En cambio sí que existen trabajos dentro de la ingeniería del software que atienden temas educativos. Por este motivo, aprovechando las similitudes que existen entre la ingeniería del software y AOSE, se plantea investigar qué problemas se han identificado en la ingeniería del software y qué se está haciendo para atenderlos, con el fin de buscar soluciones para AOSE. Parte de estos trabajos están dedicados a la confección de mejores planes de estudios.

A la hora de contemplar los contenidos que deben encontrarse en un plan de estudios, es tradicional que determinadas materias se consideren fundamentales (como aquellas asociadas a los créditos troncales), otros sean considerados importantes pero no fundamentales (un ejemplo común son los créditos obligatorios) y finalmente se encuentran materias que, siendo relevantes, se consideran optativas para cada centro de estudios. Como se puede ver a continuación, estructurar los conocimientos según estas tres categorías no es un trabajo nada trivial. Además de un análisis profundo de la disciplina, necesita un amplio consenso, sobre todo, a nivel de instituciones.

2.2.1. *"Body of Knowledge"* sobre ingeniería del software

Se distingue entre describir el conjunto de conocimientos presumibles en un Ingeniero del Software y el plan de estudios que lo hace posible. En el primer grupo, el representante más significativo es el *Software Engineering Body of Knowledge* (SWEBOK), siendo un compendio del conocimiento, técnicas y habilidades necesarias para practicar la ingeniería del software. En el segundo, destacan como representantes principales el *Software Engineering Education Knowledge* (SEEK) y *Computing Curricula* (CC).

SEEK se desarrolla con SWEBOK como referencia principal, y se presenta dentro del documento *Software Engineering 2004* [Abran et al., 2004], siendo el resultado de un esfuerzo conjunto entre la ACM y el *IEEE Computer Society*. La perspectiva aplicada al desarrollo de SEEK, es la de considerar SE como una disciplina ingenieril con su base en la informática. Sin embargo, apunta a que su relación con la *Computer Science* es más fuerte que en otras ingenierías tradicionales.

SEEK divide en diferentes áreas el conocimiento que debe trabajar un estudiante de ingeniería del software. Estas áreas se granulan en unidades, que finalmente se estructuran en temas. Las áreas que se incluyen en SEEK como parte del mínimo conocimiento fundamental que se debe trabajar en la docencia de la ingeniería del software, son diez: Fundamentos de la computación, Fundamentos matemáticos e ingenieriles, Práctica profesional, Modelado y

análisis de software, Diseño de Software, Verificación y validación de software, Evolución del software, Proceso de desarrollo de software, Calidad del software, y Gestión de software.

La finalidad del conjunto de conocimientos que se encuentran recogidos dentro de SEEK, será la de considerarse también como *guidelines* o directrices para construir los planes de estudio. Los planes de estudio serán un elemento muy importante, dado que implementan de forma concreta los contenidos que se han visto, y configuran la formación que se desea que el alumno tenga a su disposición al finalizar sus estudios.

CC [Shackelford et al., 2006], al igual que SEEK, se concentra en el nivel de título de grado. Sin embargo, a diferencia de SEEK, CC va más allá al considerar no sólo SE, sino las otras disciplinas principales como *Computer Science, Computer Engineering, Information Systems e Information Technology*. Además, señala las similitudes y diferencias entre estas disciplinas, destacando las diferentes características que se observan en los graduados de las diferentes disciplinas.

Estos *Body of Knowledge* también consideran importante señalar las diferencias que se encuentran entre la ingeniería del software e ingenierías más tradicionales. Estas pueden ser, por ejemplo, que su base científica consiste en la *Computer Science* en lugar de ciencias naturales, o que la matemática discreta tiene más relevancia que el cálculo y la matemática continua. También se puede destacar que la fase de fabricación, en el sentido tradicional, no existe; que el software no sufre degradación, y que la parte de evolución del software como parte del ciclo de desarrollo no existe dentro de otras ingenierías.

2.2.2. Elaborando de planes de estudio para la ingeniería del software

Dentro de los autores que razonan acerca de los planes de estudio para la ingeniería del software, David Lorge Parnas resulta relevante. Parnas, además de ser conocido por ser uno de los primeros en aplicar principios y técnicas ingenieriles al software, es un destacado defensor de la ingeniería del software como una ingeniería legítima. Por su parte, Parnas propone qué contenidos deben enseñarse tanto en la enseñanza de computer science [Parnas, 1990], como de ingeniería del software [Parnas, 1999].

Parnas, defiende ambas disciplinas como entes distintos, primero la Computer Science como una disciplina diferente de las ciencias físicas o de la ingeniería industrial, y más tarde la ingeniería del software como algo más que un contenido de la *Computer Science*. De hecho, la relación entre las ciencias físicas y la ingeniería industrial le sirven a Parnas como un buen espejo para exponer su visión de la relación entre computer science e ingeniería del software: igual que la física compone la fuente de conocimiento sobre la que se basa la ingeniería industrial para conformar una especialización de la ingeniería, el conocimiento dentro de la computer science debería servir para entender la ingeniería del software como una especialización de la ingeniería.

Esto quiere decir que el resultado de estudiar ingeniería del software debería ser un ingeniero, alguien capaz de solucionar problemas, donde en el caso de la ingeniería del software se producen soluciones en forma de (productos) software. Esta persona, por tanto deberá

ser capaz de diseñar un producto apropiado a un problema concreto de forma correcta y eficiente, deberá conocer la forma de construirlo, conocerá y aplicará técnicas para comprobar su corrección, y además será capaz de identificar qué tareas serán necesarias para su apropiado mantenimiento. Nociones que van más allá de tan sólo ser un buen programador.

Parnas además defiende que, al considerar la ingeniería del software como rama de la ingeniería, las personas que la estudien deberán estar familiarizados con áreas propias de otras especializaciones de la ingeniería, como teoría de control o termodinámica. Además, la aproximación a los contenidos debe hacerse desde los aspectos fundamentales. Este aspecto puede pasar desapercibido por parecer obvio pero, por poner un ejemplo, a menudo se encuentran cursos de programación orientada a objetos que se limitan a enseñar un lenguaje concreto. En consecuencia, si el lenguaje se vuelve obsoleto, la educación que recibe el alumno será obsoleta también.

En general se puede observar que la visión de Parnas de la ingeniería del software y las directrices que propone para la confección de planes de estudio, no están muy lejanas de las que se siguen durante la creación de SEEK. Sin embargo, mientras que SEEK considera que la ingeniería del software debe tener una relación más próxima a su ciencia base que otras ingenierías, Parnas argumenta que esta relación debe ser similar a la que tienen las demás.

Por otra parte, enseñar los fundamentos no quiere decir que se deba huir de nuevas influencias. En [Ali, 2006], Ali defiende que unos estudios destinados a formar a un profesional que deberá adaptarse a cambios continuos, deberían mostrar la misma capacidad de adaptarse a los cambios. En esta línea propone cinco sugerencias para mejorar la educación en la ingeniería del software e influenciar futuros planes de estudios.

- Afirma que es deseable acercar al estudiante como individuo al mundo real, donde se producen aplicaciones serias que serán usadas lo antes posible. Esto se puede conseguir utilizando la implementación de aplicaciones que serán usadas dentro de la universidad, consiguiendo un ámbito serio y un objetivo con utilidad, que mejora la motivación y predisposición del alumno.

- Propone que las técnicas que se inculcan al alumno deben estar actualizadas, recibiendo un entrenamiento que les prepare para ser profesionales del software, que se encuentran en situación de formación continua.

- Destaca el sentido interdisciplinar de las ingenierías, donde es habitual que profesionales de especialidades diferentes cooperen en un mismo proyecto. Este sentido interdisciplinar, deberá fomentarse, utilizando para ello proyectos que requieran conocimientos de otros campos, yendo así más allá de los contenidos habituales.

- Argumenta la importancia de introducir avances relevantes en la ingeniería del software dentro de los planes de estudios, refiriéndose como entidad susceptible de ser añadida las nuevas técnicas y tecnologías.

- En su última sugerencia, se refiere a la importancia de la capacidad para la comunicación. Esta capacidad debe fomentarse y reforzarse desde los primeros cursos, solicitan-

do comunicación de forma oral y por escrito a los alumnos (informe de sus progresos, por ejemplo).

Por su parte, Shaw [Shaw, 2000] coincide con Parnas en reclamar un acercamiento integral hacia la ingeniería, identificando una serie de hitos y aspiraciones para la pasada década. También coincide con Ali al reclamar más dinamismo como una característica imprescindible de los ingenieros. Shaw identifica diferentes defectos en los alumnos que siguen los planes de estudio actuales: comenzar a codificar antes de razonar el problema, implementar el primer diseño que se encuentra, codificar desde el punto de vista del implementador y no del cliente. También propone diferentes soluciones para solventarlos: presentar teoría y modelos en contextos prácticos, requerir al menos dos diseños razonados antes de permitir el paso a la implementación, consultas con usuarios finales. En estos dos conjuntos resulta un elemento central la manipulación y modificación de software anteriormente diseñado e implementado, en lugar de la clásica visión en donde al alumno se le asignan problemas que debe solucionar elaborando propuestas desde cero. También señala como un gran error las tareas y ejercicios de usar y tirar clásicos en los planes de estudio, porque este tipo de trabajos no fomentan la construcción de software mantenible y bien documentado.

Este tipo de problemas y defectos son relevantes para AOSE porque no se consideran desde la base durante el estudio de la ingeniería del software. Dado que el estudio de AOSE se suele llevar a cabo después de los primeros años de contacto con la ingeniería del software, estos problemas deberán ser atendidos y considerados durante el desarrollo del estudio de AOSE.

Shaw [Shaw, 2000] argumenta que se debe enseñar a ser crítico con los diseños, y que para enseñar a manejar y construir buen software, se debe presentar al alumno sistemas que sirvan de buenos ejemplos para crear un punto de partida. Shaw señala que, además, es deseable que este software que se use de ejemplo sea ejecutable. Shaw también destaca el contexto de los ejemplos utilizados como otra característica deseable, dado que un buen contexto favorece que los ejemplos sean recordados posteriormente, y por tanto, favorece de esta manera la reusabilidad.

Las opiniones recogidas durante el estudio y sus diferentes ángulos, unidas a los *Body of Knowledge* existentes, conforman un marco más completo de la línea a la que se reclama que avance la educación en la ingeniería del software.

2.2.3. Conclusiones

Se observa por los trabajos estudiados que existe interés por mejorar la educación en la ingeniería del software. Mejorar la educación de esta disciplina debería implicar producir mejores ingenieros.

En esta línea, los diferentes *Book of Knowledge* que definen una base para la ingeniería del software y su educación, suponen un buen referente para mejorar AOSE y la educación en AOSE. Sería interesante disponer de un compendio que definiera el conocimiento necesario para practicar AOSE. Como ni CC ni SEEK consideran directamente AOSE, sería conveniente un trabajo que lo considerara de forma más específica.

Estos trabajos, además, encajan en un trasfondo definido por dotar a la ingeniería del software de un mayor carácter ingenieril frente al enfoque de ciencias de computación. Este asunto también es importante para AOSE, porque una preocupación importante de AOSE es tener mayor impacto dentro de la industria.

Respecto a las formas de mejorar la educación en la ingeniería del software, en los trabajos investigados se aprecia también la demanda de ejemplos educativos. El ejemplo educativo, se reconoce por los autores como un elemento que cuenta con la opinión favorable de educadores y alumnos, y sin embargo sus posibilidades no se han aprovechado suficientemente. Además, se ha identificado como una característica deseable de estos ejemplos la capacidad de ser ejecutables (proyectos que se puedan ejecutar, por tanto, no será suficiente proyecto que aún esté en fase de diseño) y contar con un contexto definido.

2.3. Practicando la Ingeniería del Software

Dentro de la educación en la ingeniería del software también existen distintos trabajos que atienden la práctica. Estos trabajos presentan ventajas y dificultades de utilizar diferentes aproximaciones. Tanto de las ventajas, como de los inconvenientes, se podrá extraer ideas para mejorar la enseñanza de la práctica de AOSE.

Se considera la ingeniería del software como "la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, modificación y mantenimiento de software, es decir, la aplicación de ingeniería al software" [IEEE, 1990]. De esta definición podemos extraer que un proyecto es un elemento primordial en la ingeniería. Si es así, sería lógico asumir que los autores se centran en el concepto de proyecto y lo practica exhaustivamente, pero no ocurre así. Se podría hablar de dos escuelas:

- Presentar los contenidos de manera que se puedan aplicar al desarrollo de un proyecto con el fin de obtener un resultado lo menos "de juguete" posible. Dentro de estos trabajos, se observa la respuesta que tiene esta propuesta sobre los alumnos, y las hipótesis que se plantean para mejorarla.

- Ejemplos concretos que ilustren aspectos específicos de la ingeniería.

2.3.1. Ingeniería del software centrada en proyecto

Aprender ingeniería, no supone aprender un proceso sino entenderlo. Por eso, a la hora de plantear la docencia de una ingeniería, algunos autores llegan a la conclusión de que la mejor forma de aprender una ingeniería es practicarla [Mills et al., 2003]. Dado que, como se ha visto en la sección anterior, existe una gran preocupación por integrar la ingeniería del software en el conjunto de disciplinas ingenieriles, no es descabellado pensar que la misma técnica se puede aplicar para la ingeniería del software.

En [Gnatz et al., 2003], los autores exponen su experiencia al utilizar esta aproximación en su clase de ingeniería del software. Además, proponen que tanto el proyecto como la dinámica del mismo se asemejen lo más posible a un caso real. Esto se hará con un enfoque de

auditoría, donde un representante de una empresa real propone un producto, y los alumnos, estructurados en diferentes equipos de desarrollo, negocian la funcionalidad que deberá contener. Este método, como los propios autores reconocen, presenta algunos problemas como los conflictos que se encuentran al intentar conseguir un objetivo docente mientras se intenta alcanzar una configuración real, dado que son dos actividades con necesidades diferentes. En este orden de cosas, los autores expresan sus reflexiones al encontrarse ante la disyuntiva de involucrar a todos los alumnos en todos los aspectos o de separar su atención, con fin de dar como resultado un producto más sólido. En este caso los autores eligen el primer caso, que parece más beneficioso para la experiencia docente del alumno, que participará en todas las etapas del desarrollo.

Respecto a las conclusiones que tanto los autores como los propios alumnos destacan del transcurso del proyecto [Gnatz et al., 2003], se extrae la experiencia de enfrentarse con los problemas de planificación, comunes en los proyectos reales. Un ejemplo de problema puede ser el pecar de exceso de optimismo. Los alumnos además, destacan los problemas de comunicación que surgen al trabajar en grupo, las dificultades que aparecen al manejar proyectos complejos, y el propio proceso de desarrollo. Pese a que los autores se muestran satisfechos con los resultados, los problemas de planificación llevaron al recorte de la experiencia de mantenimiento, y no pudo trasladarse al alumno la importancia de la buena documentación.

Se puede entender que este tipo de experiencias son muy convenientes a la hora de enseñar ingeniería del software, dado que el alumno tiene la posibilidad de experimentar las diferentes actividades que supone el desarrollo de un proyecto. Además, el alumno tiene la oportunidad de experimentar los desarrollos en grupo (que rara vez se pueden tratar dentro de otras asignaturas), o de participar dentro de aplicaciones cuya elaboración se presenta inabarcable para un solo implementador y del conjunto de técnicas utilizadas para que se alcance el producto deseado.

En esta línea, [Pucher and Lehner, 2011] exponen sus experiencias después de varios años de utilizar aprendizaje basado en proyecto. Un enfoque que, entre otras ventajas, motiva al alumno a involucrarse. En su opinión, siguiendo esta aproximación se cubren varios principios educativos importantes, porque a través de ella los contenidos se escogen con respecto a situaciones futuras, para las que se puede ofrecer asistencia. Además, los contenidos a tratar están orientados por el nivel de conocimiento y los métodos de la ciencia concreta a la que pertenecen. Los autores señalan a su vez que será muy importante la elección de ejemplos representativos que cubran los casos típicos.

Dentro de sus reflexiones [Pucher and Lehner, 2011], identifican factores que consideran a tener en cuenta a la hora de seguir este procedimiento. A su vez, dentro de estos factores críticos, se observan los diferentes niveles de motivación observados en los alumnos (o bajos o muy altos) o el nivel de experiencia dentro de este tipo de dinámica, tanto de los alumnos como del profesor. El primero, condiciona porque los alumnos sin experiencia tienden a atascarse en las primeras etapas del proyecto, un tiempo que probablemente no se pueda recuperar, y dado el tiempo limitado, será influyente en el resultado final. Además, el alumno con poca experiencia puede sentirse ridículo al acudir al profesor con preguntas triviales, provocando que esa duda no se resuelva y dan la posibilidad de que puedan apa-

recer problemas más tarde. El segundo, será determinante en el sentido de experiencia en ese proyecto específico, que influirá la capacidad del profesor para guiar apropiadamente el desarrollo del proyecto.

También, aunque consideran que aporta mejoras interesantes para los alumnos, también observan una serie de desventajas, como la dificultad que encuentra el docente para hacer evaluaciones justas. En una situación donde existe una gran diversidad de proyectos, podrá no poder encontrarse una forma estandarizable de valorar el trabajo. Precisamente, la coexistencia de diferentes proyectos supone otra desventaja, dado que a los ojos de los alumnos, puede dar una impresión caótica.

Dentro de la evaluación de los proyectos [Pucher and Lehner, 2011], los autores afirman que aparecen resultados interesantes respecto de la propuesta original del proyecto. En los proyectos propuestos por el equipo docente, la calificación (en términos estadísticos) resulta más alta, y sin embargo, los autores afirman que los alumnos que desarrollan proyectos propuestos por alumnos aprenden más.

Por otra parte, en [Claypool and Claypool, 2005], los autores proponen que además, este proyecto pueda consistir en diseñar un juego de ordenador. Para esta cuestión, los autores preparan una serie de proyectos basados en juegos que cumplen con sus criterios. Esta opción suele ser atractiva porque la oportunidad de diseñar un juego puede parecer más entretenida, al desarrollar una herramienta de ámbito lúdico. Además, como afirman en [Pucher and Lehner, 2011], captar el interés del alumno se considera deseable.

Esta alternativa presenta una experiencia con diversas lecturas, algunas muy útiles para el alumno. Por ejemplo, alumnos que se consideren aficionados a los juegos de ordenador, pueden encontrar que los compañeros que no comparten esa afición, y por ello no entienden el producto, ni las necesidades que debe cubrir. De esta forma tan sencilla, el alumno puede entender la gran importancia de comprender el dominio del producto antes de lanzarse a su desarrollo. En este sentido, el contexto en que se sitúa el proyecto resulta importante para el alumno. Mientras que el desarrollo de una aplicación para un dominio extraño puede dejar al alumno ajeno al contexto, la construcción de una herramienta de fines recreativos parece mejorar la predisposición del alumno.

Algunos autores identifican la globalización de la ingeniería del software como un aspecto a tratar dentro de la enseñanza de esta. Siguiendo esta idea, en [Petkovic et al., 2006], Petkovic et al, muestran su experiencia al introducir el concepto de globalización dentro de su asignatura de ingeniería del software, donde proponen comparar el transcurso de proyectos desarrollados por grupos formados por alumnos de su universidad y por grupos globales (grupos que combinan alumnos locales con alumnos de la universidad de Fulda, Alemania). Además, expondrán una serie de líneas de mejora, unas propuestas por los alumnos y otras propuestas por los autores. En este sentido, los alumnos parecen agradecer las el tiempo de clase destinado al trabajo en grupo, y el método de evaluación que, a diferencia de otras aproximaciones donde se realiza una evaluación continua, no evalúa el producto desarrollado de forma iterativa (si en cambio, el acercamiento a las técnicas y procesos que se intentan introducir).

Se puede ver que estos autores identifican como un elemento principal en la ingeniería del software su aplicación a un proyecto concreto, para lo cual presentan diferentes recetas.

Se puede concluir a su vez de estos trabajos que promover el acercamiento de los alumnos a unas condiciones próximas a la realidad de un desarrollo es importante, sobre todo en cuanto a dinámica de trabajo y comunicación. También, se puede identificar que la temática del proyecto parece ser un aliciente para estimular el interés, y con él, el rendimiento.

Resumimos en la tabla 2.1 las impresiones revisadas.

Autores	**Tipo de proyecto**	**Objetivo de mejora**	**Origen del proyecto**	**Dificultades observadas**
Gnatz et al	Proyecto real	Disposición laboral	Empresa	Planificación, comunicación
Pucher & Lehner	Múltiples proyectos	Motivación	Docentes / Alumnos	Estancamiento / Predisposición docente
Claypool & Claypool	Juego	Motivación	Docente	No especificadas
Petkovic et al	Proyecto global	Globalización	Docente	Comunicación, sincronización

Cuadro 2.1: *Resumen de los trabajos investigados que utilizan aproximación basada en proyecto.*

2.3.2. Enfoque de ejemplos concretos.

Por otra parte, dado que el desarrollo de grandes aplicaciones como objetivo de un curso en ingeniería del software no es nuevo, no es raro encontrar autores que se desmarcan de esta forma de enseñar ingeniería. Ya desde el año 1990, [Parnas, 1990] explicaba su rechazo a la aproximación basada en proyecto. Según Parnas, utilizar este enfoque no es adecuado porque pueden propiciar el que el alumno se atasque en superar dificultades concretas, ya sea referidas a la tecnología que está utilizando para el desarrollo del proyecto, u otros aspectos que resultan irrelevantes o superficiales. Este fenómeno es frecuente [Parnas, 1990], y provoca que los alumnos, al encontrar este tipo de dificultades caigan rápidamente en la trampa de aceptar un código que simplemente parece que funciona bien, sin darse cuenta de que realmente no es correcto. Otros autores se refieren a este suceso como el síndrome "¡pero funciona!" [Gómez-Martín and Gómez-Martín, 2009]. Para Parnas, utilizar el desarrollo de aplicaciones como criterio de evaluación es un sinsentido. En su lugar, propone que la implementación de programas debería ser similar a la utilización de la resolución de problemas en las asignaturas de matemáticas. Es decir, Parnas propone una dinámica de tareas basadas en problemas y ejercicios de escala reducida, disminuyendo el enfoque, y permitiendo una mayor atención a los aspectos fundamentales.

Otro inconveniente denunciado por [Parnas, 1990], será que el desarrollo de estas aplicaciones caerá en saco roto, al ser productos que probablemente nunca serán usados por nadie. Esta situación, donde se carece de ningún tipo de respuesta o realimentación por parte de un usuario (en inglés, feedback), ahonda en el error del alumno que cree que el producto es correcto, cuando probablemente no será así. Además, el que el alumno sea consciente de ello, no es beneficioso, dado que "no lo va a usar nadie" no se tomarán las molestias precisas para aplicar medidas sistemáticas que aseguren la corrección del producto que se está realizando.

Compartiendo la visión de la ingeniería del software de Parnas, [Offutt, 2013] propone una dinámica diferente a la hora de presentar la ingeniería del software a los alumnos. Esta forma alternativa de enseñar ingeniería se cimenta en tres aspectos, que son: buscar pensamiento divergente en lugar de pensamiento convergente, fomentar el aprendizaje colaborativo en lugar del individual y la utilización de conjuntos heterogéneos de tareas, en lugar de homogéneos.

Estas tres claves se identifican al revelar aspectos importantes de la práctica de la ingeniería. Si dentro de las clases se presentan problemas donde tan solo se contempla una solución, y los alumnos deberán aplicar un proceso de refinado para encontrarla, estamos fallando en hacer hincapié en una situación fundamental de la ingeniería, donde lo importante es identificar las diferentes soluciones posibles a un problema, y de entre ellas, identificar la mejor basándonos en el contexto y los recursos disponibles.

Por otra parte, el aprendizaje individual, clásico a la hora de evitar el plagio por parte del alumno, resulta poco producente en la práctica de la ingeniería, donde los proyectos se desarrollan entre grupos y el desarrollo es colaborativo. Además, la programación de una misma serie de tareas para todos los alumnos falla en fomentar el diferente repertorio de habilidades que tendrán los alumnos. En su lugar, [Offutt, 2013] propone ofrecer distintas opciones, buscando que el alumno se desarrolle en los aspectos que mejor se ajustan a su capacidad y habilidades.

Aunque algunas ideas de Offutt no son incompatibles con el aprendizaje basado en proyectos, por la forma en que plantea su alternativa, da a entender que coincide con Parnas en que tareas de menor tamaño pueden resultar más adecuadas. De cualquier modo, ambos autores proponen ideas que pueden ser interesantes para aplicar a AOSE.

2.3.3. Enseñanza de metodologías

Dentro del estado del arte existen también autores que trabajan buscando las claves de la enseñanza de metodologías, con el fin de plantear cómo enseñarlas o de identificar la más apropiada para el proceso docente. En este sentido, será interesante también recoger sus impresiones acerca de qué características hacen a una metodología más adecuada que otra para utilizarla en las aulas.

En [Hazzan and Dubinsky, 2003], sus autores indagan buscando los principios que marcan la enseñanza de una metodología. Entre estos principios también incluyen directrices sobre cómo implantar estas técnicas en un departamento. Eligen diez, que dividen en cuatro categorías que se distinguen por estar relacionadas con la forma de configurar el curso, cómo enseñar la metodología, los aspectos sociales o con el meta objetivo de que la metodología

que se presenta deje marca en el alumno (debe invitar a que el alumno recurra a ella fuera del curso, como muestra de que es una herramienta útil).

Si se presta atención a lo estrictamente relativo a la enseñanza de una metodología, se reducen a cuatro principios:

1. Sintetizar las características de la metodología que se enseña dentro del aula y señalar los aspectos en que se debe poner el énfasis.

2. Ajustar la metodología para la práctica de la docencia universitaria.

3. Reflejar los conceptos de la metodología en el entorno de desarrollo.

4. Situar la metodología que se está enseñando en el contexto de la práctica de la ingeniería del software en la industria y mostrar su relación con otras técnicas y metodologías.

Para mostrar un ejemplo de cómo hacer este esfuerzo, utilizan su experiencia en la enseñanza de la metodología eXtreme Programming (XP).

- Con respecto al principio 1, a su juicio, las doce prácticas más relevantes de XP se pueden agrupar según dos dimensiones: una según aspectos técnicos o aspectos sociales, y otra según conciencia cognitiva, es decir, lo conscientemente que debe practicar. Por ejemplo, la refactorización de código supone una tarea relativa a aspectos técnicos con alta consciencia, la programación por parejas es una tarea relativa a aspectos sociales y con una consciencia media, y la integración continua supone una tarea técnica de baja consciencia. Se entiende que las tareas con baja consciencia son más fáciles de entender que las tareas con alta consciencia.

- Con respecto a los principios 2 y 3, uno de estos ajustes se introdujo porque se temía que los alumnos hicieran estimaciones al alza con el fin de reducir su carga de trabajo, por lo que se eligió adaptar ligeramente las reglas que las tareas fueran justas. Además, para que el entorno tuviera capacidad para fomentar el uso de las prácticas de la metodología, se limitó el número de ordenadores disponibles (de forma que se forzara la programación por parejas), y además se limitó la integración a las sesiones conjuntas.

- Referido al principio 4, al tratarse de XP, parte de la introducción a ella debería hacerse desde un punto de vista donde se argumentan las causas que han dado lugar a las metodologías ágiles.

Evidentemente, al enseñar una metodología distinta a XP, la reflexión sobre cómo reflejar sus aspectos fundamentales puede ser muy diferente, pero la aproximación de [Hazzan and Dubinsky, 2003] es un buen referente acerca de cómo se puede plantear este problema.

En [Halling et al., 2002], muestran su experiencia al enseñar la metodología de proceso unificado (*Unified Process*). Sus impresiones muestran que los alumnos a menudo muestran

dificultades con aspectos como el paradigma orientado a objetos, tanto en diseño como en implementación. Además, señalan los problemas que encuentran los alumnos al diseñar, siendo una dificultad destacable el tener que relacionar unos diagramas con otros. Al igual que en el caso anterior, los autores afirman que fue necesario modificar ligeramente la metodología para la implantación dentro del curso. Estos cambios afectan tanto a los roles considerados en la metodología como a los artefactos, siendo necesario en ambos casos añadir nuevos elementos.

En [Runeson, 2001], Runeson presenta un pequeño experimento donde se compara el rendimiento de unos alumnos de primer curso y unos alumnos de posgrado, a los que se les intenta enseñar la metodología *Personal Software Process* (PSP), con el fin de evaluar si las metodologías de desarrollo son adecuadas para introducirse de manera temprana, y mantener un enfoque ingenieril desde el principio de la titulación. Para llevar a cabo la comparación, se utilizaron diferentes baremos que contemplarán aspecto relativos a la calidad del producto implementado, ya sea en términos de programación (tamaño de los códigos producidos y número de errores encontrados, etc), o en términos del progreso siguiendo el proceso (tiempo empleado en las tareas que forman parte de las prácticas de la metodología, la mejora que produce el uso de la metodología a lo largo del proceso según se van adaptando a ella, su actitud hacia el uso de dicha metodología).

Los resultados en general son desfavorables a los alumnos de primero. Sin embargo, de esta experiencia, Runeson extrae algunas conclusiones interesantes. Por ejemplo, encuentra que los alumnos de primer curso se muestran bastante más receptivos a las ventajas del uso de una metodología de desarrollo. Este dato muestra la importancia de inculcar este tipo de prácticas desde etapas tempranas en lugar de esperar a los últimos cursos, donde el estudiante puede tener una serie de hábitos que, sencillamente, no está dispuesto a cambiar. Aun así, los alumnos de primero no están preparados para este tipo de desafíos que implican desarrollos fuertes, dado que encuentran muchas dificultades en aspectos como la programación, que no son el objetivo. Debido a estos motivos, la introducción de la metodología de desarrollo como parte de la ingeniería del software fue finalmente movida al segundo curso.

En [Bolognesi et al., 2006], los autores presentan su propuesta para modelar sistemas orientados a organizaciones, para lo cual combinan prácticas de la ingeniería del software como el proceso unificado y principios propios de las simulaciones basadas en agentes. En sus reflexiones, apuntan algunas preguntas típicas que les plantean los alumnos sobre el modelado. Estas preguntas se refieren a los diagramas necesarios para modelar el sistema correctamente (los diagramas que son necesarios, y por qué estos y no otros), a las propiedades de un buen diseño (cómo conseguir un buen diseño) y a cómo relacionar los modelos con el producto final, que será el programa.

Estos problemas de diseño partían del hecho de que a los alumnos les resultaba muy complicado saber qué modelar y cómo expresar el modelo. En este sentido, y como una herramienta muy útil para entender el modelado, los autores señalan a la generación de código como un importante punto de referencia. Su solución para esta situación de desconocimiento, fue plantear como estrategia la presentación de un sistema completamente desconocido y de su diseño en UML, utilizando una aproximación ligera hacia la ingeniería inversa.

En la tabla 2.2 se encuentran recogidas las impresiones del trabajo de los autores inves-

tigados en esta sección.

Autores	**Metodología**	**Dificultades**
Hazzan & Dubinsky	eXtreme Programming	Adaptación de la metodología al entorno educativo
Halling et al	Unified Process	Paradigma, Correspondencia entre diagramas
Runeson	Personal Software Process	Codificación, Resistencia al uso de la metodología
Bolognesi et al	Unified Process	Diseño, Correspondencia Diseño-Implementación

Cuadro 2.2: *Metodologías aplicadas a la educación en IS con las dificultades detectadas por los autores.*

2.3.4. Conclusiones

Se ha visto en los trabajos investigados que la utilización de proyectos para enseñar ingeniería del software es importante porque el proyecto es un elemento principal en esta disciplina. Sin embargo la introducción de proyectos en las clases está expuesta a problemas de planificación o comunicación, y suponen un esfuerzo importante por parte del docente. Además requieren que el alumno se encuentre motivado, por lo que el enfoque de algunos trabajos se dirige hacia mejorar la motivación.

Otros puntos de vista como la propuesta de Parnas (explicada en la sección 2.3.2), pueden privar al alumno de experimentar el ciclo de vida de un proyecto, pero sin embargo, al proponer tareas más pequeñas, le vuelven susceptibles a la utilización de ejemplos. Algo similar ocurre respecto a la propuesta de Offut (explicada en la sección 2.3.2).

Respecto al estudio de la enseñanza de metodologías, se observa que para presentar la metodología puede ser necesario adaptar tanto el entorno educativo como la propia metodología. A su vez, es esperable que no todos los métodos de la metodología se pueden implantar en el aula de la misma manera, unos serán más sencillos de entender que otros.

Se observa en los trabajos investigados resistencia a los conceptos de diseño una vez que existe cierta base de programación. Para enseñar una metodología, puede ser necesario hacer énfasis en el diseño, y utilizar herramientas de generación automática de código puede ser de ayuda.

La generación automática de código también es apropiada para el modelado de sistemas, aunque existen dificultades para establecer una correspondencia Modelo-Código. Respecto al modelado, se observan problemas a la hora de establecer correspondencia entre los distintos diagramas que forman el diseño. También se encuentran dificultades al diferenciar cualitativamente un diseño de otro, o a elegir los modelos a incluir.

2.4. Teorías de aprendizaje

La educación en la práctica de la ingeniería del software, ya sea mediante proyectos, mediante recursos de menor escala o a través de la adhesión a una metodología concreta, es muy susceptible de beneficiarse de las teorías de aprendizaje. Esto es apropiado porque las teorías de aprendizaje estudian aspectos cognitivos con el fin de mejorar la asimilación de información, proponiendo distintas sugerencias que generalmente se refieren a situaciones prácticas.

Existen diferentes teorías dentro del campo de la psicología que tratan sobre el aprendizaje. Estas teorías cubren diferentes niveles de procesos cognitivos, y a menudo identifican aspectos sociales y aspectos individuales.

Teniendo en cuenta el contexto de la educación en AOSE, donde se tiene como objetivo aprender tareas difíciles como son el modelado de sistemas o la programación de agentes, resultan relevantes los trabajos que estudien la asimilación de información compleja. En [Pollock et al., 2002], utilizan la teoría de carga cognitiva [Sweller, 1994] para aplicar descomposición a información compleja, y miden su mejora en el rendimiento del aprendizaje. Según la teoría de carga cognitiva, la cantidad de información que se puede intentar procesar es limitada. En consecuencia, asimilar varios conceptos relacionados en paralelo puede no ser posible. Por el contrario, se podrán asimilar varios conceptos en serie si estos no están relacionados, al poder ser aprendidos y entendidos (parcialmente) de forma individual, para posteriormente ser procesados en su conjunto y alcanzar una comprensión completa.

La clave, por tanto, para aprovechar este tipo de descomposición, será encontrar una división capaz de presentar la información por separado, perdiendo la menor información posible. Esta teoría resulta muy útil para comprender cómo funciona el aprendizaje y qué podemos hacer para mejorarlo. Sin embargo, encontrar la mejor forma de segmentar la información resulta una tarea totalmente *ad hoc*.

Otra línea de trabajo, examina las propiedades que debe poseer el objeto de aprendizaje (lo que se pretende que aprenda el sujeto) para que el aprendizaje sea posible. En [Marton and Pang, 2006], proponen una posible lista de estas condiciones, basándose en existencia de variaciones. Es decir, a la hora de identificar una determinada cualidad, influirá de manera significativa el ser capaces de identificar al mismo tiempo su opuesto. Estas condiciones son: contraste, separación, generalización y fusión.

Tomando a la vista como ejemplo, un niño no podrá diferenciar la necesidad de gafas si no experimenta la diferencia entre ver a través de ellas, o ver sin ellas (contraste); al ver que con gafas o sin ellas, el resto del mundo permanece igual, verá que la variación en la vista está separada del resto de propiedades (separación); también observará que el resultado de ver a través de las gafas no depende de si mira un objeto concreto u otro (generalización); por último, si realiza la acción de quitarse las gafas, experimentará un cambio en la visión, identificando que son dos sucesos relacionados (fusión).

Estos estudios sobre aprendizaje son relevantes para la educación en AOSE porque aportan conceptos que pueden enriquecer el material docente. En concreto, podemos utilizarlos para mejorar los ejemplos de diseños y programas que se utilicen en el aula.

2.4.1. Teorías de aprendizaje aplicadas a la ingeniería del software

En realidad, aunque no sea un objetivo y puede que no lo hagan de forma consciente, las alternativas encontradas sobre cómo presentar el contenido a los alumnos utilizan técnicas que se encuentran enmarcadas dentro de las distintas teorías de aprendizaje existentes. En [Navarro and Van Der Hoek, 2009], reflexionan sobre el papel de las teorías de aprendizaje en la educación dentro de la ingeniería del software. Durante el proceso, repasan los modelos y teorías que consideran más destacables, ya sea por su uso efectivo o por su potencial. Estos elementos destacables serían: *Learning by doing, Situated Learning, Keller's ARCS Motivation Theory, Model-Centered Instruction, Discovery Learning, Learning through Failure, Learning through Reflection* y *Elaboration*.

Donde Learning by doing, se refiere al aprendizaje activo, en el cual el alumno emplea los conocimientos en un proceso. Situated Learning se refiere a la utilización de situaciones reales para introducir más realismo en la experiencia. La teoría ARCS de Keller, concentra una experiencia docente efectiva a través de un estado de motivación, que se alcanza con el cumplimiento de cuatro requisitos: atención (fomentar el interés del alumno, de forma que se capture su atención), relevancia (el resultado o el proceso deben ser relevantes, a ser posible marcar un hito), confianza y satisfacción (al alumno se le deben plantear dificultades que, al superarlas influyan confianza y así le produzca una satisfacción). Model-Centered Instruction construye el proceso docente alrededor de un modelo que podrá ser de tres tipos: entorno, causa-efecto y rendimiento-humano. De esta forma la experiencia consistirá en explorar el modelo. Una aproximación similar se produce en Discovery Learning, donde se fomenta el descubrimiento de conceptos a partir del supuesto de que el conocimiento que se asienta mejor es el que descubre uno mismo. Learning through Failure propone que el alumno aprenderá más de sus errores que de otras experiencias, de forma que el proceso docente se configura de acuerdo a fomentar o incluso provocar los errores del alumno. Learning through Reflection incluye una etapa de reflexión a posteriori acerca de la experiencia docente, ya sea mediante un diálogo con el profesor o redactando un documento, donde vuelca sus impresiones. Elaboration sostiene que el aprendizaje se realiza mejor de forma incremental, y que por tanto el proceso docente se debe basar en este principio.

Siguiendo con su análisis, [Navarro and Van Der Hoek, 2009] identifican que la mayoría de las aproximaciones acerca de cómo plantear la experiencia docente de la ingeniería del software se pueden agrupar en tres grupos: realistas, temáticos y simulaciones.

- Las aproximaciones realistas incluyen nociones de realismo, como puede ser la introducción de situaciones reales dentro del desarrollo de un proyecto, como pueden ser los cambios en los requisitos por parte del cliente.
- Las temáticas se concentran en aspectos concretos que puedan no cubrirse en visiones más generales, como puede ser una metodología de desarrollo concreta. Precisamente porque se espera que este aspecto produzca una mejora del resultado.
- Las aproximaciones que utilizan simulaciones suelen considerar el desarrollo de un producto dentro de condiciones simuladas, siendo habituales que estén basadas en necesidades de la industria o el desarrollo de juegos.

Estas aproximaciones [Navarro and Van Der Hoek, 2009], de forma premeditada o no, consciente o no, utilizan una o más de estas teorías. Se pueden encontrar distintos ejemplos que confirmen este tipo de impresiones, sin ir más lejos, dado que al plantear aprendizaje basado en proyectos se está practicando Learning by doing. De manera más concreta, en el trabajo de los Claypool [Claypool and Claypool, 2005] encontramos que al proponer el desarrollo de un juego de ordenador para presentar los contenidos se practica Learning through Reflection y se cumplen los requisitos de Keller.

En la tabla 2.3, se pueden ver los trabajos estudiados en la sección 2.3.1 con las correspondencias que se identifican con las teorías de aprendizaje enumeradas en el trabajo de Navarro y van der Hoek. Respecto a los requisitos ARCS de Keller, solamente se han incluido en los trabajos que, explícitamente, hacen énfasis en captar el interés del alumno.

Autores	**Teorías de aprendizaje implicadas**
Gnatz et al	Learning by doing, Situated Learning
Pucher & Lehner	Learning by doing, ARCS
Claypool & Claypool	Learning by doing, Learning through Reflection, ARCS
Petkovic et al	Learning by doing, Situated Learning

Cuadro 2.3: *Teorías de aprendizaje detectadas en los trabajos revisados en la sección 2.3.1.*

Se observa que no se repite este método con la tabla 2.2 para incluir el método de aprendizaje. Se ha elegido omitir el método de aprendizaje debido a que los trabajos investigados en la sección 2.3.3 ni lo mencionan explícitamente ni aportan suficiente información para deducirlo (como ocurre en la tabla 2.3), salvo pinceladas concretas. Por ejemplo, en el trabajo de Hazzan y Dubinsky [Hazzan and Dubinsky, 2003], las precauciones que toman para adaptar el entorno al proceso, supone una práctica clara de *Situated Learning*. Precisamente porque no se menciona de forma explícita el uso de este tipo de métodos, este ejemplo reafirma la impresión de que la teoría de aprendizaje se presenta de forma inconsciente en los trabajos de educación en SE.

Finalmente es importante señalar que la utilización de las teorías docentes para el diseño de la experiencia educativa o una herramienta de soporte, no es el único campo de aportación posible. Como Navarro y van der Hoek afirman [Navarro and Van Der Hoek, 2009], son apropiadas también para tareas de evaluación, haciendo un uso retrospectivo de las mismas. Es decir, corroborar que las teorías de aprendizaje que se utilizan, sean efectivamente utilizadas.

2.4.2. Teorías de aprendizaje aplicadas al diseño de ejemplos

En [Atkinson et al., 2000], se revisan distintos trabajos destinados a la efectividad del uso de ejemplos. Identifican distintas factores que pueden mermar su efectividad, que clasifican en tres niveles: intra-ejemplo (factores internos al ejemplo), inter-ejemplo (entre ejemplos relativos a la misma lección), y a nivel de comprensión por parte del estudiante. Los factores intra-ejemplo influyen en cómo se diseña y se estructura el ejemplo.

En [Ward and Sweller, 1990], sus autores desarrollan una serie de experimentos relativos a medir la eficacia de los ejemplos. En esta ocasión, la teoría de carga cognitiva se utiliza en el diseño de los experimentos. Como material, utilizan ejemplos para mejorar el aprendizaje de problemas en diversas subáreas de la física (óptica geométrica, por ejemplo). Los autores exponen las resultados de sus experimentos, donde se observa un potencial de mejora mediante un uso extensivo de ejemplos, si se estructuran correctamente.

Dado que los ejemplos no suponen una mejora educativa en sí mismos, ni reducen la carga cognitiva por sí solos, estas características deberán introducirse en el ejemplo modificando su estructura. En su opinión [Ward and Sweller, 1990], estructurar correctamente un ejemplo se refiere a evitar dividir la atención del estudiante entre dos o más focos de información. En concreto, la atención se debe dirigir al estado del problema y a las posibles alternativas. Ward y Sweller, además, enumeran resultados obtenidos de la aplicación de ejemplos y sugerencias para el uso de estos:

1. Practicar con ejemplos obtiene mejores resultados que practicar con problemas.
2. No necesariamente cualquier ejemplo es efectivo.
3. Los efectos poco (o nada) efectivos pueden convertirse en efectivos modificando su estructura y eliminando la necesidad de dividir la atención por parte del estudiante.
4. Un exceso de información asociada a los ejemplos no solo no es necesario sino que puede resultar perjudicial. Los mejores ejemplos concentran su información de forma unitaria.

Respecto a la modificación de los ejemplos, introducir explicaciones textuales adicionales entre las subetapas del ejemplo, consigue reducir la división de la atención y produce una mejora.

Relativo al diseño y la estructura de los ejemplos, la integración de etapas y subobjetivos del ejemplo resultan de importancia. En [Catrambone and Holyoak, 1990], sus autores identifican que estos elementos se deben resaltar. Resaltar estos elementos puede incluso suponer presentarlos por separado.

2.4.3. Conclusiones

El estudio de teorías de aprendizaje por un lado avala el uso de perspectivas basadas en proyecto por su profunda relación con *Learning by doing*, y además apunta conclusiones respecto a los ejemplos educativos:

1. En la medida de lo posible, los ejemplos no deben segmentar la atención del alumno, por lo que el ejemplo debe tener cohesión y una estructura unitaria.

2. Los problemas complejos pueden no ser buenos ejemplos, y ser más efectivos al dividirse en ejemplos más pequeños que se estudien de forma aislada. Esto supone valorar la simplicidad.

3. Los recursos gráficos mejoran su efectividad si cuentan con apoyo textual, pero al mismo tiempo una sobrecarga de información puede ser perjudicial.

2.5. Programación en Ingeniería del Software

Trasladar una especificación a código es una actividad habitual en el desarrollo de sistemas. Pese a que existen esfuerzos para minimizas el espacio entre diseño e implementación, la implementación de manera manual es inherente al desarrollo de cualquier proyecto. Por este motivo, estudiar el aprendizaje de la programación tiene una relación importante respecto de la educación en SE, y por tanto con AOSE. Como durante el desarrollo de este trabajo no se ha encontrado evidencias de trabajos destinados al aprendizaje de lenguajes AOP, estudiaremos los trabajos acerca de la educación en programación en general, buscando trasladar las ventajas que se identifiquen a AOP.

La problemática acerca de cómo se debe enseñar a construir programas es tan antigua como los propios lenguajes de programación y se encuentran publicaciones al respecto desde los años 70. Por esto mismo, se revisan las impresiones dentro del campo de la enseñanza de la programación, recogiendo las conclusiones que se exponen en cuanto a las dificultades identificadas y sus soluciones. Por ejemplo, a menudo se tienden a señalar como dificultades: cómo afrontar un problema concreto, cómo encontrar bugs dentro del su propio código, etc.

Una dificultad importante en la docencia de la programación, es que programar no trata de obtener un producto de forma inmediata y además no es lineal (no podemos considerar una secuencia algorítmica de pasos por la cual obtener el resultado, y ni mucho menos esperar que este sea siempre óptimo). Para enseñar programación, parte de los objetivos a los que se debe aspirar [Bennedsen and Caspersen, 2008], es demostrar al alumno que:

- en el trascurso de la codificación, utilizar muchos pasos cortos es mejor que pocos pasos más largos.

- el resultado de cada paso debería probarse, porque el proceso de desarrollo es incremental.

- decisiones tomadas anteriormente a lo largo del proceso habitualmente tienen que replantearse y el código probablemente deba ser refactorizado.

- cometer errores es común, también para programadores experimentados.

- los errores de compilación pueden ser engañosos o erróneos, y que un código compile correctamente no es ninguna garantía de que sea correcto.

- la documentación online resulta muy valiosa, y a menudo es uno de los mayores apoyos durante un desarrollo.

- existe una forma sistemática (aunque no lineal) de desarrollar una solución para un problema dado.

Una filosofía muy extendida a la hora de presentar tecnología de desarrollo consiste en la construcción de aplicaciones Hello World. Estas aplicaciones, consisten en fragmentos muy sencillos de programa que buscan mostrar "Hello World!" (o textos similares) por pantalla. Naturalmente, el uso de estas aplicaciones de forma introductoria se utiliza también en el aprendizaje de programación. Mediante estos pequeños programas, se consigue que el alumno observe el programa en ejecución, y al ver el código en funcionamiento, se le invita a interactuar, dejándole en una situación de predisposición para modificarlo. El aspecto psicológico conseguido al poner código en funcionamiento será beneficioso.

Otra opción muy empleada para los primeros pasos será la utilización de código de ejemplo; como en [Kölling and Barnes, 2008], donde los autores afirman que una de las primeras tareas que debe afrontar el alumno, es examinar con detenimiento programas ya codificados (evidentemente, una vez familiarizados con aspectos como la sintaxis del lenguaje) con el fin de familiarizarse con ellos. Esto parece muy interesante como medio para evitar el estancamiento inicial de no saber por donde comenzar. Sin embargo, esta opción por si sola no será jamás suficiente, pues un código no deja de ser texto, y representar en texto el carácter dinámico de la programación o el comportamiento en ejecución de un programa no es algo muy viable.

Por eso, combinado con el conocimiento adquirido de un Hello World, donde es más importante la forma (como se ejecuta la aplicación) que el contenido (¡Hola Mundo!), el alumno, lanzando los ejemplos docentes, podrá adquirir conocimiento acerca del resultado en ejecución del código que ha inspeccionado.

Un problema notable, puede ser entender perfectamente qué hace el programa o entender qué pasa durante la ejecución de éste. Por este motivo, cobran fuerza las herramientas visuales, como Jeliot (que muestra de forma gráfica como un programa en Java es interpretado) o Alice (un entorno 3D para la docencia de programación).

En el ámbito de la enseñanza de la programación, existe un problema entorno al concepto de paradigma, que no se define claramente desde un primer momento. De este modo, la mayoría de cursos de introducción a la programación utilizan un paradigma orientado a objetos, a menudo sin siquiera comentar que existen otros. Otros cursos eligen utilizar un paradigma de programación funcional. El paradigma y el lenguaje elegidos sesgan la visión inicial que se tendrá del proceso de programación. Ocurrirá lo mismo al elegir, como es habitual, hacer énfasis en abstracciones o en componentes.

En [Lahtinen et al., 2005], los autores presentan un estudio aplicado a varias universidades, con el que buscan identificar las dificultades principales de los alumnos en los primeros cursos de programación, recogiendo las opiniones de profesores y alumnos. Resulta significativo que en lo relativo a los conceptos que presentan mayor dificultad a los alumnos, los profesores coinciden con los propios alumnos en señalar los siguientes: recursión, punteros y

referencias, tipos abstractos de datos, gestión de errores y la utilización de librerías. También se coinciden en señalar como tareas que presentan más problemas: entender cómo diseñar un programa para un problema concreto, dividir funcionalidad en subprogramas (funciones y procedimientos) y depurar su propio código.

Por último, se destaca que los alumnos parecen considerar a los ejemplos como el recurso docente más valorado. Este es un dato de mucha importancia, porque desarrollar el material para la asignatura supone una gran carga para el profesor y, a menudo, no se le da la importancia suficiente respecto a otros materiales en forma de apuntes, ejercicios o transparencias de clase.

2.5.1. Conclusiones

En el estudio de la educación en programación se han encontrado referencias positivas hacia el uso de ejemplos. Estas referencias, que provienen tanto de profesores como de alumnos, invitan a pensar que su uso debería ser más extensivo. Sin embargo, no se han encontrado referencias hacia el contenido de los ejemplos. Se entiende que consideran los ejemplos a incluir como algo a la elección del docente, pero no parece que planteen lo efectivos o no de estos ejemplos.

Sí que se hacen referencia al uso de herramientas visuales como Alice o Jeliot, lo que resalta la importancia de los componentes visuales para comprender los programas.

Respecto a las características que deban poseer estos ejemplos, al apuntar hacia programas estilo *Hello World!*, da a entender que es deseable contar con un ejemplo ejecutable y sencillo.

2.6. Conclusiones

En las conclusiones de la sección 2.2.3 se ha encontrado que mejorar la educación en la ingeniería del software puede suponer una mejora para la ingeniería del software. Por extensión, se puede reconocer que esta mejora puede tener ventajas para AOSE. También se ha podido confirmar espacio para la mejora mediante ejemplos. Se entiende que esta mejora es también trasladable a AOSE, donde no se ha encontrado que este problema se haya atendido.

En cuanto a los ejemplos, por las conclusiones de las secciones 2.2.3, 2.3.4, 2.5.1 y 2.1.7 observamos que será deseable que sea ejecutable y sencillo, no contenga más información de la necesaria, muestre cohesión en su estructura y cuente con un contexto definido. Hay más características candidatas, como se indica al final de esta memoria, pero se dejan para trabajo futuro.

Respecto al enfoque educativo, se ha visto en la sección 2.3.4 que presentar IS mediante proyectos obtiene buenos resultados, una práctica que también se observa levemente en AOSE (ver sección 2.1.7). Se puede opinar que esta práctica puede ser beneficiosa para AOSE, ya que parece serlo para IS.

La enseñanza de las metodologías IS no pueden compararse con AOSE porque no se han encontrado evidencias suficientes, pero sí se ha encontrado que factores como la codificación manual pueden suponer un obstáculo, por lo que utilizar desarrollo dirigido por modelos y generación automática de código parece razonable. Esto puede servir para concentrar la atención en los conceptos de la metodología. Por este motivo se considera INGENIAS como metodología objetivo para el diseño de los ejemplos, pues ofrece generación adecuada de código y amplia cobertura del ciclo de vida del proyecto (ver sección 2.1.7).

Por la sección 2.3.4 también se puede concluir que determinados aspectos de la metodología se observarán mejor por separado, pudiendo recibir ejemplos propios.

Capítulo 3

Contribución

Se ha observado en las secciones 2.2.3 y 2.3.4 que los ejemplos docentes pueden suponer una mejoría en el proceso de aprendizaje. Sin embargo, no se han encontrado trabajos que definan características específicas de un ejemplo educativo para SE ni para AOSE.

Definir las características deseables para que un ejemplo de una metodología AOSE sea efectivo conforma la contribución de este trabajo de investigación.

3.1. Ejemplos docentes.

Un ejemplo AOSE puede referirse a diferentes productos del proceso ingenieril, como una especificación, documentación o código.Esta distinción supone contemplar distintos tipos de ejemplo, y esta distinción afecta también a su complejidad.

En la sección 2.4, se indicaba que los autores recomiendan aplicar descomposición para mejorar el proceso cognitivo como paso previo a la asimilación de conocimiento complejo. Esto quiere decir que deberemos hacer lo posible para descomponer los ejemplos complejos. Para conseguir esto, se consideran los ejemplos de ámbito de proyecto o de ejemplos concretos. Tanto los primeros como los últimos podrán ser de mayor o menor escala.

Los ejemplos de metodologías AOSE se suelen presentar como casos de estudio o tutoriales, donde ambos suelen presentar ámbito de proyecto y poseer escala moderada. Estos ejemplos se limitan a mostrar las características de la metodología a la que pertenecen, a menudo de forma sesgada e incompleta. Aunque en estos recursos se observa cierto valor educativo, su función es más divulgativa que educativa. Como se ha visto en 2.3.4, no es válido ni efectivo cualquier ejemplo, incluso determinados ejemplos pueden llegar a ser perjudiciales.

Se ha podido ver en la sección 2.6 que es deseable que un ejemplo sea ejecutable y sencillo, no contenga más información de la necesaria, muestre cohesión en su estructura y cuente con un contexto definido. Estas características han sido extraídas del trabajo investigado, es decir, se deducen de la literatura. A estas condiciones, se les añade la existencia de aspectos específicos que tengan relación con los agentes software. Esto parece razonable, dado que el enfoque de este trabajo son los ejemplos AOSE. También se incluye la característica de

contar con un componente visual que permita apreciar la ejecución, una característica que se aprecia en la programación, como se ha destacado en la sección 2.5.1. Sin embargo esta característica parece razonable también para la ingeniería, y se propone incluirla. Se usa como hipótesis de trabajo que cumplir con estas condiciones hace un ejemplo más efectivo.

3.2. Características deseables.

El método para presentar las características es el siguiente: después de enumerar las características, se procede a profundizar en ellas de forma individual. Para profundizar en ellas, primero se justifican se forma general, y luego con respecto a los diseños. Se atienden los diseños explícitamente porque serán el medio de evaluación de las características en la sección 4. El motivo por el que se efectúa esta distinción, es porque presentar de forma general las características deseables en un ejemplo AOSE puede resultar interesante para su aplicación al diseño de ejemplos que estén orientados a una implementación rápida con lenguajes AOP. Así, se pueden enunciar una serie de características deseables que, de forma horizontal, se apliquen a todos los ejemplos AOSE. Como se ha enunciado, se combinan características deducidas del estudio del arte (ejecutable, concreto, cohesionado y con un contexto definido) con características que se infieren durante este trabajo (aspecto característico, componente visual).

La justificación del interés a estas características se incluye en las secciones siguientes.

3.2.1. Ejecutable.

Se ha visto que los ejemplos docentes son un recurso bien considerado por los alumnos en la sección 2.5 pág 34, y demandado por los docentes en la sección 2.2 pág 19. Dentro de estos ejemplos, la valoración de los alumnos es más positiva respecto a los ejemplos ejecutables que frente a los fragmentos de programas o a los diseños sin completar, como se menciona en la sección 2.5.1 pág 34.

De estas opiniones se puede deducir que genera más interés manejar una tecnología que funciona. Al mostrar un ejemplo capaz de lanzarse en ejecución, se invita al alumno a interactuar con el ejemplo, dado que el ejemplo puede tener como finalidad el enseñar la forma de poner en funcionamiento un sistema, integrar tecnologías o una característica metodológica concreta.

La capacidad de ser ejecutable es importante para los diseños. En la sección 2.1 se ha inspeccionado con detalle algunas metodologías orientadas a agentes, donde se puede encontrar que algunas no contemplan la implementación. Algunas como Prometheus, pese a que poseen una herramienta de generación de código, se limitan a la generación de templates. Esta generación no es suficiente para obtener un sistema ejecutable. En su lugar, se ha elegido utilizar la metodogía INGENIAS, cuya herramienta IDK necesita una intervención menor.

3.2.2. Componente visual.

En la sección 2.5.1 pág 34 se ha indicado la conveniencia de incorporar elementos visuales. Aunque no siempre será posible, un ejemplo se debería poder ejecutar de forma visual, dado que esto les hará más atractivos. Esto no necesariamente quiere decir que se pretenda utilizar siempre una interfaz gráfica, se puede considerar suficiente el uso de logs o mensajes por pantalla. Otra alternativa sería incorporar herramientas visuales de depuración o inspección, que reflejen lo que hace el ejemplo. Puede observarse que esta característica está muy relacionada con la capacidad del ejemplo para ser ejecutable.

Esta característica se menciona explícitamente porque lanzar a ejecución un programa o un sistema determinado, puede no contener ningún componente visual (no tiene por qué mostrar nada de forma textual o gráfica). Esta circunstancia no ayuda a que se comprenda lo que está haciendo o cómo lo hace (que es la finalidad del ejemplo), dado que se puede tratar de una ejecución transparente.

Respecto a los diseños, se puede argumentar que los diagramas y modelos que se producen como parte del proceso de diseño de un sistema multiagente siguiendo una metodología orientada a agentes, conforman un componente visual suficientemente sólido para comprender el sistema que se está diseñando. Sin embargo, esto no es así. Como una de las características principales de los agentes es su autonomía, prever el comportamiento del agente a lo largo de su vida dentro del sistema, no será algo trivial para el alumno novicio, ni siquiera en sistemas sencillos.

3.2.3. Aspecto característico de agentes software.

Identificar a los agentes con sus características es importante para que los ejemplos contribuyan a la familiarización del estudiante con AOSE. Por este motivo, no contemplan los ejemplos que no exploten las características relevantes de los agentes, del mismo modo que no podríamos considerar un buen ejemplo de orientación a objetos si no se utilizan clases e instancias, siendo el objetivo mostrar cómo funciona la herencia o el polimorfismo.

3.2.4. Concreción.

En las secciones 2.4 pág 31 y 2.4.3 pág 31, se ha identificado que el exceso de información en un ejemplo puede no mejorar su eficacia sino incluso resultar perjudicial. También en la sección 2.3.2 pág 23, se apuesta por que los ejemplos sean concretos. Por eso, si se propone un ejemplo que muestra una característica concreta de una metodología o de un lenguaje que se está presentando o, más generalmente, de los agentes software, el resto del ejemplo debe ser lo más escueto posible. Por tanto se trata de concreción a nivel de conceptos, resaltando el que se pretende presentar en el ejemplo.

A la hora de mostrar un aspecto o una característica, es preferible mantener la atención en ella. Esta atención se puede perder si se intenta mostrar un ejemplo que cuenta con varias características complejas, salvo que el propósito sea justamente mostrar cómo combinar

características de forma apropiada. Hablar de concreción es referirse a que un ejemplo debe evitar en la medida de lo posible el exceso de información.

Si se quieren evidenciar varias características, una alternativa sería usar una batería de ejemplos, mostrando cada característicade forma escalonada y donde los diseños de distintos ejemplos evolucionan haciendo énfasis en diferentes características según van creciendo. Por ejemplo, se podría considerar un diseño donde se muestran capacidades sociales en forma de interacción, al que después se le añade un mayor componente de autonomía, y finalmente se le incluye un componente de inteligencia.

3.2.5. Cohesión.

Es deseable que la estructura de un ejemplo sea compacta y los elementos (junto con sus relaciones) se presenten con claridad. Esto supone que no aparezcan elementos donde no se esperan o donde no exista relación clara y necesaria con el resto de elementos.

Se ha visto en la sección 2.3.4 pág 27 que este es un problema característico del desarrollo dirigido por modelos, pero realmente se puede considerar aplicable a cualquier ejemplo.

Al considerar la cohesión, se hace a nivel del diseño, no a nivel de modelo o diagrama. Como los elementos puede aparecer en distintos modelos, hay relaciones que se pueden establecer en más de un tipo de ellos. Podría optarse por definir cada elemento completamente a nivel de diagrama individual. Sin embargo esta no es la cohesión que se pretende, porque implica relaciones repetidas en cuanto el ejemplo considera más de un modelo o diagrama. Si que se podrá argumentar beneficioso replicar alguna relación en diagramas diferentes para ejemplos suficientemente complejos, si con ello se mejora la claridad.

Por tanto se valorará positivamente dentro de un diseño que la aparición de determinadas instancias se relacione de forma natural entre los diferentes diagramas y modelos.

3.2.6. Contexto definido.

En la medida de lo posible, se debe señalar qué se espera del ejemplo, cuál es su objetivo.

Al examinar un ejemplo, será importante conocer qué pretende hacer, o qué tarea realiza. Se ha visto en la sección 2.2.3 pág 19, la utilización de ejemplos que "sirven para algo", favorecen el ser recordados posteriormente y sirven de herramienta de consulta, consiguiendo reusabilidad. En este sentido, trabajar ejemplos utilizando casos de estudio puede ser conveniente, porque tradicionalmente tienen un propósito y un trasfondo bien definidos.

En cuanto a los diseños, respecto al contexto no se aprecian consideraciones adicionales que se deban aplicar exclusivamente a los diseños.

3.2.7. Conclusiones.

En definitiva, las características principales que se han identificado como deseables son las siguientes:

1. Que sea ejecutable.

2. Tendrán que contar con algún componente visual.

3. Deberán contener algún aspecto característico de los agentes software.

4. Preferiblemente, debe ser concreto.

5. Debe tener cohesión.

6. Debe contar con un contexto definido.

Se sigue asumiendo la hipótesis de trabajo de que un buen ejemplo docente para AOSE es aquel que se puede mostrar en ejecución, exhibe una característica concreta o cuenta con un conjunto cohesionado de elementos, y cuenta con un contexto definido que presenta su función.

Capítulo 4

Evaluación de las características

Tras exponer las características relevantes identificadas para los ejemplos educativos en AOSE, se aplican estas características en la elaboración de ejemplos concretos de la metodología INGENIAS. En la tabla 4.1 se puede ver un cuadro resumen de la notación utilizada en los diagramas pertenecientes a los ejemplos.

Cada ejemplo tiene la misma estructura: primero se procede a introducir una descripción del ejemplo incluyendo su desarrollo y finalmente se expone la evaluación del ejemplo a partir de las características.

Es importante señalar que el desarrollo textual del ejemplo (su explicación) forma parte del mismo. Como se apunta en la sección 2.4.3, la eficacia de los recursos gráficos (en este caso los diagramas que pertenecen al ejemplo) mejora con la inclusión de texto que describa tales elementos. Es decir, si se incluye la explicación, la eficacia del ejemplo mejora a los modelos y diagramas por si solos. Por eso, se considera el desarrollo como parte del ejemplo.

Para presentar los ejemplos, se opta por hacerlo de manera estructurada. Esto facilitará el que cada elemento se vea con claridad. Para estructurar la presentación de los ejemplos se propone el siguiente método:

1. Comenzar por una descripción general del ejemplo.
2. Presentar los modelos de forma secuencial e individual según convenga al propósito del ejemplo.
3. Describir los diagramas adicionales necesarios para diseño.
4. Exponer la salida del sistema.

Antes de proceder a la presentación de los ejemplos, se reconoce que los ejemplos propuestos son preliminares. Esto provoca que la evaluación de las características, como se verá, no siempre es satisfactoria, lo que indica que los ejemplos deben ser mejorados.

Notación	Descripción
	Representa un agente.
	Representa un rol que juega el agente.
	Representa una tarea que desempeña el agente.
	Representa un objetivo que persigue el agente.
	Representa un *FrameFact*, un hecho.
	Representa un estado mental del agente.
	Representa una interacción entre agentes.
	Representa una interacción en ejecución.
	Representa una unidad de interacción.
	Representa una aplicación interna.
	Representa un *INGENIASComponent*.
	Representa un *INGENIASCodeComponent*.

Cuadro 4.1: *Fragmento de la notación de INGENIAS que se utiliza en los ejemplos.*

4.1. Conversación básica entre agentes software.

Como primer ejemplo, se propone una conversación entre dos agentes. Este tipo de interacción básica, se presenta como una de las alternativas de plantilla para los proyectos desarrollados con la herramienta IDK. Inspirándose en esta plantilla, se elige modificarla y elaborar una versión con menos elementos. El motivo para no presentar la versión original y construir una propia, es que para mostrar una interacción se proporciona más información

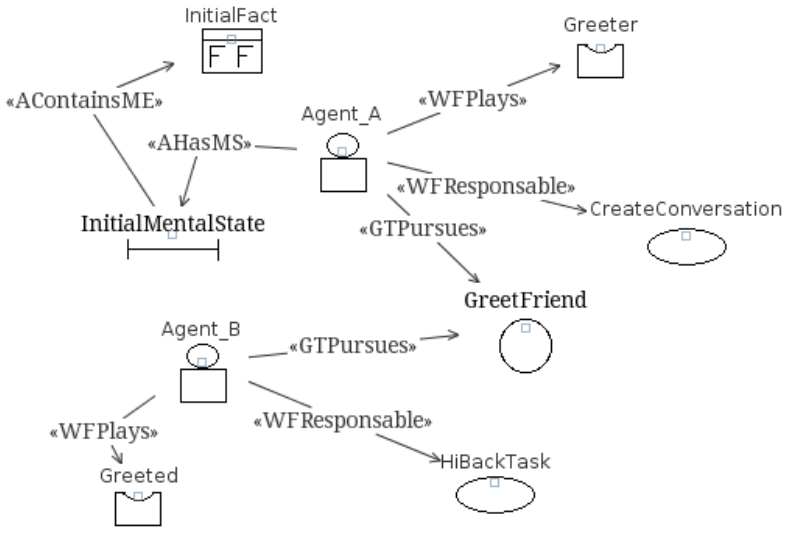

Figura 4.1: *Diagrama de agente del ejemplo Conversación básica entre agentes software.*

de la necesaria.

La conversación que se produce entre los agentes, es sencilla y consiste en que se saluden entre ellos: el agente que inicia la conversación (Agent_A) saluda, y el destinatario devuelve el saludo (Agent_B).

Para conseguir esto, se necesitan cuatro diagramas, a saber: un diagrama de agente, un diagrama de objetivos y tareas, un diagrama de interacción y un diagrama de componentes.

Diagrama de agente.

El diagrama de agente, define las clases de agentes que se encuentra en el sistema, como indica la figura 4.1. Dentro de esta especificación, se encuentran los objetivos que persigue, las tareas de las que es responsable, los roles que representa dentro del sistema y el estado mental inicial con que se desplegará. Para este ejemplo, se consideran dos clases de agentes, que jugarán dos roles distintos. Dado que para iniciar una conversación es necesario ejecutar una tarea (en este caso la tarea *CreateConversation*), se puede incluir la entrada de esta en el estado mental inicial del Agente_A, de forma que se usará como desencadenante de la conversación. En este caso, los agentes tan sólo contemplan un objetivo (el objetivo *GreetFriend*), que representa la intención de saludarse mutuamente.

Al utilizar agentes software es frecuente utilizar interacciones de algún tipo. Para definir estas, debemos incluir los roles que juegan los agentes. Si un agente no juega ningún rol, no podrá participar en ninguna interacción. También viene bien usar roles para aglutinar la funcionalidad que se supone de un agente que participa en una o varias interacciones. En este caso, ver figura 4.1, los roles son *Greeter* y *Greeted*. *Greeter* se encarga de iniciar la conversación, mientras que *Greeted* responde al saludo.

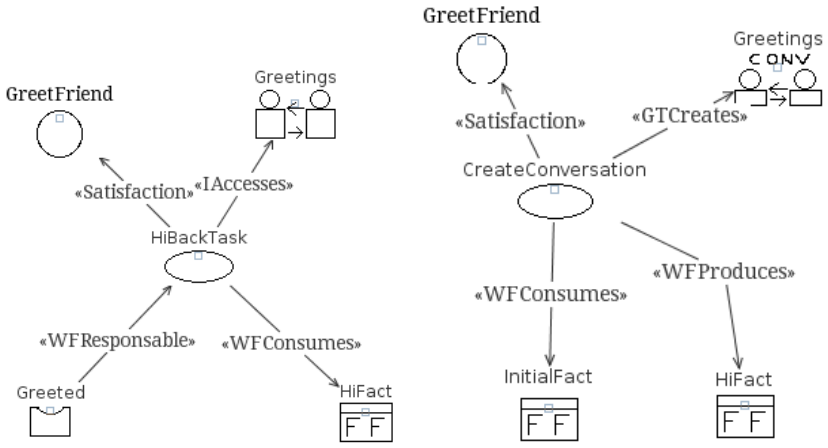

Figura 4.2: *Diagrama de objetivos y tareas del ejemplo Conversación básica entre agentes software.*

Diagrama de objetivos y tareas.

El diagrama de objetivos y tareas, ver figura 4.2, define la entrada, la salida, el objetivo, las interacciones con las que se relacionan las tareas y la jerarquía entre los objetivos. Como en este caso sólo se cuenta con un objetivo (*GreetFriend*), estos elementos quedarán en segundo plano. Como se ve en el diagrama, las tareas también pueden crear conversaciones. En INGENIAS, una conversación es una instancia de interacción en ejecución. Para que una tarea se encuentre bien definida, se debe especificar su responsable, sus datos de entrada, y su objetivo asociado, siendo elementos accesorios los datos de salida o su relación con conversaciones (una tarea no tiene obligación de crear conversaciones).

En INGENIAS, un mismo elemento puede aparecer en varios modelos y diagramas, y el conjunto de elementos que componen su definición puede no encontrarse concentrado en un mismo diagrama. Por ejemplo, en la tarea *CreateConversation* vemos que no se ha asignado un responsable, y sin embargo esto ya se ha definido en el diagrama anterior de la figura 4.1.

Respecto a los datos de entrada de las tareas, vemos que *CreateConversation* consume el hecho inicial que se deposita de inicio para que actúe como desencadenante. Este hecho inicial será imprescindible, dado que para que una tarea se ejecute, primero se debe planificar como disponible para su ejecución, y una tarea sólo se planificará para ejecución si cumple dos condiciones, tener listos sus datos de entrada y tener una relación de satisfacción con alguno de los objetivos que persigue el agente.

Se puede observar también que la tarea que comienza la conversación y realiza el saludo, también genera unos datos de salida que serán usados como datos de entrada por la tarea encargada de responder.

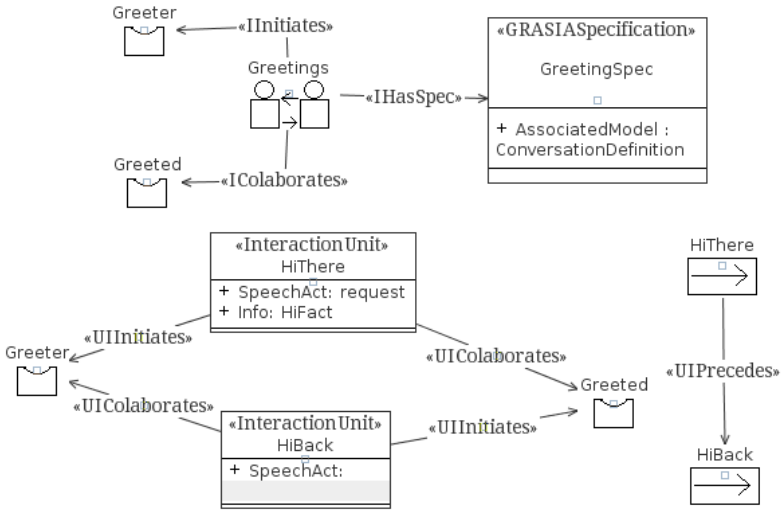

Figura 4.3: *Diagrama de interacción del ejemplo Conversación básica entre agentes software.*

Diagrama de interacción.

En el diagrama de interacción, ver figura 4.3 se ve cómo se define la interacción de la que es instancia la conversación que tiene lugar entre los dos agentes. Por un lado, se encuentra la definición abstracta de la interacción, que incluye los roles involucrados y la especificación del protocolo que implementa la interacción. Respecto a los roles involucrados, sólo uno de ellos debe ser quien comience la interacción, es decir, el iniciador. A su vez debe de haber al menos un colaborador. Se debe determinar el protocolo que especifica la interacción. INGENIAS incluye elementos propios para especificar una interacción, en forma de diagramas GRASIA!.

Estos diagramas tienen como elemento fundamental la unidad de interacción, la *InteractionUnit* que representan el acto de intercambiar información de alguna forma entre dos actores. Estos diagramas se incluyen dentro de los modelos de interacción, y se pueden componer por separado (en otro diagrama) o dentro del propio diagrama que define la interacción. Este es este caso, donde el diagrama que define la especificación de la interacción, es el mismo diagrama que la enuncia de forma abstracta.

Debido a que la conversación entre los dos agentes es muy sencilla, tan sólo se necesitan dos unidades de interacción para definir el protocolo que seguirá dicha conversación. Para que el protocolo sea correcto, a cada unidad de interacción se le debe asociar un rol iniciador y una serie de colaboradores. Como esta información no es suficiente para definir de forma unívoca un orden determinado, se impone dicho orden mediante relaciones de precedencia entre las unidades de interacción.

A las unidades de interacción se les puede especificar también una acción del habla (*SpeechAct*) dentro de los posibles contemplados por el estándar FIPA. Además, las unidades de interacción puede contener información que será intercambiada entre los agentes. Esta

es una propiedad muy importante. Anteriormente se ha visto visto que el *Agent_A*, al ejecutar la tarea *CreateConversation*, produce el hecho *HiFact*. Este hecho será usado por *Agent_B* para ejecutar la tarea que representa la respuesta al saludo. Sin embargo, para que esto sea posible, debemos hacer llegar a *Agent_B*, los datos producidos por *Agent_A*. Para conseguirlo, se adjuntan estos datos en la unidad de interacción, que actúa de transporte. Una vez transportados, los datos estarán disponibles para la tarea *HiBackTask*, que se planifica y se ejecuta.

Diagramas adicionales.

El diagrama de componentes asocia las tareas a un componente de código, un *INGENIASCodeComponent*. Una tarea puede asociarse con tan sólo un único componente de código. Esta relación es fundamental porque sin ella no se podrá generar el código que implementa la tarea. Como se ve en el diagrama de componentes de la figura 4.4, en estos componentes se puede añadir el código que se desea que ejecute la tarea. El código que se introduce dentro de los componentes se lleva directamente al conjunto de clases que implementan esta especificación, reemplazando lo que hubiera antes.

En el caso de este ejemplo, las tareas imprimen los respectivos saludos de los agentes por pantalla. En la figura 4.4 se encuentra el diagrama de componentes utilizado. También se incluye en la figura los valores que toman los campos de los *INGENIASCodeComponent* que se utilizan.

Salida del sistema.

Se observa en la figura 4.5 que la salida del sistema muestra lo que se pretende, los dos agentes interaccionan saludando cordialmente. Como alternativa para inspeccionar la interacción, se puede utilizar el sniffer de JADE. En la figura 4.6, se puede ver la traza de la conversación.

4.1.1. Evaluación del ejemplo.

Por último, se evalúa el ejemplo en función de las características:

- Ejecutable: El ejemplo es ejecutable, dado que utilizando la herramienta IDK y la especificación presentada en el ejemplo es suficiente para generar el código de su implementación. Es decir, la información presentada es suficiente para reproducir completamente el ejemplo. También se ha propuesto una ejecución alternativa que representa de forma más interesante la comunicación de los agentes.

- Componente Visual: El componente utilizado consiste en mensajes por consola. Estos mensajes ayudan a confirmar que se produce la interacción, que se intercambia la información pretendida y que se ejecutan las tareas diseñadas. Aunque sea un mecanismo rudimentario, puesto que se ejecutan únicamente dos tareas durante el funcionamiento del ejemplo, puede considerarse suficiente.

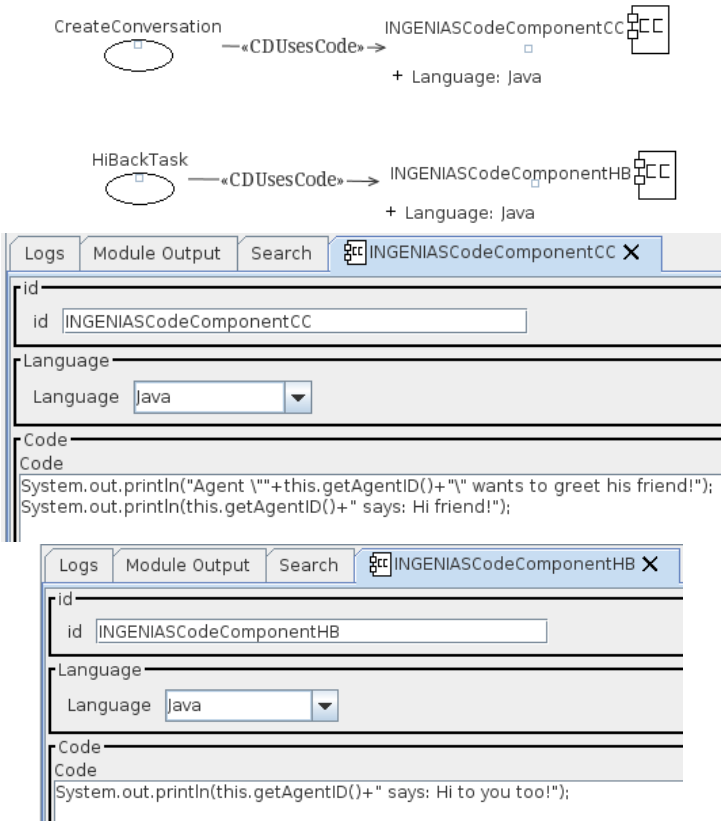

Figura 4.4: *Diagrama de componentes del ejemplo Conversación básica entre agentes software.*

```
[java] Agent "Agent_A" wants to greet his friend!
[java] Agent_A says: Hi friend!
[java] Agent_B says: Hi to you too!
```

Figura 4.5: *Salida por pantalla del ejemplo Conversación básica entre agentes software.*

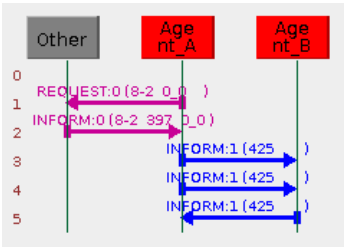

Figura 4.6: *Traza de la conversación en el sniffer de JADE.*

Ejecutable	Componente Visual	Aspecto característico	Concreción	Cohesión	Contexto
Sí	Consola	Interacción	Sí*	Sí	Lanzamiento de una interacción

Cuadro 4.2: *Tabla resumen de la evaluación del ejemplo Conversación básica entre agentes software.*

- Aspecto característico: Se trata de estudiar la forma de realizar una interacción. El aspecto característico se puede identificar claramente porque el conjunto del ejemplo gira alrededor de una interacción entre agentes, incluyendo de los elementos necesarios para definirla y ejecutarla.
- Concreción: se puede decir que el ejemplo es concreto porque en el diseño se muestran los mínimos elementos para proporcionar funcionalidad. Sin embargo, evaluar la concreción del texto es más difícil. Si se elige el ejemplo para ser uno de los primeros en proporcionarse al alumno, el nivel de explicaciones puede considerarse adecuado. Si no es así, hay un riesgo elevado de que contenga demasiadas explicaciones.
- Cohesión: se considera razonable argumentar que el ejemplo demuestra cohesión. Esto se entiende así porque se representan todos los elementos necesarios exclusivamente para el objeto del ejemplo. Además cada elemento que forma parte del diseño aparece en cada diagrama donde su presencia tiene sentido. Como el ejemplo es sencillo, no hay necesidad de incluir más relaciones para aumentar la claridad de este.
- Contexto: Interacción básica entre agentes. Un par de agentes tiene como objetivo intercambiar información, en este caso, un saludo. En pocas palabras, se puede decir que su contexto es demostrar un forma simple de función social.

La tabla 4.2 muestra un resumen de la evaluación (los asteriscos en la tabla significa que la valoración requiere argumentación).

4.2. Despliegue de un sistema de agentes homogéneos.

Se identifica como un aspecto característico de los agentes software el hecho de que no suelen realizar sus funciones en solitario, sino cooperando, formando un sistema multiagente. En este ejemplo, se muestra cómo desplegar un sistema multiagente de agentes homogéneos, es decir, donde todos los agentes son de la misma clase de agente. El objetivo de estos agentes es cooperar intercambiando cierta información con el agente que la solicite.

Para diseñar la vista del sistema que representa el ejemplo se utilizan seis diagramas: diagrama de agente, diagrama de objetivos y tareas, diagrama de entorno, diagrama de

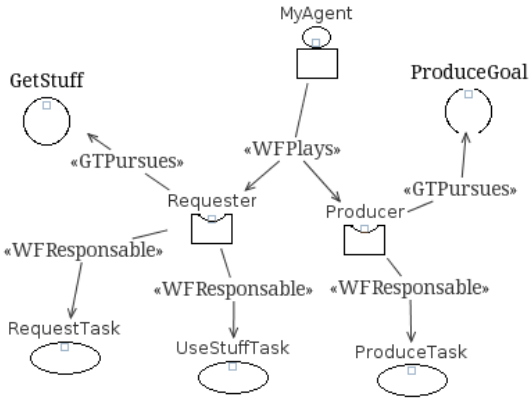

Figura 4.7: *Diagrama de agente del ejemplo Despliegue de un sistema de agentes homogéneos.*

interacción, diagrama de componentes y diagrama de despliegue. Finalmente, se evalúa el ejemplo respecto a las características de la sección 3.2.

Diagrama de agente.

En la figura 4.7 se observa como el diagrama de agente define la clase de agente que poblará el sistema. Como el propósito es hacer un sistema homogéneo, tan sólo se define una clase. Como se desea que los agentes cooperen, deberán definirse al menos dos roles. Estos roles tendrán objetivos y tareas asociados diferentes. Por una parte se encuentra el conjunto relacionado con la solicitud y ejecución de los datos que el agente demanda, y por el otro el relacionado con producir estos datos.

Diagrama de objetivos y tareas.

La figura 4.8 se puede ver el diagrama de objetivos y tareas. En él se observa cómo se diseña la relación entre las tareas y los objetivos, donde cada una satisface sus respectivos de acuerdo con el rol responsable. La tarea *RequestTask* instancia una determinada conversación, y a la vez esa misma tarea consume los eventos recibidos por la percepción que más adelante se verá asignada al agente. Las otras dos tareas irán consumiendo la información entrante y produciendo otra nueva, mientras van mostrando mensajes por consola. Además, se define el acceso de estas tareas al tipo de interacción que instancia la conversación, dado que de otro modo no podrían acceder a los contenidos que se intercambian.

Diagrama de entorno.

En la figura 4.9 se encuentra el diagrama de entorno. En el diagrama de entorno se encuentra la aplicación que modela la percepción del agente. Por una parte se define la

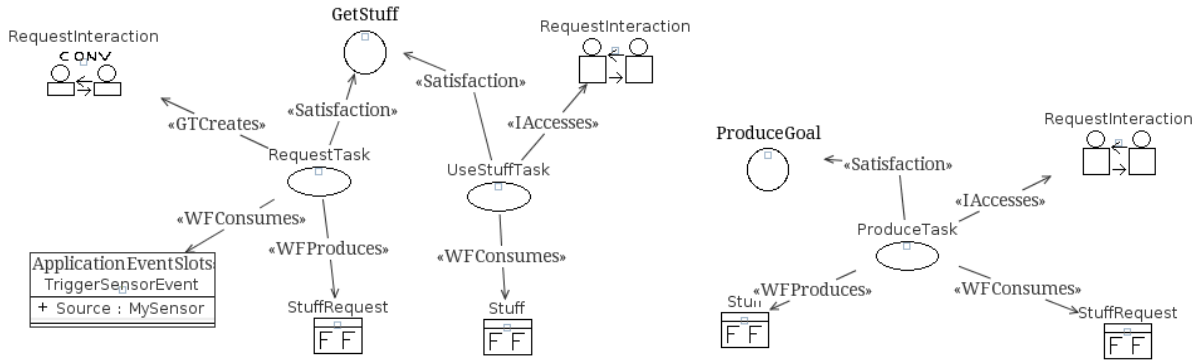

Figura 4.8: *Diagrama de objetivos y tareas del ejemplo Despliegue de un sistema de agentes homogéneos.*

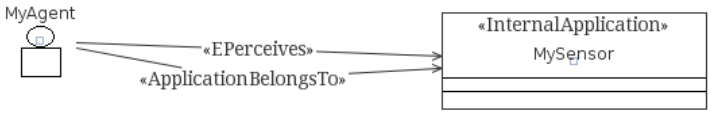

Figura 4.9: *Diagrama de entorno del ejemplo Despliegue de un sistema de agentes homogéneos.*

relación de propiedad necesaria para tener acceso a ella, y por otro se define la forma en que se percibe la información, que en este caso es en forma de "ApplicationEventSlots".

Diagrama de interacción.

En el diagrama de interacción, que se puede ver en la figura 4.10, se encuentra la definición abstracta de la interacción que tendrá lugar dentro del sistema, junto con la especificación del protocolo que seguirá, que entre otras cosas incluye la información que se intercambiará. Al ser una interacción con un número de agentes implicados arbitrario, es necesario considerar la cardinalidad. La cardinalidad define el número de roles que participan en la interacción, siendo esta 1, 0..* o 1..*. Al especificar la cardinalidad en 1..*, se convierte en participantes de la interacción a todos los agentes que puedan representar el rol *Producer*, siendo al menos uno. Este nivel de difusión del mensaje es justamente lo que se pretende. La cardinalidad no se aplica al rol iniciador de la interacción porque sólo puede ser uno.

Diagramas adicionales.

Dentro del diagrama de componentes de la figura 4.11, se relacionan las tareas con el código que ejecutarán, y también la aplicación con un componente *INGENIASComponent*. Esta relación se necesita porque las aplicaciones se enlazan con ficheros que representan su interfaz java.

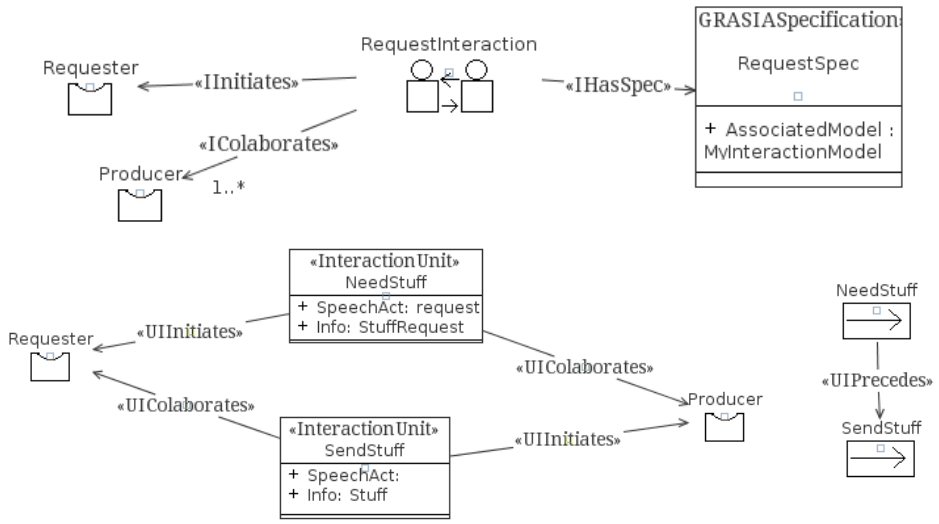

Figura 4.10: *Diagrama de interacción del ejemplo Despliegue de un sistema de agentes homogéneos.*

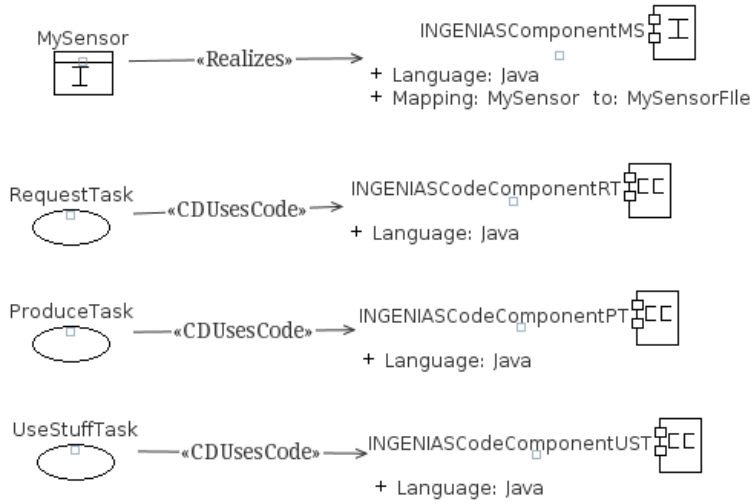

Figura 4.11: *Diagrama de componentes del ejemplo Despliegue de un sistema de agentes homogéneos.*

Figura 4.12: *Diagrama de despliegue del ejemplo Despliegue de un sistema de agentes homogéneos.*

```
[java] MyAgent_3DeploymentUnitByType says: My sensor informs me that I need some stuff
[java] MyAgent_3DeploymentUnitByType says: Anyone can help me here?
[java] MyAgent_3DeploymentUnitByType says: Sure! Happy to help!
[java] MyAgent_0DeploymentUnitByType says: Sure! Happy to help!
[java] MyAgent_4DeploymentUnitByType says: Sure! Happy to help!
[java] MyAgent_1DeploymentUnitByType says: Sure! Happy to help!
[java] MyAgent_2DeploymentUnitByType says: Sure! Happy to help!
[java] MyAgent_3DeploymentUnitByType says: Thanks that was usefull!
[java] MyAgent_3DeploymentUnitByType says: Thanks that was usefull!
[java] MyAgent_3DeploymentUnitByType says: Thanks that was usefull!
[java] MyAgent_3DeploymentUnitByType says: Thanks that was usefull!
[java] MyAgent_3DeploymentUnitByType says: Thanks that was usefull!
```

Figura 4.13: *Salida del ejemplo Despliegue de un sistema de agentes homogéneos.*

Finalmente, se introduce el diagrama de despliegue, que se puede ver en la figura 4.12. En este diagrama se pueden incluir distintas configuraciones con las que desplegar el sistema. En este caso, se elige un diagrama de despliegue *UnitByType* que sencillamente genera un número a determinar de instancias del tipo indicado. Para el caso del ejemplo, se elige instanciar cinco veces el agente.

Salida del sistema.

Se observa en la figura 4.13 la salida del sistema, donde se encuentra un aspecto interesante derivado de las decisiones de diseño. En concreto, se ve que el agente que solicita datos, también responde a la petición.

Esto sucede porque en INGENIAS un mismo agente puede jugar varios roles en una misma interacción. De esta forma, nuestra decisión de formar un sistema multiagente poblado por agentes homogéneos, unida a la cardinalidad de la interacción (todos los agentes que puedan jugar ese rol), tienen como consecuencia que el mismo agente que inicia la conversación, responde también a la petición, dado que es capaz de representar los dos roles. Respecto a los mensajes que se muestran, la última batería de mensajes por consola los produce la tarea que procesa la información recibida. Otra vez, dado que el agente ha participado como productor, se procesan los datos tantas veces como se envían, cinco veces, una por cada agente que la envía, incluido el iniciador.

4.2.1. Evaluación del ejemplo.

Por último, se evalúa el ejemplo en función de las características:

- Ejecutable: el ejemplo está pensado para ejecutarse en modo depuración desde IDK. IDK incluye una interfaz de depuración que permite tener acceso a diferentes logs, el estado mental de los agentes y a las aplicaciones. En concreto, este modo depuración permite generar eventos bajo demanda, de modo que la ejecución está más controlada que si hiciera de forma automática. También hace que la legibilidad del ejemplo sea mayor, y su ejecución no suponga un violento spam de mensajes.

- Componente Visual: El componente visual es mixto. Por una parte, la ejecución de tareas producen mensajes por consola. Por otra, se utiliza la vista de aplicación del sistema generado en modo depuración para provocar la generación de eventos, interactuando así con el agente.

- Aspecto característico: En este caso se muestra una interacción entre un número más grande de agentes, formando un pequeño sistema multiagente. Esto permite observar cómo cooperan intercambiando información sencilla dentro de un despliegue más extenso. También se incluye percepción por parte de los agentes.

- Concreción: La concreción del ejemplo se resiente por culpa del planteamiento. Al proponer un despliegue de un sistema homogéneo, se consigue aumentar la concreción del diagrama de agente, sin embargo, esta decisión lleva a que finalmente se incluya un diagrama de entorno para que no se pierda el control del sistema. Es decir, finalmente no hay un elemento central y concreto en el ejemplo. Por un lado está una interacción sencilla como la del ejemplo de la sección 4.1, a la que se añade percepción de los agentes y la semántica de un despliegue, aspectos razonablemente sencillos pero que obligan a dividir la atención del lector.

- Cohesión: Los elementos se relacionan con aceptable claridad, sobretodo gracias al hecho de que son pocos. Sin embargo, durante la explicación, no se incluye ninguna referencia hacia las relaciones de responsabilidad que existen entre las tareas y los roles, ni se explica qué quiere decir que un rol sea el responsable de una tarea. Entender las relaciones que se encuentra dentro de un diagrama es un requisito para entender cómo se relacionan los distintos diagramas, y las distintas dependencias que puedan existir.

- Contexto: El contexto del ejemplo no queda del todo claro durante la explicación. La intención del ejemplo es diseñar el despliegue de un sistema multiagente, donde sus agentes son homogéneos e interactúan en una relación de uno a muchos. Aún así, se le puede hacer una crítica importante al ejemplo. Un ejemplo como este puede servir como oportunidad para mostrar lo rápidamente que se puede descontrolar un sistema multiagente, donde diferentes entidades actúan de forma autónoma. Esto se podría diseñar eligiendo que, en una versión inicial del ejemplo, los agentes dispararan la interacción mediante un estado mental inicial, como en el ejemplo de la sección

Ejecutable	Componente Visual	Aspecto característico	Concreción	Cohesión	Contexto
Sí*	Consola*	Interacción / Percepción	Poca	Sí*	Despliegue SMA

Cuadro 4.3: *Tabla resumen de la evaluación del ejemplo Despliegue de un sistema de agentes homogéneos.*

4.1. Posteriormente se podría haber presentado la opción de dotar a los agentes de percepción.

La tabla 4.3 muestra un resumen de la evaluación (los asteriscos en la tabla significa que la valoración requiere argumentación).

4.3. Mediación simple de un agente.

Las interacciones que ocurren dentro de un sistema multiagente no siempre tienen forma de conexión entre dos únicos pares. Una interacción habitual que no cumple este patrón, es la mediación entre agentes.

Para mostrar un ejemplo sencillo de mediación, se contempla un escenario en el cual se produce una interacción de tipo productor-consumidor a través de un intermediario. Es decir, el agente Agent C (C de cliente) solicita un determinado servicio al agente Agent M (M de mediador), quien evalúa la petición y selecciona un agente Agent S (S de servicio) para atender la solicitud, o responda que no es posible.

Es evidente que este escenario contempla muchas variaciones naturales y se abstrae de la selección del agente S por parte de M. Estas decisiones se toman por motivos de simplicidad.

Aunque, como se intuye, el interés del ejemplo se concentra alrededor del diagrama de interacción que define la mediación entre los agentes, se prefiere comenzar examinando el diagrama de agentes, donde se recogen los elementos principales sobre los que después se construye el resto del ejemplo. Posteriormente se trata el diagrama de tareas y objetivos, para finalmente inspeccionar el diagrama de interacción. Tras lo cual se tratan los diagramas adicionales y la salida del sistema.

Diagrama de agente.

En el diagrama de agente de la figura 4.14 se encuentran las clases de agente mencionadas en el enunciado del escenario. Estos son: el agente cliente (*Agent C*), el agente mediador (*Agent M*) y el agente servidor (*Agent S*). Se observan también sus roles unívocamente determinados, de forma que se aumenta la simplicidad. Como desencadenante de la interacción, en esta ocasión se opta por la inserción de un estado mental en el agente cliente, de forma que todo el proceso ejecute de forma automática.

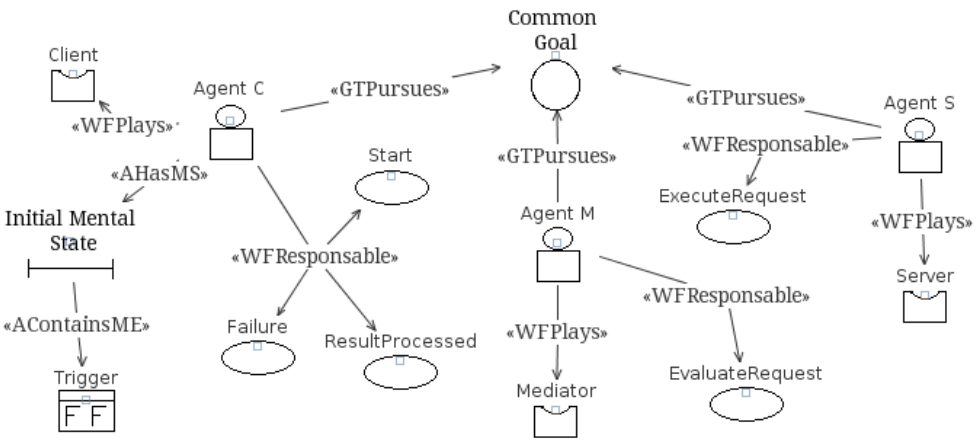

Figura 4.14: *Diagrama de agente del ejemplo Mediación simple de un agente.*

Diagrama de objetivos y tareas.

Las diferentes tareas que se adelantan en el diagrama de agentes, manejan la producción de información que más tarde se propaga a través de la interacción. La relación entre las tareas y esta información se diseña en el diagrama de objetivos y tareas, que se puede ver en la figura 4.15. Este diagrama siempre es interesante porque deja constancia de en qué punto se genera la información. Por ejemplo, se elige diseñar la toma de decisiones por parte del agente M mediante la ejecución de la tarea *EvaluateRequest*. Esta tarea tiene como salida dos Frame Facts que se insertan más tarde en dos mensajes distintos.

Diagrama de interacción.

En la figura 4.16 se puede encontrar el diagrama de interacción. En él, se ve el conjunto de mensajes que conforman el protocolo propuesto, junto con la información que se transporta en ellos. Como novedad respecto a los ejemplos 4.1 y 4.2, se puede observar que el rol *Mediator* envía dos mensajes al mismo tiempo. Este comportamiento se introduce para representar las distintas posibilidades que se contemplan en el escenario. Es decir, continuar con el proceso o comunicar al rol iniciador que la petición no se podrá atender.

Dentro de estos mensajes, se introduce la información que se generaba anteriormente mediante la tarea *EvaluateRequest*: *ClientRequestInfo* y *NoServerAvailable*. El mecanismo para forzar la elección de ramas, será la definición de timeouts en interacciones (en este caso 100), como se ve en el diagrama sobre la arista correspondiente al lado iniciador de la *InteractionUnit RefuseRequest*. De esta forma, al terminarse el tiempo, la rama que contiene esa unidad de interacción se considerará abortada, consiguiendo la disyuntiva que se pretende.

Se puede ver en las relaciones de precedencia que se encuentran entre las *InteractionUnit* la secuencia que forma el protocolo . En la rama que continúa con el protocolo, la ejecución llega al rol servidor. El agente ejecutará la petición y obtendrá unos resultados encapsulados en *ExecutionResult*. Posteriormente, estos resultados son enviados al rol *Mediator*, que más

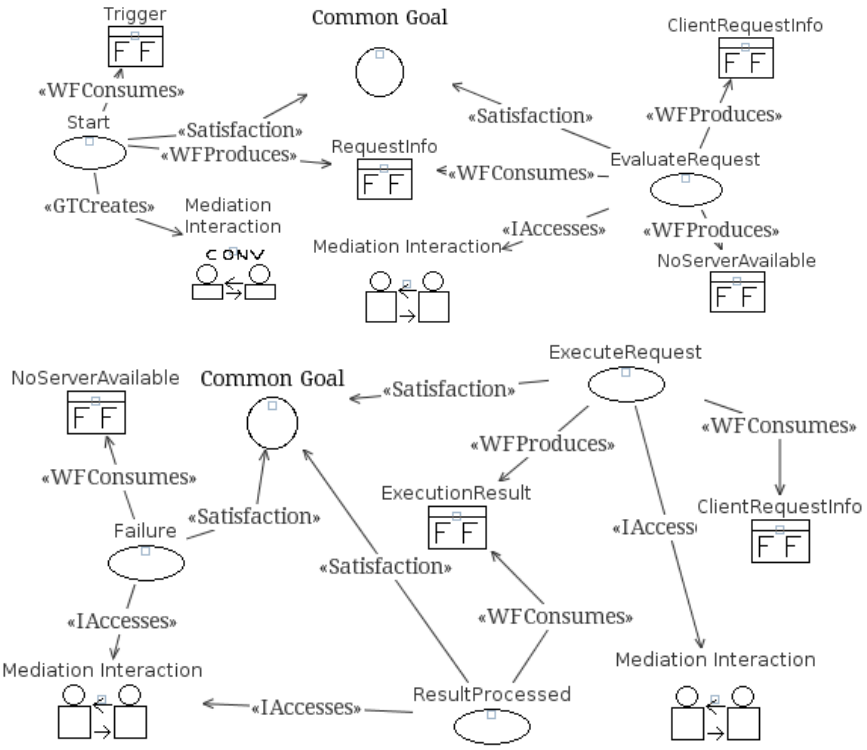

Figura 4.15: *Diagrama de objetivos y tareas del ejemplo Mediación simple de un agente.*

tarde se los reenvía al rol *Client*.

Diagramas adicionales

Como diagrama adicional es suficiente con utilizar un diagrama de componentes. Este diagrama de componentes tan sólo incluye las tareas con sus componentes de código asociados. No se incluye el diagrama por claridad. Se puede encontrar descrito un diagrama similar al utilizado en el ejemplo 4.1.

Salida del sistema

El sistema en ejecución sencillamente muestra la traza de invocaciones de las tareas. La salida del sistema se puede ver en la figura 4.17. Se observa que en primer lugar el agente cliente envía su petición al agente mediador, que evalúa la petición. El agente mediador, traslada la petición al agente servidor, que atiende la petición. Cuando el agente servidor obtiene los resultados, los envía al agente mediador, que sin aplicarles ninguna manipulación, los reenvía al agente cliente. Finalmente, el agente cliente anuncia que ha procesado la respuesta.

4.3.1. Evaluación del ejemplo.

Por último, se evalúa el ejemplo en función de las características:

- Ejecutable: Este ejemplo se puede ejecutar de la manera habitual. En cuanto al modo de ejecución, puede ser preferible utilizar el modo depuración, con el objetivo de observar mejor la dinámica de los agentes. En el caso de las capturas del ejemplo, se ejecutó sin funciones de depuración.

- Componente Visual: Según avanza el número de mensajes y la complejidad de las interacciones, el componente visual de la salida por consola comienza a quedarse muy escaso. Sería deseable un mecanismo que gráficamente pueda mostrar con exactitud qué está pasando en la conversación. Siempre será un soporte de ayuda observar la traza de las interacciones que proporciona INGENIAS en su modo depuración. Sin embargo este recurso es mejorable, porque las trazas de conversaciones no se relacionan con facilidad con los logs. En este sentido, sería deseable algo más gráfico y explícito.

- Aspecto característico: El aspecto característico consiste en una interacción sencilla donde aparecen más de dos roles, y más de una única rama en la conversación.

- Concreción: Prácticamente todo el interés del ejemplo radica en la interacción, y el resto de elementos son secundarios. Sin embargo estos elementos enlazan con la interacción. En este sentido, se ha hecho todo lo posible por reducir el exceso de información. Además la interacción se produce de manera automática, por lo que el sistema comienza y termina con la conversación.

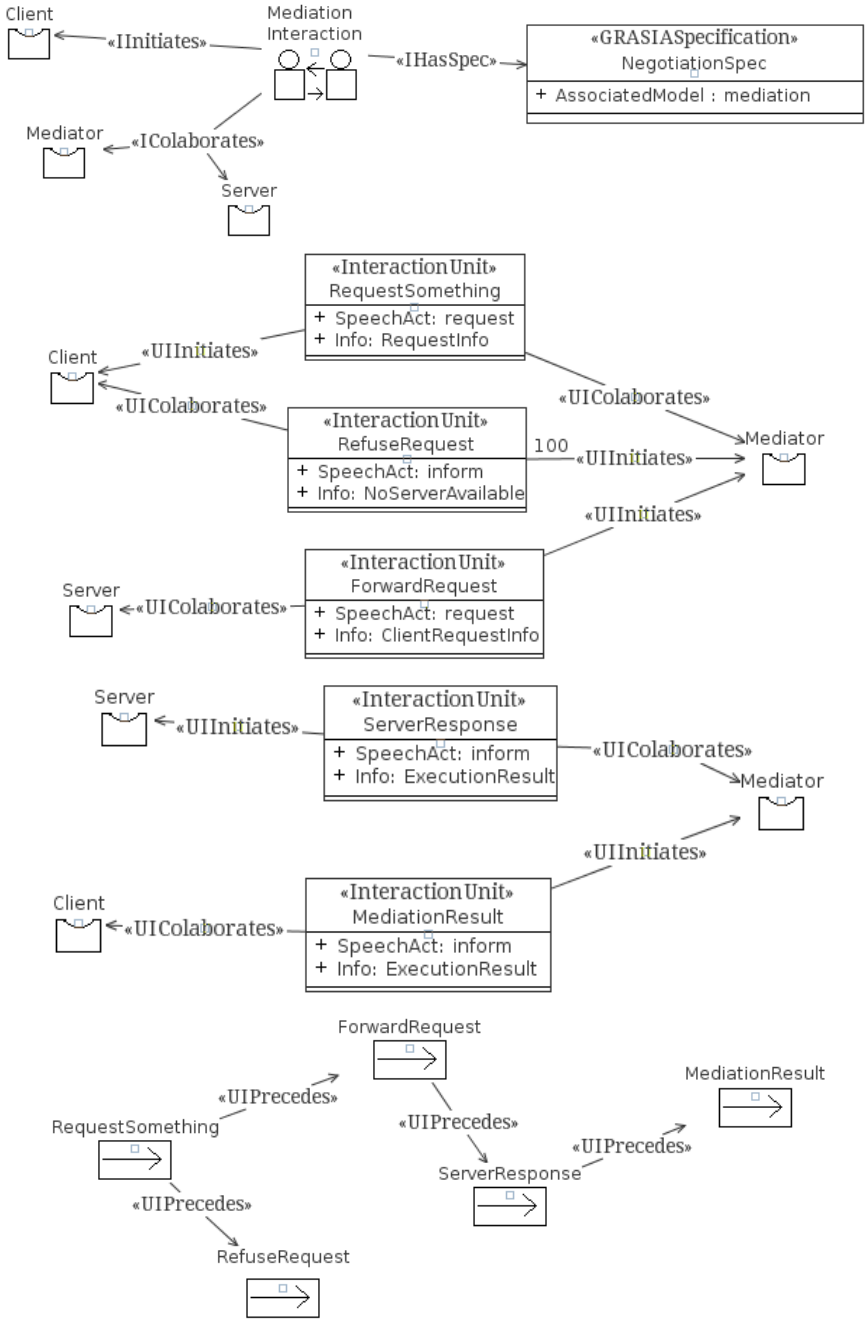

Figura 4.16: *Diagrama de interacción del ejemplo Mediación simple de un agente.*

```
[java] Agent C: Sending request to mediator.
[java] Agent M: Evaluating request.
[java] Agent S: Processing Request.
[java] Agent C: Server response processed.
```

Figura 4.17: *Salida del ejemplo Mediación simple de un agente.*

Ejecutable	**Componente Visual**	**Aspecto característico**	**Concreción**	**Cohesión**	**Contexto**
Sí	Consola*	Interacción	Sí	Sí	Mediación entre agentes

Cuadro 4.4: *Tabla resumen de la evaluación del ejemplo Mediación simple de un agente.*

- Cohesión: La cohesión del ejemplo es adecuada porque cada elemento aparece donde tenga sentido. A su vez, las relaciones de los elementos se producen también en lugares razonables. Sin embargo, los diagramas parecen alborotados.
- Contexto: El contexto es modelar el escenario de una interacción concreta. En este caso, una en la cual un agente ejerce de mediador en una relación productor-consumidor, donde se producen y consumen distintos *FrameFact* y se ejecutan diferentes tareas.

La tabla 4.4 muestra un resumen de la evaluación (los asteriscos en la tabla significa que la valoración requiere argumentación).

Capítulo 5

Conclusiones

Durante este trabajo de investigación se han estudiado las características principales deseables en un ejemplo AOSE para que sea considerado efectivo. En concreto, se han estudiado los ejemplos de metodologías AOSE. Estas características se han extraído de diversos trabajos que revisan la experiencia y dificultades de la educación en la ingeniería del software, además de trabajos relativos a las teorías de aprendizaje, que estudian formas de aprender de manera más efectiva. También se ha considerado incluir características que se han elegido por propia iniciativa, como la característica de contener aspectos representativos de los agentes software. Incluir este tipo de característica, que es exclusiva del dominio, parecía sensato y razonable.

Tras revisar la literatura, se tiene la impresión de que no preocupa la forma en que se enseña AOSE. Todo parece resolverse con un artículo o con un tutorial. Este contrasentido motiva a buscar trabajos que estudien las dificultades al aprender ingeniería del software en paradigmas diferentes. El problema de utilizar este método es que los beneficios van a ser limitados y, a partir de cierto punto, si se pretende mejorar la educación en AOSE, se debería lanzar una línea de investigación con este tema por objeto. El motivo es que hay conceptos propios de AOSE que la ingeniería del software clásica no considera, y una aplicación directa de la educación en la ingeniería del software no es suficiente.

En cuanto a los ejemplos, pese a que se considera que las características identificadas deben ser consideradas, estas características son difíciles de cuantificar. Por ejemplo, valorar la cohesión y la concreción resulta difícil por la limitada cantidad de elementos en los ejemplos desarrollados. Además, la forma de cuantificar estas características debe ser suficientemente flexible como para que se puedan considerar diferentes diseños, dado que en ingeniería suele haber más de una solución válida para un problema. Es decir, la concreción o la cohesión no deberían ser un obstáculo para incluir información adicional, pero sin embargo deben estar consideradas para el ejemplo sea más efectivo (sus elementos se puedan relacionar adecuadamente y se evite el exceso de información).

A su vez, existen características, como el componente visual, que por sí solo el ejemplo puede no ser capaz de cumplir. Por ejemplo, si se considera un ejemplo de un proyecto (un objeto de aprendizaje de mayor escala), es posible que todo el funcionamiento del sistema multiagente sea transparente. Esa propiedad no significa que el ejemplo no pueda ser

interesante, significa que necesita ser modificado. Pero sin una herramienta específica para mostrar el funcionamiento de los agentes dentro del sistema, los componentes visuales que se pueden adaptar a los ejemplos resultan demasiado rudimentarios. En cuanto a los ejemplos en INGENIAS, pese a disponer de la funcionalidad del modo depuración, esta funcionalidad puede no parecer suficientemente explícita. Sería deseable que los logs y las trazas de las interacciones se relacionaran de forma más evidente. Una alternativa sería usar el sniffer de JADE, que permite observar estas comunicaciones en forma de diagrama de secuencia.

5.1. Trabajo futuro.

Existen diferentes líneas de trabajo futuro:

- Producir medidas de las características identificadas. Si se consiguen hacer cuantificables y representables dentro de los ejemplos, se podrán contrastar ejemplos con diferentes niveles de estas, mejorando las condiciones del objeto de aprendizaje. Esto incluiría también listar qué se requiere para cada una de las características.
- Estudiar técnicas de representación para conseguir un recurso visual adecuado para los agentes software, y definir la forma de adaptar los ejemplos existentes para usar esta representación. La aspiración ideal sería utilizar un enfoque como el de la enseñanza de la programación, donde se considera construir aplicaciones que sirven para ver cómo funcionan otras. Estas técnicas podrían estudiarse para su incorporación a INGENIAS o como una herramienta aislada.
- Utilizar las características en la evaluación de ejemplos de proyectos, donde se puedan extraer nuevas conclusiones.
- Investigar las distintas estructuras de ejemplos educativos, buscando definir una estructura adecuada para los ejemplos AOSE.
- Estudiar la inclusión de otras características candidatas, como pueden ser las relativas a definir la complejidad o con la forma en que se estructuran u ordenan los diagramas.
- La efectividad del ejemplo tendría que ser valorada también de otra forma, no sólo por la presencia de características o su evaluación informal, como se ha hecho aquí. Para una evaluación más científica de este aspecto, una forma consistiría en elaborar cuestionarios preguntando a los usuarios acerca de la valoración sobre las características estudiadas (concreción o contexto, por ejemplo) y su presencia en el ejemplo. Por el coste de esta aproximación, se decidió postergarlo para un futuro trabajo de tesis.

Bibliografía

Abran, A., Moore, J. W., Bourque, P., Dupuis, R., and Tripp, L. (2004). Guide to the software engineering body of knowledge, 2004 version. *IEEE Computer Society*, 1.

Ali, M. R. (2006). Imparting effective software engineering education. *ACM SIGSOFT Software Engineering Notes*, 31(4):1–3.

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of educational research*, 70(2):181–214.

Bennedsen, J. and Caspersen, M. E. (2008). Exposing the programming process. In *Reflections on the Teaching of Programming*, pages 6–16. Springer.

Bolognesi, A., Ciancarini, P., and Moretti, R. (2006). On the education of future software engineers. In *Software Engineering Education in the Modern Age*, pages 186–205. Springer.

Bordini, R. H. and Hübner, J. F. (2006). Bdi agent programming in agentspeak using jason. In *Computational logic in multi-agent systems*, pages 143–164. Springer.

Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience.

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

Burrafato, P. and Cossentino, M. (2002). Designing a multi-agent solution for a bookstore with the passi methodology. In *Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002)*, pages 27–28.

Catrambone, R. and Holyoak, K. J. (1990). Learning subgoals and methods for solving probability problems. *Memory & Cognition*, 18(6):593–603.

Claypool, K. and Claypool, M. (2005). Teaching software engineering through game design. *ACM SIGCSE Bulletin*, 37(3):123–127.

Cossentino, M. (2005). From requirements to code with the passi methodology. *Agent-oriented methodologies*, 4:79–106.

Dastani, M. (2008). 2apl: a practical agent programming language. *Autonomous agents and multi-agent systems*, 16(3):214–248.

EASS (2000). Courses in european agent systems summer school. "http://www.dfki.de/easss/".

EASS (2008). Courses in european agent systems summer school. "http://centria.di.fct.unl.pt/events/easss08/course_details.html#Intro".

EASS (2009). Courses in european agent systems summer school. "http://agents009.di.unito.it/easss_courses.html".

EASS (2010). Courses in european agent systems summer school. "easss2010.emse.fr/programme.php".

EASS (2011). Courses in european agent systems summer school. "http://eia.udg.edu/easss2011/index.php?content=programme".

EASS (2012). Courses in european agent systems summer school. "http://easss2012.webs.upv.es/programme.html".

Giorgini, P., Kolp, M., Mylopoulos, J., and Castro, J. (2005). Tropos: A requirements-driven methodology for agent-oriented software. *Agent-Oriented Methodologies*, page 20.

Gnatz, M., Kof, L., Prilmeier, F., and Seifert, T. (2003). A practical approach of teaching software engineering. In *Software Engineering Education and Training, 2003.(CSEE&T 2003). Proceedings. 16th Conference on*, pages 120–128. IEEE.

Gómez-Martín, M. A. and Gómez-Martín, P. P. (2009). Fighting against the 'but it works!'syndrome. In *XI International Symposium on Computers in Education (SIIE 2009)*.

Gómez-Sanz, J. J. and Fuentes-Fernández, R. (2013). Understanding agent-oriented engineering methodologies. *The Knowledge Engineering Review*.

Halling, M., Zuser, W., Kohle, M., and Biffl, S. (2002). Teaching the unified process to undergraduate students. In *Software Engineering Education and Training, 2002.(CSEE&T 2002). Proceedings. 15th Conference on*, pages 148–159. IEEE.

Hazzan, O. and Dubinsky, Y. (2003). Teaching a software development methodology: the case of extreme programming. In *Software Engineering Education and Training, 2003.(CSEE&T 2003). Proceedings. 16th Conference on*, pages 176–184. IEEE.

IEEE (1990). Ieee standard glossary of software engineering terminology.

Jennings, N. R. (1999). Agent-oriented software engineering. In *Multi-Agent System Engineering*, pages 1–7. Springer.

Kölling, M. and Barnes, D. (2008). Apprentice-based learning via integrated lectures and assignments. *Reflections on the Teaching of Programming*, pages 17–29.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, volume 37, pages 14–18. ACM.

Liu, S., Takahashi, K., Hayashi, T., and Nakayama, T. (2009). Teaching formal methods in the context of software engineering. *ACM SIGCSE Bulletin*, 41(2):17–23.

Marton, F. and Pang, M. F. (2006). On some necessary conditions of learning. *The Journal of the Learning sciences*, 15(2):193–220.

Mills, J. E., Treagust, D. F., et al. (2003). Engineering education—is problem-based or project-based learning the answer? *Australasian Journal of Engineering Education*, 3:2–16.

Navarro, E. O. and Van Der Hoek, A. (2009). On the role of learning theories in furthering software engineering education. *Software Engineering: Effective Teaching and Learning Approaches and Practices*, pages 38–59.

Nunes, I., Cirilo, E., de Lucena, C. J., Sudeikat, J., Hahn, C., and Gomez-Sanz, J. J. (2011). A survey on the implementation of agent oriented specifications. In *Agent-Oriented Software Engineering X*, pages 169–179. Springer.

Offutt, J. (2013). Putting the engineering into software engineering education. *Software, IEEE*, 30(1):96–96.

Padgham, L. and Winikoff, M. (2005). Prometheus: A practical agent-oriented methodology. *Agent-Oriented Methodologies*, pages 107–135.

Parnas, D. L. (1990). Education for computing professionals. *Computer*, 23(1):17–22.

Parnas, D. L. (1999). Software engineering programs are not computer science programs. *Software, IEEE*, 16(6):19–30.

Pavón, J. and Gómez-Sanz, J. (2003). Agent oriented software engineering with ingenias. In *Multi-Agent Systems and Applications III*, pages 394–403. Springer.

Pavon, J., Gomez-Sanz, J. J., and Fuentes, R. (2005). The ingenias methodology and tools. *Agent-oriented methodologies*, 9:236–276.

Petkovic, D., Thompson, G., and Todtenhoefer, R. (2006). Teaching practical software engineering and global software engineering: evaluation and comparison. In *ACM SIGCSE Bulletin*, volume 38, pages 294–298. ACM.

Pollock, E., Chandler, P., and Sweller, J. (2002). Assimilating complex information. *Learning and instruction*, 12(1):61–86.

Pucher, R. and Lehner, M. (2011). Project based learning in computer science–a review of more than 500 projects. *Procedia-Social and Behavioral Sciences*, 29:1561–1566.

Runeson, P. (2001). Experiences from teaching psp for freshmen. In *Software Engineering Education and Training, 2001. Proceedings. 14th Conference on*, pages 98–107. IEEE.

Shackelford, R., McGettrick, A., Sloan, R., Topi, H., Davies, G., Kamali, R., Cross, J., Impagliazzo, J., LeBlanc, R., and Lunt, B. (2006). Computing curricula 2005: The overview report. *ACM SIGCSE Bulletin*, 38(1):456–457.

Shaw, M. (2000). Software engineering education: a roadmap. In *Proceedings of the conference on The future of Software Engineering*, pages 371–380. ACM.

Sterling, L. S. and Taveter, K. (2009). *The art of agent-oriented modeling*. Mit Pr.

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and instruction*, 4(4):295–312.

Ward, M. and Sweller, J. (1990). Structuring effective worked examples. *Cognition and instruction*, 7(1):1–39.

Winikoff, M. and Padgham, L. (2004). The prometheus methodology. *Methodologies and Software Engineering For Agent Systems*, pages 217–234.

Wooldridge, M. and Ciancarini, P. (2001). Agent-oriented software engineering: the state of the art. *Lecture notes in computer science*, pages 1–28.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312.

Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2005). Multi-agent systems as computational organizations: the gaia methodology. *Agent-oriented methodologies*, pages 163–171.