Tomás Ruiz-López

Un enfoque dirigido por modelos para el desarrollo de servicios para sistemas ubicuos basado en propiedades de calidad

Bajo la dirección académica de:
José Luis Garrido Bullejos
María José Rodríguez Fórtiz

Universidad de Granada
Departamento de Lenguajes y Sistemas Informáticos

Editor: Editorial de la Universidad de Granada
Autor: Tomás Ruiz López
D.L.: GR 1947-2014
ISBN: 978-84-9083-112-0

Tomás Ruiz-López: *Un enfoque dirigido por modelos para el desarrollo de servicios para sistemas ubicuos basado en propiedades de calidad*, Tesis Doctoral, © Mayo 2014.
Universidad de Granada, Departamento de Lenguajes y Sistemas Informáticos.

DIRECTORES:
José Luis Garrido Bullejos, María José Rodríguez Fórtiz

WEBSITE:
http://www.ugr.es/~tomruiz

E-MAIL:
tomruiz@ugr.es

La memoria titulada **«Un enfoque dirigido por modelos para el desarrollo de servicios para sistemas ubicuos basado en propiedades de calidad»**, que presenta D. Tomás Ruiz López para optar al título de Doctor en Informática, ha sido realizada en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad de Granada, dentro del Programa Oficial de Doctorado en Tecnologías de la Información y las Comunicaciones, bajo la dirección de los doctores D. José Luis Garrido Bullejos y Dña. María José Rodríguez Fórtiz.

Doctorando

Tomás Ruiz López

Directores de la Tesis

José Luis Garrido Bullejos

María José Rodríguez Fórtiz

El doctorando Tomás Ruiz López y los directores de tesis José Luis Garrido Bullejos y María José Rodríguez Fórtiz garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis y hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se hace referencia a sus resultados y/o publicaciones.

Granada, 22 de Mayo de 2013

Doctorando

Tomás Ruiz López

Directores de la Tesis

José Luis Garrido Bullejos

María José Rodríguez Fórtiz

Esta tesis doctoral es el resultado de un trabajo de investigación financiado por una beca de Formación del Profesorado Universitario (FPU), referencia AP2010-0867, otorgada por el Ministerio de Educación, Ciencia y Deporte en Diciembre de 2011.

El doctorando realizó una estancia de investigación en la Universidad de Texas en Dallas, bajo la tutorización del Dr. Lawrence Chung, que fue financiada por el Vicerrectorado de Relaciones Internacionales de la Universidad de Granada.

El doctorando es miembro del grupo de investigación en Modelado y Desarrollo de Sistemas Software Avanzados (MYDASS), donde ha participado en los siguientes proyectos:

- Proyecto Nacional TIN2012-38600 financiado por el Ministerio de Economía y Competitividad con fondos FEDER.
- Proyecto TIN-6600, financiado por la Oficina de Innovación de la Junta de Andalucía.
- Proyecto Sígueme financiado por la Fundación Orange.
- Proyecto de Innovación Docente de la Universidad de Granada titulado «Extensión de la Plataforma de Debate Virtual para dar soporte al Análisis y Mejora de la Usabilidad y Evaluación del Alumnado».
- Proyecto de Innovación Docente de la Universidad de Granada titulado «Trabajo en Grupo Multidisciplinar para Alumnado y Profesorado de Máster».

AGRADECIMIENTOS

Cuando cumplí 13 años supe que quería ser Ingeniero en Informática. Quizá ahora no sea raro, pero en aquel momento supongo que no sería lo más normal. Al fin y al cabo, uno suele tener vocación de médico o de maestro, pero no es muy habitual tener vocación de informático. Por aquel entonces ni siquiera tenía ordenador en casa (que a día de hoy, unos 13 años después, suena bastante más raro aún), pero en cuanto empecé a interesarme por la informática, mis padres me compraron mi primer Pentium I. Es algo que siempre tendré que agradecerles. Pero si quiero darles las gracias a ellos en primer lugar, no es por comprarme un ordenador (o pagarme una carrera y un año viviendo en los Estados Unidos). El motivo principal por el que quiero agradecer a mis padres, Tomás y Elisa, es por haberme enseñado y transmitido unos valores que son los que hacen que hoy esté aquí: el sentido de la responsabilidad, el esfuerzo por superarme, el valor del trabajo profesional y bien hecho, el no rendirme y dar lo mejor de mí, el ser autónomo y valerme por mí mismo, el ser capaz de tomar decisiones importantes, y un montón de valores más que me llevan a intentar ser mejor persona cada día. Si sólo pudiera escribir un agradecimiento, iría para vosotros.

Como decía, casi toda mi vida he querido ser Ingeniero en Informática, pero nunca me había planteado realizar un doctorado, hasta que pasé el que ha sido el mejor año de mi formación en la Universidad de California en Irvine. Quiero dar las gracias al Dr. André van der Hoek por haberme dado la oportunidad de trabajar con su equipo y haberme dado el empujoncito para escribir esta tesis doctoral.

Hablando con gente que ha pasado por el proceso de escribir una tesis doctoral, muchos hacen la comparación con «un parto muy largo», así que tengo que agradecer a Lidia por haber sido quien me ha acompañado en los momentos más difíciles de este «embarazo». Gracias por estar ahí escuchándome siempre con tu mejor sonrisa y dándome ánimos para continuar cuando las cosas se ponen cuesta arriba.

Quiero agradecer también a la gente que me ha acompañado en este proceso de formación. A José Luis y María José, mis directores, y a Manolo y Kawtar, que no figuran como tales pero que podrían serlo perfectamente, por haberme dado siempre los

mejores consejos y comentarios, que sin duda han contribuido a mejorar cualquier idea que haya propuesto en este documento. A Lawrence Chung y a su equipo de la Universidad de Texas en Dallas, en especial a Rutvij Mehta, Sam Supakkul y Tom Hill, por haberme acogido durante los tres meses de mi estancia de investigación, haberme hecho sentir uno más del grupo y haber contribuido a mejorar mi investigación. A toda la gente del grupo de investigación MYDASS con la que he tenido la oportunidad de trabajar durante estos años en varios proyectos, que también han contribuido tanto a mi desarrollo profesional como personal.

Por último, quiero agradecer a todas aquellas personas que son importantes para mí. A mi «hermanito» Juan, por implicarme en sus proyectos y confiar en mí para llevarlos a cabo, y por darme muchos momentos de distracción y diversión. A mis amigos Carlos y Álvaro, por compartir juntos nuestro gran proyecto, Everyware Technologies, y por nuestras largas charlas y momentos de desconexión. A mis amigos Carmen, Elisabeth, Roberto, Araceli, Sita y Conce, por haber estado ahí durante todos estos años, en los buenos momentos y en los no tan buenos, y por todos esos ratos de diversión que hemos tenido (y que tendremos).

A todos vosotros, y a todos los que me haya podido dejar fuera, muchas gracias.

RESUMEN

La Computación Ubicua, propuesta por Mark Weiser en 1991, es un nuevo paradigma de interacción persona-ordenador que aboga por la incorporación de múltiples unidades de computación distribuidas en el entorno del usuario, inmersas en objetos de la vida diaria, de manera que su uso sea transparente y no intrusivo, y se comuniquen entre sí para facilitar al usuario el desarrollo de sus tareas. Entre las características más destacables de este nuevo paradigma se encuentran la consciencia del contexto (habilidad de los sistemas para detectar y reaccionar ante cambios en su entorno de ejecución) o la adaptatividad (habilidad de los sistemas para cambiar su estructura o comportamiento para cubrir mejor con una necesidad).

Desde entonces, se han propuesto numerosos sistemas que tratan de implementar las ideas de la Computación Ubicua y que pueden encontrarse en la bibliografía. En los últimos años, gracias a avances tales como el desarrollo de las tecnologías móviles o el incremento de velocidad en las conexiones de red inalámbricas, se ha posibilitado aún más el desarrollo de tales sistemas. Sin embargo, el desarrollo de estos sistemas se ha realizado de forma ad-hoc, sin contar con un soporte metodológico apropiado que guíe a los ingenieros de software en el proceso de análisis, diseño e implementación de los mismos, y sin tener en cuenta las características especiales del nuevo paradigma.

En primer lugar, el análisis de los requisitos de los Sistemas Ubicuos ha estado frecuentemente centrado en los Requisitos Funcionales, relegando los Requisitos No Funcionales (*Non Functional Requirements*, NFR), tales como escalabilidad, robustez o seguridad, a un segundo plano. No obstante, la importancia de este tipo de requisitos es crítica para el diseño de software de calidad, especialmente cuando se toma en consideración la naturaleza dinámica y cambiante, dependiente del contexto, de los Sistemas Ubicuos.

Por otra parte, los métodos de diseño existentes en la bibliografía emplean el paradigma de la Arquitectura Orientada a Servicios. Aunque dicho enfoque es particularmente apropiado para el diseño de Sistemas Ubicuos, las propuestas existentes se centran más en la adquisición y gestión del contexto, principalmente mediante el empleo de ontologías, que en dotar al diseño de las estructuras necesarias para adquirir, manipular y adaptarse en base a los cambios ocurridos en dicho contexto.

Finalmente, algunos trabajos existentes optan por la aplicación de técnicas basadas en Ingeniería Dirigida por Modelos para la implementación del Sistema Ubicuo diseñado. En general, dichas técnicas corresponden a estrategias de generación de código para diferentes plataformas, en lugar de proponer un enfoque más global que cubra todo el ciclo de vida del software y permita realizar transformaciones de modelos desde la especificación de requisitos hasta dicha generación de código.

Para cubrir estas carencias detectadas, esta tesis doctoral propone un enfoque dirigido por modelos para el desarrollo de servicios para sistemas ubicuos basado en propiedades de calidad, el cual se ha denominado MDUBI. Este enfoque comienza en la etapa de Ingeniería de Requisitos, para la cual se ha propuesto un método, llamado REUBI (*Requirements Engineering method for Ubiquitous Systems*), el cual está orientado a realizar un tratamiento sistemático de los NFR teniendo en cuenta las características especiales de los Sistemas Ubicuos e incorporando mecanismos para representar y gestionar la influencia del contexto en los requisitos. Posteriormente, se ha propuesto un método de diseño de servicios, llamado SCUBI (*Service Component-based Design Method for Ubiquitous Systems*). Dicho método promueve la creación de servicios para sistemas ubicuos, los cuales deben ser construidos en términos de componentes. Incorpora estructuras que permiten la adquisición, gestión y razonamiento del contexto, así como mecanismos para adaptar el servicio ante cambios en el mismo.

El enfoque MDUBI contiene un conjunto de transformaciones que permiten, a partir de un modelo de requisitos conforme al método REUBI, obtener de manera (semi-)automática modelos de diseño conforme al método SCUBI, y posteriormente transformarlos para su implementación en plataformas concretas mediante técnicas de generación de código.

La propuesta ha sido validada mediante su aplicación a un sistema real consistente en un servicio de localización híbrido capaz de dar soporte y adaptarse en tiempo real al posicionamiento en interiores y exteriores haciendo uso de diferentes métodos y tecnologías de localización, diseñados e implementados como componentes.

ABSTRACT

Ubiquitous Computing, as proposed by Mark Weiser in 1991, is a new human-computer interaction paradigm that advocates for the incorporation of multiple computing units distributed around the user's environment, immerse in daily life objects, in such a way that their use is transparent and unobtrusive, and that communicate to each other in order to ease the performance of user's tasks. Among the most remarkable features of this paradigm are context-awareness (ability of the systems to sense and react to changes in the execution environment) or adaptivity (ability of the systems to change their structure or behavior to better fit some needs).

Since then, numerous systems aiming to implement the ideas of Ubiquitous Computing have been proposed and can be found in the bibliography. In the past few years, thanks to advances such as the development of mobile technologies or the speed increment in wireless network communications, the development of such systems has been more feasible. However, the development of these systems has been done in an ad-hoc manner, without the appropriate methodological support to guide software engineers in the process of analysis, design and implementation, and without taking into consideration the special features of the new paradigm.

First, the analysis of the requirements of Ubiquitous Systems has been frequently focused on the Functional Requirements, relegating Non-Functional Requirements (NFR), such as scalability, robustness or security, to the background. Nevertheless, the importance of these type of requirements is critical to the design of high quality software, specially when considering the changing and dynamic, context-dependent, nature of Ubiquitous Systems.

Besides, existing design methods in the bibliography make use of Service Oriented Architecture paradigm. Although this approach is particularly appropriate to the design of Ubiquitous Systems, the existing proposals are focused in the acquisition and management of context, mainly with the use of ontologies, rather than focusing on providing the design with structures to acquire, manipulate and adapt themselves to contextual changes.

Finally, some existing works apply Model-Driven Engineering techniques for the implementation of the designed Ubiquitous System. Generally, such techniques correspond to code genera-

tion strategies for different platforms, instead of proposing a more general approach covering the whole software lifecycle and enabling the model transformation from the requirements specification to code generation.

In order to fulfill the identified shortages, this doctoral thesis proposes a model-driven approach for the development of services for ubiquitous systems based on quality properties, which has been named MDUBI. This approach starts in the Requirements Engineering stage, where a Requirements Engineering method for Ubiquitous Systems (REUBI) has been proposed. REUBI aims to perform a systematic treatment of NFR taking into consideration the special features of Ubiquitous Systems and incorporating mechanisms to represent and manage the influence of context on the requirements. After that, a Service Component-based Design Method for Ubiquitous Systems (SCUBI) has been proposed. SCUBI promotes the creation of services for Ubiquitous Systems, which have to be built in terms of components. It incorporates structures for the acquisition, management and reasoning about context, as well as mechanisms to adapt the service upon contextual changes.

MDUBI contains a set of transformations that enable, from a requirements model conforming to REUBI, obtaining in a (semi-) automatic manner design models conforming to SCUBI, and later transforming them for their implementation in concrete platforms by means of code generation strategies.

The proposal has been validated through its application to a real system consisting of a hybrid positioning service which is able to support indoor and outdoor positioning, adapting itself in real time, making use of different localization methods and technologies, which have been designed and implemented as components.

ÍNDICE GENERAL

Parte I

Introducción

1 INTRODUCCIÓN

Índice

1.1 Introducción

Evolución de la Informática

En los últimos años, la informática ha evolucionado desde sus aplicaciones iniciales orientadas a realizar tareas repetitivas y de almacenamiento de datos sobre máquinas enormes, empleadas por varias personas, hasta la era de los ordenadores personales (PCs) que existen actualmente en la mayoría de los hogares. A partir de este punto, ha comenzado una tercera era, influenciada y conocida como la era de la **Computación Ubicua** (figura 1.1). La **ubicuidad** es la propiedad por la cual una entidad existe o se encuentra en todos los sitios al mismo tiempo.

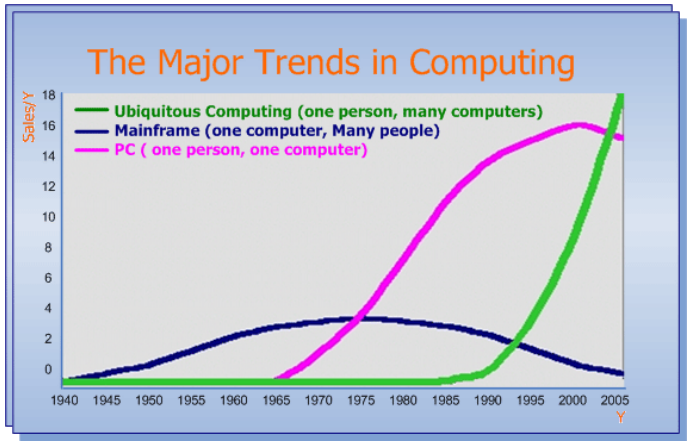

Figura 1.1: Evolución de las diferentes etapas de la informática.

Paradigmas de Interacción

La **Computación Ubicua** es un **Paradigma de Interacción Persona-Ordenador** que está ganando cada vez mayor aceptación gracias a los recientes avances dentro de este área en los últimos años (desarrollo de tecnologías móviles, avance de Internet, aparición de numerosos sensores y actuadores, implantación de entornos inteligentes...). Dentro de los Paradigmas de Interacción se distinguen los siguientes:

- **Computación Personal o de Sobremesa:** es el paradigma de interacción predominante hasta hace unos años, consistente en la interacción entre un usuario con su computador (PC, ordenador portátil, tablet PC...). Habitualmente, el usuario interactúa con el ordenador aislado del entorno, aunque el desarrollo de las tecnologías inalámbricas le permiten un mayor contacto con el entorno.

- **Realidad Virtual:** consiste en la simulación por ordenador en la que se crea por medios visuales un mundo que parece realista y dinámico que responde a las órdenes del usuario. Las claves de este paradigma son la interactividad en tiempo real y el sentimiento de inmersión al participar en lo que se desarrolla en la escena. Típicamente involucra el uso de hardware específico, como cascos de realidad virtual, gafas polarizadas, dispositivos hápticos... Tiene el problema de suponer un alto coste tanto en dispositivos como en la realización de la simulación, y de producir cansancio en el usuario.

- **Realidad Aumentada, Dual o Mezclada:** trata de reducir las interacciones con el ordenador utilizando la información del entorno como una entrada implícita. Existen dos corrientes en este paradigma: el resultado de aplicar la realidad virtual en el mundo real y el uso de dispositivos que aumentan la realidad e interaccionan directamente con ella.

- **Computación Ubicua:** propuesta por Mark Weiser [Wei93], trata de extender la capacidad de computación al entorno del usuario, permitiendo que la capacidad de información esté presente en todas partes en forma de pequeños dispositivos que permiten interacciones de poca dificultad e interconectados entre sí. Su diseño y localización deben ser ideados especialmente para la tarea objeto de interacción; por tanto, la computación deja de estar localizada en un único punto (el escritorio) para diluirse en el entorno.

Computación Ubicua

La Computación Ubicua (*Ubiquitous Computing*, *Pervasive Com-*

puting, *everyware*) fue descrita por primera vez por Mark Weiser en 1991 [Wei91]. La esencia de su visión era la creación de entornos con gran cantidad de elementos de cómputo y con capacidad de comunicación, integrados de forma inapreciable para las personas. Uno de sus objetivos más importantes es la integración de dispositivos computacionales en el mayor grado posible, de manera que formen parte de la vida cotidiana y que permitan a los usuarios centrarse en las tareas que deben realizar en lugar de hacerlo en las herramientas que deben emplear. Enviar la computación a un «segundo plano» implica (i) la integración de la tecnología en los objetos de la vida diaria (figura 1.2), y (ii) la no interferencia de dichos elementos en las actividades para las que son usados, proporcionando un uso más cómodo, sencillo y útil de los mismos.

Figura 1.2: Integración de la tecnología en una taza.

Características de la Computación Ubicua

Así, la Computación Ubicua incorpora cuatro nuevos conceptos:

- **Uso eficaz de espacios inteligentes** (*Effective use of perceptive spaces*): se basa en la detección del estado de un individuo y de sus necesidades, deducidas de dicho estado, ya sea en la oficina, sala de reuniones, clase, domicilio, coche... El espacio inteligente surge cuando varios dispositivos inteligentes coinciden en el mismo espacio físico e interactúan colaborativamente para dar soporte a los individuos que se encuentren en él. La domótica es la aplicación más popular de este concepto.

- **Invisibilidad** (*Invisibility*): aunque actualmente la tecnología de la que disponemos está lejos de la completa desaparición de la tecnología de la consciencia del usuario, es necesario tener presente la idea de la mínima distracción del usuario cuando se diseñan estos sistemas. La invisibilidad require cambios drásticos en el tipo de interfaces que se emplean para comunicarse con los ordenadores, siendo necesarias tecnologías de reconocimiento de voz, gestos, comprensión del lenguaje natural y del texto manuscrito, síntesis del lenguaje hablado y escrito, y representaciones gráficas.

- **Escalabilidad localzada** (*Localized scalability*): el concepto de localidad de servicios en Computación Ubicua es fundamental frente a la universalidad de servicios en Internet. Los usuarios disponen de capacidades asociadas al contexto en el que se encuentran, careciendo de sentido, por ejemplo, que las aplicaciones domóticas situadas en el domicilio particular tengan que estar escrutando las necesidades del usuario que se encuentra trabajando en ese momento en la oficina.

- **Ocultación del condicionamiento desigual** (*Hiding uneven conditioning*): dependiendo de la infraestructura y del desarrollo tecnológico disponible, la distribución de los servicios puede ser muy poco uniforme. En esta situación el principio de invisibilidad puede no cumplirse ya que el usuario detectaría desagradables transiciones. Este requisito es hoy día el más alejado respecto a la situación ideal; los sistemas que incorporan computación ubicua pueden estar aislados, sin continuidad entre unos y otros.

Investigación en Computación Ubicua

En el campo de la investigación en Sistemas Ubicuos pueden encontrarse numerosos trabajos y desarrollos de los conceptos propuestos por el paradigma de la Computación Ubicua, de los cuales se mencionan algunos relevantes. El *Proximity Toolkit* [MDMBG11] es un kit de desarrollo para facilitar la creación de sistemas en los que la interacción se basa en la proximidad del usuario a ciertos dispositivos o a otros usuarios. El *framework* proporciona información próxemica (orientación, distancia, movimiento, identidad y localización) en forma de una API para que el desarrollador pueda crear las interacciones oportunas en su sistema. Además, se muestran varias aplicaciones del kit, tales como un sistema de anuncios que requiere la atención del usuario, o una experiencia musical basada en la proximidad a un dispositivo.

Avrahami *et al.* [AYL11] exploran la posibilidad de incorporar Sistemas Ubicuos dentro de los coches. En su estudio, muestran estadísticas del número de accidentes de tráfico que causan lesiones a los ocupantes del vehículo como consecuencia de llevar objetos sueltos dentro del vehículo. Para tratar de reducir este hecho, proponen un sistema basado en cámaras y sensores que tratan de determinar qué objetos están sueltos y avisar al conductor de un peligro potencial.

Kortuem *et al.* [KKFS10] estudian la posibilidad de transformar objetos de la vida diaria en objetos inteligentes (*smart objects*) para ayudar a las personas a realizar diversas tareas. Presentan una taxonomía de clasificación de estos objetos en las que se analizan tres dimensiones de diseño: conciencia (capacidad del dispositivo para reconocer eventos del mundo real), representación (abstracciones de programación del dispositivo) e interacción (capacidad del dispositivo para conversar con el usuario en términos de entradas, salidas, control y feedback). Se muestra un caso de estudio de un sistema de alerta a trabajadores que arreglan una carretera para informarles, mediante un dispositivo que llevan puesto, de los posibles riesgos para su salud del uso prolongado de cierta maquinaria.

Ghose *et al.* [GSB+13] presentan un sistema de monitorización de la salud en personas mayores y con problemas crónicos de salud, llamado UbiHeld. Para ello, dada la extensión de uso de los teléfonos móviles inteligentes, hacen uso de esta plataforma para tener acceso al sistema de monitorización de la salud en todas partes. Para analizar el movimiento de la persona, combinan el teléfono móvil junto con Kinect en lugar de cámaras para reducir la intrusividad en la privacidad del usuario. Por último, para analizar el estado mental de la persona, tienen acceso a los contenidos publicados por el paciente en redes sociales.

1.2 Planteamiento del problema

Carencias en la Investigación sobre Sistemas Ubicuos

Del análisis de los Sistemas Ubicuos disponibles en la bibliografía sobre investigación en Computación Ubicua se desprende que el foco de atención es el desarrollo de nueva funcionalidad y la inclusión de las tecnologías más novedosas en los sistemas que se deben desarrollar, mientras que las características no funcionales de los sistemas reciben poca atención por parte de los diseñadores [MCY99] [CNYM00]. Algunos NFR, como la eficiencia del sistema o la usabilidad, habitualmente se están teniendo en cuenta en el desarrollo de Sistemas Ubicuos, mientras que otros, como la reutilización o la escalabilidad, son relegados a un

segundo plano, o no son siquiera tratados. Además, la mayoría de estos sistemas se construyen de manera *ad-hoc*, sin un soporte metodológico que guíe al diseñador y le permita tomar decisiones de manera sistemática, tanto en el análisis de los requisitos del sistema, como en su posterior diseño.

Requisitos No Funcionales

Aunque los Requisitos No Funcionales (*Non-functional Requirements*, NFR) determinan en gran medida la calidad de los sistemas [MCY99], y esto es particularmente cierto en el ámbito de la Computación Ubicua dadas sus características especiales, pocos trabajos existen en la bibliografía que permitan tratar de manera sistemática con los NFR en Sistemas Ubicuos. Entre las características especiales de la Computación Ubicua relacionadas con los NFR destacan:

Características especiales de los Sistemas Ubicuos

- **Consciencia del contexto:** los sistemas ubicuos monitorizan ciertas variables de su entorno de ejecución que resultan de interés para su funcionamiento, así como para tratar de adaptarse y cumplir con los requisitos de calidad que se esperan de ellos [Fero6].

- **Dinamicidad del entorno:** relacionado con el punto anterior, los entornos de implantación de los sistemas ubicuos son variables y cambiantes. Dichos cambios no afectan solo a la parte funcional del sistema (por ejemplo, ofrecer diferentes servicios según la ubicación), sino que implican también cambios para mantener la calidad del sistema [AFN].

- **Heterogeneidad de soluciones tecnológicas:** la amplia variedad de desarrollos tecnológicos existentes, cada uno con diferentes propiedades de calidad, hace que el diseñador se enfrente a un proceso de selección que, habitualmente, está guiado simplemente por la incorporación de los últimos desarrollos, en lugar de contar con un proceso sistemático de selección basado en la elección de aquella tecnología que presente mejores propiedades de calidad [RLGBC10].

- **Adaptatividad y personalización:** aparte de tener en cuenta las características del entorno de ejecución del sistema ubicuo, este tipo de sistemas deben tomar en consideración las preferencias de los usuarios, tanto a nivel funcional para ofrecer aquellos servicios que les sean de interés, como no funcional, para proporcionarlos con las propiedades de calidad (velocidad, usabilidad...) que los usuarios esperan de ellos [DBS+02].

- **Variabilidad en la priorización:** en los sistemas software tradicionales los requisitos reciben una prioridad de satisfacción que no varía durante la ejecución del sistema. Sin embargo, en este tipo de sistemas dicha prioridad puede cambiar a lo largo de la ejecución del sistema, por lo que el sistema debe ser capaz de determinar los cambios que deben ocurrir en su estructura interna para proporcionar un servicio de calidad de acuerdo a las nuevas prioridades [SLK06].

Reutilización de conocimiento en Ingeniería de Requisitos

Además, en multitud de ocasiones, contar con un método de Ingeniería de Requisitos que garantice un tratamiento sistemático de los NFRs no es suficiente para realizar esta tarea de manera satisfactoria. En efecto, es una tarea difícil y que requiere un conocimiento amplio y profundo sobre el dominio de aplicación del sistema, y más teniendo en cuenta las características especiales de este paradigma mencionadas anteriormente. Adicionalmente, este conocimiento, hasta la fecha, ha permanecido implícito o recogido de manera informal, perdiéndose la oportunidad de reutilizarlo y llevando, en ocasiones, a errores en la comunicación.

Patrones de requisitos

Existen situaciones recurrentes en las que soluciones similares pueden aplicarse para modelar sus requisitos y garantizar el cumplimiento de sus propiedades de calidad. Indudablemente, capturar el conocimiento sobre dichos problemas y las soluciones habituales que se les dan para su posterior reutilización sería muy deseable. Una de las posibles formas de realizar este propósito consiste en la definición de patrones de requisitos. El uso de patrones en la fase de Ingeniería de Requisitos no es nuevo [SHC+10], pero ha recibido poca o ninguna importancia en el ámbito de la Computación Ubicua. De esta forma, capturar la información sobre problemas recurrentes en esta etapa en términos de patrones y su reutilización de manera complementaria al método de Ingeniería de Requisitos podría dar buenos resultados en cuanto al cumplimiento de los NFRs.

Diseño de sistemas ubicuos

Una vez obtenida una especificación y análisis de requisitos en la que se hayan tratado sistemáticamente los NFRs, el siguiente paso en el ciclo de vida del software consistiría en la realización de un diseño que diera soporte a dichos requisitos. Si se analiza la bibliografía sobre los desarrollos realizados en el ámbito de la Computación Ubicua, puede encontrarse que, en gran medida, dichos desarrollos están guiados por la elección de tecnologías concretas (habitualmente soluciones middleware), en lugar de realizar un diseño independiente de la plataforma tecnológica elegida, retrasando su elección hasta que se haya decidido cuál es la más apropiada.

Uso de SOA en sistemas ubicuos

Por las características de la Computación Ubicua, uno de los paradigmas más ampliamente usados a la hora de realizar el diseño de sistemas es el paradigma de la Arquitectura Orientada a Servicios (SOA). Este paradigma permite la creación de servicios con un grado bajo de acoplamiento y una alta reutilización [Erl08], de manera que diferentes servicios pueden instanciarse en distintos entornos y estar disponibles de manera local en aquéllas ubicaciones donde tiene sentido, facilitando el cumplimiento del principio de localidad, y además ser descubiertos de manera dinámica, favoreciendo la movilidad de los usuarios. Por ello, parece apropiado contar con metodologías orientadas al diseño de servicios para sistemas ubicuos.

Adaptación de servicios

Además, es necesario tener en cuenta la necesidad de adaptación de estos servicios conforme a cambios en el contexto. Un proceso de adaptación habitual en computación ubicua [KPP10] consta de tres fases: (i) adquisición del contexto; (ii) razonamiento sobre la adaptación a realizar; y (iii) actuación. Así, el diseño que se realice de los servicios deberá tener en cuenta este proceso para realizar la adaptación de la manera más sencilla posible.

Ingeniería dirigida por modelos

Finalmente, para tratar de realizar un proceso más orientado a la ingeniería que automatice aquellas tareas que puedan realizarse de manera sistemática, algunos autores proponen el empleo de la Ingeniería dirigida por modelos (*Model-driven Engineering*, MDE), y particularmente de la Arquitectura dirigida por modelos (*Model-driven Architecture*, MDA [Gro03b]).

1.3 Motivación

Ingeniería de Requisitos en Sistemas Ubicuos

El *NFR Framework*, propuesto por Chung *et al.* [CNYM00] es un *framework* que permite al diseñador tratar de manera sistemática con los NFR. Presenta el concepto de *softgoal* como un objetivo para el cual no existe un criterio que permita determinar su satisfacción de manera clara, sino que se deben estudiar las contribuciones de las decisiones de diseño tomadas para determinar su grado de satisfacción. De esta forma, se pueden modelar los NFR mediante un grafo que muestra sus descomposiciones y dependencias, a la vez que se puede evaluar mediante un procedimiento la satisfacción de cada uno de ellos. Sin embargo, dicho *framework* no es suficiente para realizar el análisis de Sistemas Ubicuos, ya que no contempla las características especiales de éstos, mencionadas anteriormente.

Por otra parte, existen algunos trabajos que tratan de abordar las características propias de Sistemas Ubicuos, Sistemas Conscientes o Sensibles al Contexto (*Context-aware Systems*) y Sistemas

Dinámicos: LoREM [GSB+08] trata de proponer un método para representar las adaptaciones en Sistemas Dinámicos; RE-CAWAR [SS07] propone un método de Ingeniería de Requisitos para Sistemas Conscientes del Contexto; en [JB04] [Kol05] se proponen técnicas de adquisición y representación de requisitos en Sistemas Ubicuos; UWA [Con07a] es una metodología para Aplicaciones Web Ubicuas en la que se propone una modificación a la metodología KAOS [DDMvL97] para tratar los requisitos. Sin embargo, ninguno de estos trabajos presenta un tratamiento exhaustivo y sistemático de los NFR, tal y como sería deseable dadas las características de la Computación Ubicua.

Carencias de las metodologías de diseño de servicios

El análisis de la bibliografía revela múltiples propuestas de metodologías para el análisis y diseño de servicios. Por una parte pueden encontrarse propuestas de propósito general [Ars04] [PVDH06], las cuales pueden ser útiles en el ámbito de los sistemas de escritorio, pero que presentan carencias en cuanto al tratamiento de las características propias de los sistemas ubicuos. Además, en ocasiones estas metodologías solamente dan un conjunto de guías, en lugar de proponer métodos concretos sobre cómo proceder en el diseño de los servicios. A este respecto, la propuesta de *Service Component Architecture* (SCA) [BBB+05] trata de dar un soporte más concreto y preciso, cubriendo las carencias de las metodologías anteriores; no obstante, aún presenta el inconveniente de no abordar características tales como la consciencia del contexto o la inclusión de mecanismos de adaptación. Por otra parte, existen otras metodologías de diseño de servicios [PB05] [SVP10] orientadas al diseño de sistemas ubicuos. Sin embargo, la mayoría de estas propuestas se centran en la adquisición y tratamiento del contexto mediante ontologías, en lugar de realizar una propuesta genérica de su diseño.

En el ámbito de los sistemas ubicuos existen numerosos enfoques que aplican MDA [CCAT12] [dOdPdSB09]; sin embargo, la mayoría de los trabajos encontrados en la bibliografía se centran en la transformación de modelos entre el nivel independiente de la plataforma (PIM) y el específico de la plataforma (PSM). En ninguno de los trabajos analizados relativos a sistemas ubicuos se parte del nivel independiente de la computación (CIM) para derivar el resto de modelos. Además, la mayoría de los trabajos se centran más en la generación de código para diferentes plataformas, en lugar de focalizarse en la transformación entre modelos. Por tanto, sería conveniente contar con un enfoque general que abordase las transformaciones necesarias desde el nivel CIM hasta la generación de código para distintas plataformas, cubriendo la mayor parte del ciclo de vida del software.

1.4 Objetivos

El objetivo general de esta tesis es:

> *Contribuir al avance en análisis, diseño y desarrollo de sistemas ubicuos aplicando métodos y técnicas que se pueden encontrar/idear a partir de las disciplinas de ingeniería de requisitos y del software.*

Características deseables de un método de Ingeniería de Requisitos

De manera más específica, dado que los requisitos no funcionales son cruciales para el éxito de los sistemas desarrollados, y en particular de los sistemas ubicuos dada su naturaleza dinámica y cambiante (dependiente del contexto), es necesario que los Ingenieros de Requisitos cuenten con un método que les permita tratar de manera sistemática con los requisitos y así poder derivar, eventualmente, diseños de alta calidad. En concreto, el método debería tener las siguientes características:

- Permitir el modelado abstracto de los requisitos. Ello permitirá la obtención de modelos que, eventualmente, pudieran ser transformados de manera semiautomática en otros modelos para el diseño de los sistemas ubicuos que se están analizando. Dichos modelos se construirán conforme a unos metamodelos definidos formalmente.

- Analizar los obstáculos que suponen un riesgo para la consecución de los requisitos, así como las formas de mitigarlos, haciendo que el sistema ubicuo sea más tolerante a fallos.

- Estudiar las contribuciones de diferentes decisiones arquitectónicas y de diseño a la satisfacción de los requisitos, con el objetivo de elegir de manera sistemática aquellas decisiones que cumplen los requisitos no funcionales y evitar que dicho proceso vaya dirigido por la tecnología en lugar de por la calidad.

- Representar y manejar el impacto significativo del contexto en los requisitos. De esta forma se podrá determinar qué adaptaciones son necesarias en el sistema y cuándo se deben realizar dichas adaptaciones.

- Estudiar los cambios en la priorización de requisitos basados en cambios contextuales y los *trade-offs* que implican. Así, el Ingeniero de Requisitos podrá elegir de manera sistemática y con criterio aquellas decisiones que proporcionen una mayor calidad al sistema ubicuo analizado.

Características deseables del método de Diseño

Por otra parte, teniendo en cuenta las carencias de los enfoques existentes, debe proponerse un método de diseño de sistemas ubicuos con las siguientes características:

- Proporcionar un guiado completo en el proceso de diseño, tratando de partir de la especificación de requisitos para obtener un diseño arquitectónico del sistema y detallarlo en sucesivas iteraciones.
- Tener en cuenta las características especiales de los sistemas ubicuos en el diseño de servicios.
- Promover la reutilización del software diseñado, tanto a nivel de servicio como a nivel de partes del mismo.
- Facilitar la adaptación de los servicios en base a cambios en el contexto.

1.5 Metodología

Para conseguir dichos objetivos, en esta tesis se plantea una metodología de investigación que comprende las siguientes tareas:

- Realizar un análisis de la bibliografía sobre Ingeniería de Requisitos, estudiando en primer lugar métodos de Ingeniería de Requisitos de propósito general, y posteriormente aquéllos especialmente dedicados a Sistemas Ubicuos, con el objeto de estudiar las fortalezas y debilidades de cada uno de los métodos existentes. Analizar las carencias existentes y obtener un sólido conocimiento para determinar los requisitos que debería cumplir un método de Ingeniería de Requisitos apropiado para Sistemas Ubicuos.
- Realizar un análisis de la bibliografía sobre Diseño de servicios, estudiando en primer lugar métodos de Diseño de propósito general, y posteriormente aquéllos especialmente dedicados a Sistemas Ubicuos. Se analizarán las ventajas e inconvenientes de cada uno de ellos. Se extraerán las carencias que los métodos existentes presentan, para tratar de determinar las características que debería cumplir un método de Diseño de servicios apropiado para Sistemas Ubicuos.
- Realizar un análisis de la bibliografía sobre adaptación en sistemas ubicuos, orientado a determinar qué técnicas

son las más apropiadas para determinar qué adaptaciones deben realizarse, qué mecanismos deben emplearse para ejecutarlas y a qué niveles pueden realizarse adaptaciones.

- Realizar un análisis de la bibliografía sobre Ingeniería dirigida por modelos, con el objetivo de determinar qué técnicas existen para la realización de correspondencias entre modelos y la obtención de las reglas de transformación que las realicen. Asimismo, se estudiarán los estándares existentes que permiten la especificación de estas transformaciones, tanto de modelo a modelo, como de modelo a texto para la generación de código.

- Proponer un método de Ingeniería de Requisitos para Sistemas Ubicuos a partir del estudio realizado, basado en modelos conforme a un metamodelo definido formalmente. Dicho método debe permitir la representación y el tratamiento sistemático de los requisitos no funcionales, prestando especial atención a los principales rasgos que de forma inequívoca caracterizan a la computación ubicua.

- Proponer un conjunto de patrones de requisitos que capturen algunos de los problemas observados que ocurren con frecuencia en diferentes dominios de aplicación. Los patrones propuestos permitirán la reutilización del conocimiento generado de una manera sistemática y que puedan ser aplicados de manera complementaria al método de Ingeniería de Requisitos propuesto.

- Proponer un método y un metamodelo de Diseño para Sistemas Ubicuos a partir del estudio de la bibliografía realizado, de manera que que permita materializar los requisitos capturados con el método de Ingeniería de Requisitos propuesto, o con algún otro método de Ingeniería de Requisitos que obtenga una especificación completa para sistemas ubicuos.

- Proponer un conjunto de transformaciones entre modelos que permitan aplicar un enfoque dirigido por modelos para realizar la transformación semiautomática entre el modelo de requisitos obtenido, el modelo de diseño y una implementación del mismo en diferentes plataformas.

- Realizar la definición de los métodos propuestos conforme a un modelo de especificación de procesos de software, de manera que los métodos propuestos puedan ser implementados con facilidad en herramientas de gestión de procesos,

así como que puedan ser modificados y extendidos para adaptarse de una manera sencilla a dominios específicos de aplicación.

- Mostrar la aplicabilidad del método propuesto en un caso de estudio relativo al análisis de los requisitos, diseño e implementación de un sistema de posicionamiento, el cual constituya un ejemplo representativo de los Sistemas Ubicuos que pueden diseñarse.

1.6 Organización del documento

Organización

El contenido de esta tesis doctoral se encuentra organizado de la siguiente forma (Figura 1.3):

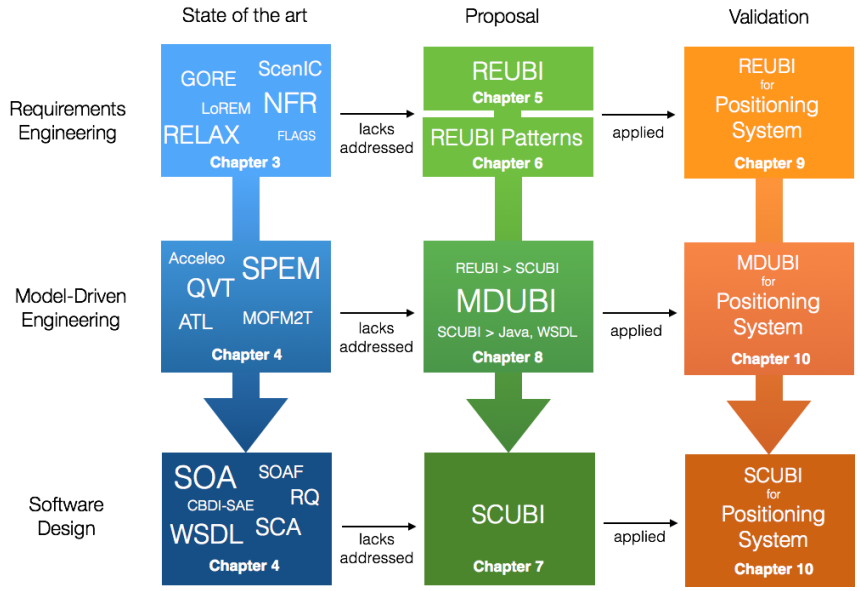

Figura 1.3: Organización del documento.

- En la Parte II se presenta una revisión bibliográfica a modo de análisis del estado de la técnica:
  - El Capítulo 3 presenta una revisión bibliográfica, examinando un amplio número de métodos de Ingeniería de Requisitos que de alguna forma se encuentran relacionados con este trabajo. Se presentan métodos de propósito general para el análisis de sistemas software, así como otros específicos para sistemas ubicuos, dinámicos, adaptativos y conscientes del contexto. Posteriormente, se realiza un análisis comparativo de las

ventajas e inconvenientes de cada método, en relación con la satisfacción de requisitos y la incorporación de características particulares de los Sistemas Ubicuos. Por último, se identifican las características ideales que debería tener un método de Ingeniería de Requisitos para Sistemas Ubicuos.

- El Capítulo 4 continúa la revisión bibliográfica, analizando los diferentes enfoques existentes para realizar Ingeniería dirigida por modelos; las propuestas existentes para la especificación de procesos, particularmente en cuanto a procesos de ingeniería del software; los métodos y estándares existentes para el diseño de servicios en el marco de una Arquitectura orientada a Servicios; y los trabajos sobre adaptación de software en Sistemas Ubicuos. En cada uno de estos apartados se analizan las ventajas e inconvenientes de cada propuesta, así como se extraen las carencias existentes para tratar de suplirlas.

- En la Parte III se presenta la propuesta realizada a partir de las carencias detectadas en el análisis bibliográfico:

  - El Capítulo 5 toma como punto de partida el análisis final del capítulo 3 que resulta del estudio del estado de la técnica con el objetivo de idear un método de Ingeniería de Requisitos para Sistemas Ubicuos, al que hemos denominado REUBI. Este método aborda los rasgos característicos de la Computación Ubicua, como pueden ser la dinamicidad del entorno, la consciencia del contexto, la heterogeneidad de tecnologías existentes o la variabilidad en la priorización de los requisitos. Se propone la construcción de un conjunto de modelos, los cuales se definen formalmente a partir de un metamodelo, así como se proporciona un procedimiento que permite evaluar, *a priori*, la satisfacción de los requisitos.
  - El Capítulo 6 presenta un conjunto de situaciones recurrentes en diferentes dominios de aplicación para capturar el conocimiento generado durante la etapa de Ingeniería de Requisitos en términos de patrones de requisitos, con el propósito de que puedan ser reutilizados posteriormente y aplicados de manera complementaria al método REUBI.
  - El Capítulo 7 propone un método de diseño de servicios basados en componentes para Sistemas Ubicuos,

el cual hemos denominado SCUBI, que trata de tener en cuenta las características de consciencia del contexto o adaptación, entre otras, para incorporarlas en el proceso de diseño y así obtener servicios que cumplan con los requisitos especificados mediante la aplicación del método REUBI.

- El Capítulo 8 propone de un enfoque dirigido por modelos para el desarrollo de sistemas ubicuos. Este enfoque, denominado MDUBI, propone un conjunto de transformaciones que permiten obtener un modelo de diseño basado en SCUBI de manera semiautomática a partir de una especificación de requisitos obtenida mediante el método REUBI. Adicionalmente, se presenta un conjunto de transformaciones para obtener modelos específicos para plataformas concretas, como Java o WSDL, junto con las correspondientes estrategias de generación de código.

- En la Parte IV se aplica la propuesta realizada al desarrollo de un sistema ubicuo:

  - El Capítulo 9 presenta un sistema real donde se intenta validar el método de Ingeniería de Requisitos propuesto. Se trata de analizar los requisitos de un sistema de posicionamiento para su correcto diseño, de manera que se tengan en cuenta la variedad de técnicas, métodos y tecnologías de posicionamiento, las diferentes contribuciones que cada una de ellas tiene para la satisfacción de determinados requisitos no funcionales, y las situaciones que provocan un cambio de prioridad en la satisfacción de los requisitos.
  - El Capítulo 10 continúa el proceso iniciado en el capítulo anterior y trata de validar el método de Diseño propuesto. Se trata de obtener el diseño del sistema de posicionamiento en términos de un servicio mediante la aplicación de las transformaciones entre modelos propuestas, y complementando el proceso con la aplicación del método SCUBI.

- Por último, en la Parte V, el Capítulo 11 muestra las conclusiones acerca del trabajo realizado. Además, se presentan posibles mejoras y ampliaciones de la propuesta, así como las líneas de trabajo futuro.

- Finalmente, en el Apéndice A puede encontrarse una lista con las publicaciones científicas realizadas durante el trans-

curso de esta tesis doctoral. En el Apéndice B se muestra la implementación de las reglas de transformación propuestas en lenguaje ATL.

1.7 Cómo leer este documento

Este documento se ha organizado de manera que el lector pueda repasar el estado de la técnica en la Parte II (Capítulos 3 y 4), continuar con la propuesta realizada en la Parte III (Capítulos 5, 6, 7 y 8) y validar la propuesta aplicándola a un sistema real en la Parte IV (Capítulos 9 y 10).

No obstante, dada la extensión y la cantidad de nuevos conceptos propuestos, puede ser conveniente comenzar la lectura del documento por la aplicación de la propuesta en la Parte IV para que el lector tenga una idea global de la estructura de la misma, sus resultados y la correspondencia de los conceptos propuestos con elementos de un sistema real. Una vez realizada la lectura de dichos capítulos, el lector puede analizar los detalles de la propuesta en los capítulos centrales de este documento en la Parte III.

2 INTRODUCTION

Índice

2.1 Introduction

Evolution of Computer Science

In the last years, Computer Science has evolved since its former applications oriented to perform repetitive tasks and massive data storage in mainframes, operated by several people, to the era of Personal Computers (PCs) that are nowadays in most houses. From this point, a third period has begun, influenced and known as **Ubiquitous Computing** (Figure 2.1). **Ubiquity** is the property of an entity of existing or being everywhere at the same time.

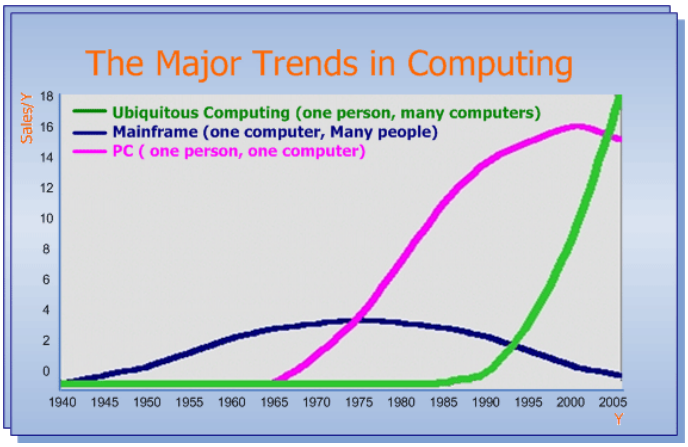

Figura 2.1: Evolution of the different stages of Computer Science.

Interaction Paradigms

Ubiquitous Computing is a **Human-Computer Interaction (HCI) Paradigm** which is gaining more and more acceptance

thanks to the recent advances in this area in the last years (development of mobile technologies, Internet, appearance of numerous sensors and actuators, deployment of intelligent environments...). There are different HCI Paradigms:

- **Personal or Desktop Computing:** it is the predominant interaction paradigm until the past few years, consisting of the interaction between a user with his/her computer (PC, laptop). Usually, the user interacts with the computer besides the environment, although the development of wireless technologies enables his/her mobility and contact with the environment.

- **Virtual Reality:** it consists of the computer simulation of a realistic and dynamic world which responds to the user's orders. The key features of this paradigm are real-time interactivity and immersion feelings to become part of what is happening in the scene. Typically, it involves the employment of specific hardware, like virtual reality helmets, polarized glasses, haptic devices... Its main drawback is the high cost of both devices and development, and produces the user to become tired.

- **Augmented Reality:** it tries to reduce the interactions with the computer by using information from the environment as an implicit input. There are two variants of this paradigm: the result of applying virtual reality to the real world and the use of devices that augment the reality and interact directly with it.

- **Ubiquitous Computing:** as proposed by Mark Weiser [Wei93], it tries to extend the computation capabilities to the user's environment, enabling it to be everywhere in the form of small devices which allow easy interactions and are interconnected to each other. Its design and localization must be specifically devised to the interaction task; thus, computing is no longer focused in a single point (the desktop) to become spread in the environment.

Ubiquitous Computing

Ubiquitous Computing (or Pervasive Computing), was first described by Mark Weiser in 1991 [Wei91]. The essence of his vision was the creation of environments full of computation elements with communication capabilities, and in a transparent way for people. One of he most important goals is the integration of computational devices into every day life object, enabling users to focus on the tasks they perform instead of the tools they

must use. Sending computation to the background entails (i) the integration of technology into daily use objects (figure 2.2), and (ii) those objects must not interfere with the activities they are used for, giving a more comfortable, easy and useful employment of them.

Figura 2.2: Integration of technology into a cup.

Features of Ubiquitous Computing

Therefore, Ubiquitous Computing incorporated four new concepts:

- **Effective use of perceptive spaces:** it is based on sensing the person's state and his/her need, inferred from that state, regardless of the location (office, meeting room, class, home, car...). The perceptive space appears when several smart devices are located in the same physical location and interact collaboratively to support the users' activities that take place at that location. Home automation is the most popular application of this concept.
- **Invisibility:** although nowadays technology is far from complete disappearance from user's awareness, it is necessary to keep in mind the idea of minimum distraction when these systems are designed. Invisibility requires drastic changes in the kind of interfaces that are used to communicate to computers, being necessary new technologies of voice recognition, gestures, oral and written natural language understanding, speech synthesis, and graphical representations.

- **Localized scalability:** the concept of locality of services in Ubiquitous Computing is fundamental in contrast to the universality of services in the Internet. Users have capabilities which are associated to the context where they are immersed; thus, there is a lack of sense if, for instance, home control applications situated at the user's home are trying to sense the user's needs when s/he is working at the office.

- **Masking uneven conditioning:** depending on the available infrastructure and technological development, distribution of services may be uneven. In this situation, the invisibility principle may not be fulfilled since the user would be able to detect transitions between environments. This requirement is currently the furthest from the ideal situation; Ubiquitous Systems may be isolated and without continuity from each other.

Research in Ubiquitous Computing

In the field of research in Ubiquitous Systems, several works and developments of the concepts proposed by the Ubiquitous Computing paradigm can be found. Several relevant works are mentioned here. The *Proximity Toolkit* [MDMBG11] is a development kit to ease the creation of systems where the interaction is based on the user's proximity to certain devices or other users. The framework provides proxemic information (orientation, distance, motion, identity and localization) in the form of an API so that the developer can create the required interaction in his/her system. Moreover, several applications of this toolkit are shown, such as an advertising system with user's attention demand or a musical experience based on the proximity to a device.

Avrahami *et al.* [AYL11] explore the possibility to incorporate Ubiquitous Systems inside the cars. In their study, they show statistics about the number of traffic accidents causing injuries to the occupants of the vehicle as a consequence of carrying loose objects inside the cabin. In order to try to reduce this problem, they propose a cameras and sensors based system to infer which objets are loose and warn the driver about a potential danger.

Kortuem *et al.* [KKFS10] study the possibility of transforming daily life objects into smart objects to help people perform different tasks. The present a classification taxonomy for these objects where they analyze three design dimensions: awareness (ability of the device to recognize real-world events), representation (programming abstractions of the device) and interaction (ability of the device to converse with the user in terms of input, output, control or feedback). A case study is presented where road-patching workers carry a wearable device to inform them

about potential health risks derived from a long exposure to the employment of certain machines.

Ghose *et al.* [GSB+13] present a health monitoring system for the elderly and people with chronic health problems, calle UbiHeld. In order to do this, given the spread of smartphones, they make use of this platform in order to access the health monitoring system everywhere. To analyze the person's movements, the combine smartphones with Kinect instead of videocameras, to reduce the intrusions in the user's privacy. Last, in order to analyze the patient's mental state, they have access to the user's social networks feed.

2.2 Problem statement

Lacks in the Research about Ubiquitous Systems

From the analysis of these and many other Ubiquitous Systems available in the bibliography about research in Ubiquitous Computing, it can be inferred that the attention is focused in the development of the functionality and the inclusion of the newest technologies in the systems to be developed, while the non functional properties of these systems receive little attention of the designers and engineers [MCY99] [CNYM00]. Some non functional requirements, such as the efficiency of the system or the usability, are usually taken into account in the development of Ubiquitous Systems, while others, such as reusability or scalability, are pushed to a second place, or they are not even addressed. Moreover, most of these systems are built in an *ad-hoc* manner, without a methodological support guiding the designer and allowing him/her to make decisions in a systematic way, both for the analysis of the requirements of the systems, and the later design.

Non Functional Requirements

Although Non Functional Requirements (NFR) mostly determine the quality of the systems [MCY99], and this is particularly true in the scope of Ubiquitous Computing given its special features, few works exist in the bibliography to enable a systematic treatment of the NFRs in Ubiquitous Systems. Among the special features of this paradigm related to NFRs, we can highlight:

Special features of Ubiquitous Systems

- **Context-awareness:** Ubiquitous Systems monitor certain variables of the execution environment that are of interest to try to adapt and fulfill the quality requirements that are expected [Fer06].
- **Dynamicity of the environment:** related to the previous bullet, the deployment environments of Ubiquitous Systems are variable and always changing. Such changes do not

only affect to the functional part of the system (for instance, offering different services depending on the location), but also entail changes to maintain the quality of the system [AFN].

- **Heterogeneity of technological solutions:** the wide variety of existing technological developments, each one of them with different quality properties, forces the designer to deal with a selection process that, usually, is simply guided by the incorporation of the latest development, instead of counting with a systematic selection process based on the election of the technology that presents the best quality properties [RLGBC10].

- **Adaptivity and customization:** besides taking into account the characteristics of the execution environment of the Ubiquitous System, this kind of systems must take into consideration the user's preferences, both at a functional level to offer those services that are of interest, as non functional, to provide them with the quality properties that the users expect from them [DBS+02].

- **Variability in the prioritization:** in traditional software systems, requirements receive a priority that does not change during the execution of the system. However, in this kind of systems, priority can change along the execution of the system; so the system must be capable of determining the changes that must occur in its internal structure to provide a quality service to the new priorities [SLK06].

Reutilization of knowledge in Requirements Engineering

Besides, in many situations, counting on a Requirements Engineering method that guarantees a systematic treatment of NFRs is not enough to perform this task successfully. In fact, it is a difficult task and requires a deep and broad knowledge about the system application domain, specially taking into account the special features of this paradigm stated above. Moreover, this knowledge has remained implicit or informally captured, losing the opportunity to reuse it and leading, in some cases, to misunderstandings in the communication.

Requirements Patterns

There are recurring situations where similar solutions can be applied in order to model their requirements and guarantee the fulfillment of their quality properties. Undoubtedly, capturing knowledge about those problems and usual solutions for later reuse would be highly desirable. One of the possible ways to perform this task is the definition of requirements patterns. The use of patterns in the Requirements Engineering stage is not new

[SHC+10], but it has received little to no attention in the scope of Ubiquitous Computing. In this way, capturing information about recurring problems in this stage in terms of patterns and their reuse complementarily to the Requirements Engineering method could bring better results to the satisfaction of NFRs.

Design of Ubiquitous Systems

Once the analysis and specification of requirements with a systematic treatment of NFRs is completed, the next step in the software lifecycle is the realization of a design to support those requirements. Analyzing the bibliography of development projects in the field of Ubiquitous Computing, it can be found that, mostly, those developments are guided by the election of concrete technologies (usually middleware solutions), instead of making a platform-independent design, deferring the election of the technology the most suitable one has been studied.

Use of SOA in Ubiquitous Systems

Because of the features of Ubiquitous Computing, one of the most widely used paradigms in the design of systems is the paradigm of Service-Oriented Architecture (SOA). This paradigm enables the creation of services with low coupling and high reusability [Erl08], in a way that different services can be instanced in distinct environments and be available in a local manner in those locations where they make sense to be, easing the satisfaction of the locality principle, and moreover, get discovered dynamically, improving the users' mobility. For these reasons, it seems appropriate to count on methodologies oriented to the design of services for Ubiquitous Systems.

Service adaptation

Moreover, it is necessary to keep in mind the adaptation needs of these services based on context changes. A typical adaptation process in Ubiquitous Computing [KPP10] consists of three steps: (i) context acquisition; (ii) reasoning about the adaptations to be made; and (iii) applying the selected adaptation. Consequently, the design of services should consider this process in order to incorporate adaptation to the design as easy as possible.

Model-driven Engineering

Finally, in order to create a engineering-oriented process that automatize those task that can be automatically performed, some authors propose the employment of Model-driven Engineering (MDE) and particularly Model-driven Architecture (MDA) [Gro03b].

2.3 Motivation

Requirements Engineering in Ubiquitous Systems

The NFR Framework, proposed by Chung *et al.* [CNYM00] is a framework that allows the designer to deal in a systematic way with the NFRs. It presents the concept of *softgoal* as an objective for which there is not a criterion to determine its satisfaction

precisely, but the contributions of the design decisions must be studied in order to determine its degree of satisfaction. In this way, NFRs can be modeled as a graph that shows their decomposition and dependencies, at the same time that there is a procedure to study the satisfaction of each of them. However, this framework is not enough to perform the analysis of Ubiquitous Systems, since it does not take into consideration its special features: context-awareness, dynamicity or prioritization variability, among others.

On the other hand, there are several works that aim to address the concrete characteristics of Ubiquitous, Context-aware or Dynamic Systems: LoREM [GSB+08] proposes a method to represent the adaptations in Dynamic Systems; RE-CAWAR [SS07] proposes a Requirements Engineering method for Context-aware Systems; in [JB04] [Kol05], several techniques for acquisition and representation or requirements in Ubiquitous Systems are presented; UWA [Con07a] is a methodology for Ubiquitous Web Applications in which a modification of the KAOS methodology [DDMvL97] is proposed to deal with requirements. However, none of these works presents an exhaustive and systematic treatment of NFRs, as it would be desirable given the features of Ubiquitous Computing.

Lacks in the methodologies for designing services

The analysis of the bibliography reveals many proposals of methodologies for the analysis and design of services. On the one hand, general purpose proposals [Ars04] [PVDH06] can be found, which can be useful in the scope of desktop systems, but have some lacks in the incorporation of the features of Ubiquitous Systems. Moreover, some of these methodologies only provide a set of guidelines, instead of proposing concrete methods about how to proceed in the design of services. To this end, the Service Component Architecture (SCA) [BBB+05] aims to provide a more concrete and precise support, covering the lacks of the previous proposals; nevertheless, it still has the drawback of not addressing features such as context-awareness or adaptation mechanisms. In the other hand, there exist some other methodologies for service design [PB05] [SVP10] oriented to Ubiquitous Systems. However, most of these proposals are focused in the acquisition and management of context by means of ontologies, instead of making a generic proposal for their design.

In the field of Ubiquitous Systems there are numerous approaches that use MDA [CCAT12] [dOdPdSB09]; however, most of the works found in the bibliography are focused in the transformation of models between a platform independent level (PIM) and a platform specific level (PSM). None of the analyzed works relative to Ubiquitous Computing starts from the computation independent level (CIM) in order to derive the rest of models.

Consequently, it would be convenient to count on a genera approach to address the necessary transformations from the CIM level to the code generation for different platforms, covering most of the software lifecycle.

2.4 Objectives

The overall goal of this research thesis is:

> *To contribute to the advance in analysis, design and development of ubiquitous systems with the application of methods and techniques that can be found/envisioned from the requirements and software engineering disciplines.*

Desirable features of a Requirements Engineering method

In a more specific way, given that NFRs are crucial for the success of the developed systems, and particularly in Ubiquitous Systems given their changing, context-dependent and dynamic nature, it is necessary that Requirements Engineers count on a method to enable a systematic treatment of requirements and, eventually, be able to derive a high quality design. More precisely, the method should have the following features:

- Allowing the abstract modeling of requirements. It will enable obtaining models that, eventually, could be transformed in a semi automatic manner into other models for the design of the Ubiquitous Systems under analysis. Such models would be built conforming to some formally defined metamodels.
- Analyzing the obstacles that thwart the fulfillment of the requirements, as well as the ways to mitigate them, making the Ubiquitous System more fault-tolerant.
- Studying the contributions of different architectural and design decisions to the satisfaction of the requirements, in order to choose in a systematic way those decisions which satisfy the NFRs and avoid the process to be driven by the technology instead of the quality.
- Representing and handling the significative impact of context in the requirements. In this way, it would be possible to determine which adaptations are necessary and when to perform them.
- Studying the changes in the prioritization of requirements based on contextual changes and the trade-offs they entail.

Thus, the Requirements Engineer could choose in a systematic way and with some criteria those decisions that provide a higher quality to the Ubiquitous System under analysis.

Desirable features in a Design method

Besides, considering the lacks of the existing approaches, a method with the following features for the design of Ubiquitous Systems should be proposed:

- Guiding engineers in the design process, aiming to start from the requirements specification in order to obtain an architectural design and detail it in continuous iterations.
- Taking into account the special features of the Ubiquitous Systems in the design of services.
- Promoting reuse of the designed software, both at a service level, as well as in parts of it.
- Easing the adaptation of services in terms of contextual changes.

2.5 Methodology

In order to reach this goal, a research methodology is proposed, comprising the following tasks:

- Analyzing the bibliography about Requirements Engineering, studying in the first place general purpose Requirements Engineering methods, and later those specially devised for Ubiquitous Computing, in order to study the strengths and weaknesses of each one of the existing methods, unearthing their lacks and obtaining a solid knowledge to determine the features that an appropriate Requirements Engineering method for Ubiquitous Systems should have.
- Analyzing the bibliography about Design of services, studying in the first place general purpose design methods, and later those specially devised to Ubiquitous Computing, in order to study the strengths and weaknesses of each one of them and extracting the lacks of the proposals, in order to determine the features of an appropriate Design method for Ubiquitous Systems should have.

- Analyzing the bibliography about adaptation in Ubiquitous Systems, oriented to determine which techniques are the most suitable to reason which adaptations must be applied, which mechanisms should be employed to execute them; and which levels can be affected by adaptations.
- Analyzing the bibliography about Model-driven Engineering, with the purpose of identifying the existing techniques for mapping between models and obtaining the transformation rules that implement them. Besides, the existing standards to specify these transformations, both from model to model, and from model to text for code generation, will be studied.
- Proposing a Requirements Engineering method for Ubiquitous Systems from the conducted study, based in models created conforming a formally defined metamodel. Such method must allow the representation and systematic treatment of NFRs, paying spacial attention to the main features that are bound to Ubiquitous Computing: dynamicity, context-awareness, variability in the prioritization of requirements, environment heterogeneity...
- Proposing a set of requirements patterns that capture some of the observed recurring problems which happen frequently in different application domains, in order to enable the reuse of the generated knowledge in a systematic manner, and which can be applied complementarily to the Requirements Engineering method.
- Proposing a method and a metamodel for the design of Ubiquitous Systems stemming from the conducted study of the bibliography, incorporating the special features of this kind of systems and allowing to realize the captured requirements in a specification resulting from the application of the proposed Requirements Engineering method.
- Proposing a set of transformations between models that enable the application of a model-driven approach in order to perform semi automatic transformations between the requirements model, the design model and an implementation in different platforms.
- Defining the proposed methods conforming a software process specification model, in order to implement them easily in process management tools, as well as to enable their modification and extension to be adapted to specific application domains.

- Validating the proposal by its implementation, employing those tools that are more suitable to this end.
- Showing the applicability of the proposed methodology in a case study of the requirements engineering, design and implementation of a positioning system, which constitutes a representative example of a Ubiquitous System that can be designed.

2.6 Organization of the document

Organization

The content of this doctoral thesis is organized as follows (Figure 2.3):

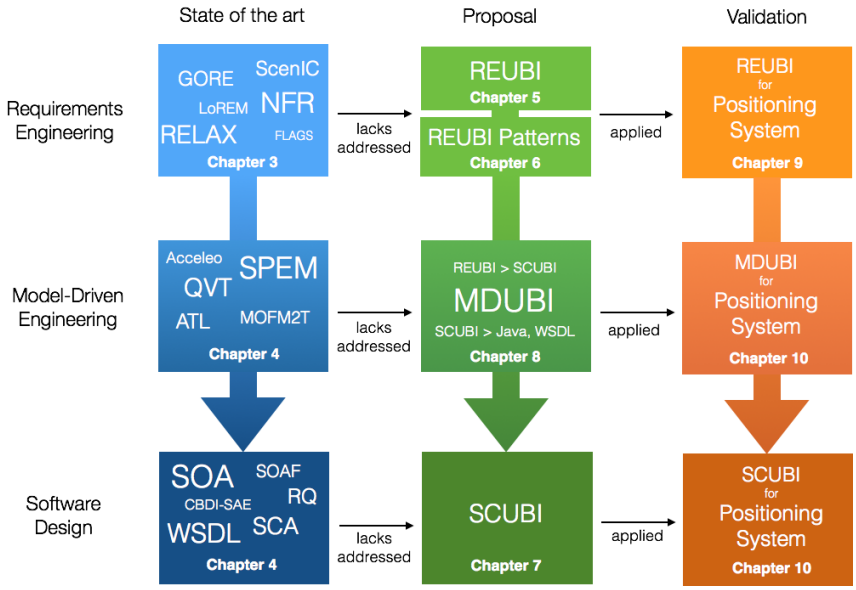

Figura 2.3: Organization of the document.

- Part II presents a bibliographical review as an analysis of the state of the art:
  - Chapter 3 presents a bibliographical review, examining a wide number of Requirements Engineering methods which are related with this research work. General purpose methods for analyzing software systems, as well as specific ones for Ubiquitous, Dynamic, Adaptive and Context-aware Systems are presented. Later, a comparative analysis is conducted showing the strengths and weaknesses of each method, relating them with the requirements and specific features

of Ubiquitous Computing. Last, the ideal features of a Requirements Engineering method for Ubiquitous Systems are outlined.

- Chapter 4 continues the bibliographical review, analyzing different approaches for Model-driven Engineering; existing proposals for the specification of processes, particularly regarding software engineering processes; methods and standards for the design of services in the scope of a Service Oriented Architecture; and the research works about software adaptation in Ubiquitous Systems. In each of these sections, strengths and weaknesses of each approach are summarized, as well as their lacks are studied in order to try to cover them in the proposal.

- Part III presents the proposal made from the identified shortcomings in the bibliographical review:

  - Chapter 5 starts from the analysis at the end of chapter 3 resulting from the state of the art with the goal of devising a Requirements Engineering method for Ubiquitous Systems, which has been named REUBI. This method addresses the main features of Ubiquitous Computing, such as dynamicity of the environment, context awareness, heterogeneity of technologies or variability in the prioritization of requirements. It proposes the construction of a set of models, which are formally defined conforming to a metamodel, as well as it provides an evaluation procedure to determine the satisfaction of the requirements.
  - Chapter 6 presents a set of recurring situations in different application domains in order to capture the generated knowledge during the Requirements Engineering stage in terms of requirements patterns, with the purpose of being reused in later developments and applied as a complement to the REUBI method.
  - Chapter 7 proposes a service design method based on components for Ubiquitous Systems, which has been named SCUBI. It aims to take into consideration the features of context-awareness o adaptation, among others, in order to incorporate them in the design process and obtaining services that fulfill the requirements specified by the application of the REUBI method.
  - Chapter 8 proposes a model-driven approach for the development of Ubiquitous Systems. This approach,

called MDUBI, proposes a set of transformations in order to obtain a design model based on SCUBI in a semi automatic manner starting from a requirements specification obtained with the application of REUBI. Additionally, a set of transformations to obtain models for target platforms, such as Java or WSDL, is presented, as well as the corresponding code generation strategies.

- Part IV presents an application of the proposal in order to develop a ubiquitous system:
  - Chapter 9 presents a real system to validate the proposed Requirements Engineering method. It aims to analyze the requirements of a positioning system for its correct design, in a way that the variety of techniques, methods and positioning technologies are considered, as well as their contribution to the satisfaction of certain NFRs, and the situations that cause a change in the priority of the requirements.
  - Chapter 10 continues the application from the previous chapter and aims to validate the proposed Design method. It aims to obtain the design of the positioning system in terms of a service by means of the application of the proposed model transformations, and complementing the process with the application of the SCUBI method.
- Last but not least, in Part V, Chapter 12 summarizes the main conclusions obtained during this research work. Additionally, some improvements and extensions of this work are outlined, as well as the future work.
- Finally, appendix A lists the scientific publications made during the realization of this doctoral thesis. Appendix B depicts the implementation of the proposed transformation rules in ATL language.

2.7 How to read this document

This document was organized so that the reader can review the state of the art in Parte II (Chapters 3 and 4), continue with the proposal in Part III (Chapters 5, 6, 7 and 8) and validate the proposal by its application to a real system in Part IV (Capítulos 9 and 10).

Nevertheless, given the extension and amount of newly proposed concepts, it could be convenient to start the read of this document from the application of the proposal in Part IV so that the reader can acquire a general idea of it, its results and the correspondence of the proposed concepts to elements in a real system. Once those chapters are completed, the reader can analyze the details of the proposal on the main chapters of this document in Part III.

Parte II

Estado de la técnica

3 INGENIERÍA DE REQUISITOS Y SISTEMAS UBICUOS

Índice

3.1 Introducción

En este capítulo se pretenden estudiar diferentes métodos de Ingeniería de Requisitos existentes, tanto de propósito general como aplicados a sistemas ubicuos o adaptativos. Nuestro propósito es analizar la idoneidad de estos métodos y técnicas para extraer y modelar los requisitos de un sistema ubicuo y, posteriormente, realizar el análisis y diseño del sistema software a implementar a partir de los resultados obtenidos. Puesto que el objetivo central es obtener software que cuente con propiedades de calidad, nos centraremos en la atención que le prestan cada uno de los métodos a la adquisición y representación de los requisitos no funcionales de los sistemas. Asimismo, estudiaremos la capacidad de cada una de las propuestas para dotar al sistema de mecanismos de adaptación, dado que una de las características principales de la computación ubicua es la adaptación al usuario y a un entorno cambiante.

Definición de Ingeniería de Requisitos

Existen numerosas definiciones de **Ingeniería de Requisitos** en la bibliografía, de entre las cuales destacamos una, atribuida a Pamela Zave [Zav97]:

Definición de Ingeniería de Requisitos

> « *La Ingeniería de Requisitos es la rama de la Ingeniería del Software que se ocupa de la relación entre los objetivos del mundo real y las funciones y restricciones de los sistemas software. También se ocupa de la relación entre estos factores y la especificación del comportamiento del software, y de su evolución a lo largo del tiempo y entre familias de software.*»

Bases de la Ingeniería de Requisitos

Esta rama de la Ingeniería del Software encuentra sus bases en otras disciplinas [NEoo], tales como **Psicología Cognitiva**, para tratar de entender las dificultades de la gente para describir sus necesidades; **Antropología**, en tanto que se realizan observaciones de las actividades humanas; **Sociología**, para estudiar los cambios culturales debidos a la informatización; **Lingüistica**, para evitar ambigüedades en las especificaciones y proporcionar comprensibilidad; y **Filosofía**, para entender las creencias de los *stakeholders* (epistemología), determinar qué es observable en el mundo (fenomenología), y conocer cuáles son las verdades objetivas (ontología).

Requisitos Funcionales y Requisitos No Funcionales

Requisitos funcionales vs. requisitos no funcionales

La distinción tradicional entre requisitos funcionales y no funcionales muestra que los primeros se centran en *qué* debe realizar el sistema, y los últimos en el *cómo* se realizan dichas funciones. Mientras que los requisitos funcionales tienen un ámbito claro y bien definido, y puede decidirse sin ambigüedad si son o no satisfechos, esto no ocurre con los requisitos no funcionales, tal como se detalla a continuación.

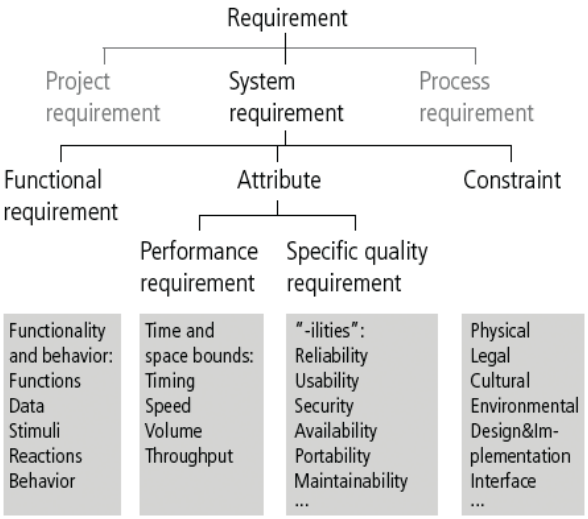

Figura 3.1: Taxonomía de Requisitos según [Glio8].

Ámbito NFRs

En primer lugar, el ámbito de aplicación de los NFR no está claramente definido. Existen requisitos no funcionales que se encuentran focalizados en partes concretas del software, por lo que su ámbito es local; sin embargo, hay otros requisitos que deben ser tratados en diferentes partes del software, es decir, son requisitos globales.

Definición NFRs

Por otra parte, los requisitos no funcionales se definen habitualmente empleando términos tales como *propiedad*, *característica*, *atributo de calidad*, *restricción* o *performance*. Sin embargo, la definición de estos términos no está clara y resulta ambigua, dando lugar a cierta divergencia entre dichos términos en cuanto a su definición, ámbito y aplicabilidad (si son requisitos del producto, del proceso o del proyecto).

Representación NFRs

Por último, existen problemas de representación de los requisitos no funcionales. Por una parte, la manera en la que se representan puede llevar a confusión. En efecto, si expresamos un requisito de seguridad como «*El sistema prevendrá cualquier acceso no autorizado*» es un requisito no funcional. Sin embargo,

el mismo requisito expresado como «*El sistema dará acceso sólo al proporcionar el nombre de usuario y la contraseña*» puede ser considerado como un requisito funcional. Por otra parte, existen problemas sobre dónde recoger estos requisitos. En ocasiones se recogen en especificaciones independientes de los requisitos funcionales; en otras, tales como los casos de uso UML, aparecen recogidos conjuntamente.

Dimensiones de los Requisitos

Estos problemas son analizados por Glinz [Glio8], quien propone como solución suavizar la distinción entre requisitos no funcionales, y caracterizar cada requisito a lo largo de cuatro dimensiones, que pueden tomar los siguientes valores:

- **Representación:** operacional, cuantitativo, cualitativo y declarativo.
- **Satisfacción:** *hard*, *soft*.
- **Tipo:** función, datos, *performance*, cualidad específica o restricción.
- **Rol:** prescriptivo, normativo o presuntivo.

Dado que esta distinción puede resultar demasiado drástica, el autor [Glio8] propone una distinción más simplificada (Figura 3.1), en la que se clasifican los requisitos como funcionales, requisitos relacionados con rendimiento (*performance*, aquéllos que se pueden medir objetivamente, mediante tiempo, espacio...), calidad (relativos a la usabilidad, mantenibilidad, extensibilidad...), y restricciones (limitaciones impuestas al sistema).

Niveles de Ingeniería de Requisitos en Sistemas Adaptativos

Mientras que la Ingeniería de Requisitos para sistemas tradicionales se centra fundamentalmente en determinar las posibles entradas para un sistema y proporcionar respuestas ante dichas entradas, cuando se trata de sistemas dinámicos, el foco de atención debe centrarse en los tipos de entradas que pueden recibirse y los tipos de salidas que se pueden producir.

Niveles de Ingeniería de Requisitos

Si consideramos un sistema preparado para adaptación, S_{AR} (*adapt-ready system*), podemos notar por S_i cada uno de los comportamientos del sistema dinámico, y por D_i el dominio de S_i, es decir, el conjunto de entradas que son admisibles cuando el sistema está en S_i. Una vez establecido este modelo, Berry *et al.* [BCZ05] proponen cuatro niveles de abstracción sobre la Ingeniería de Requisitos para sistemas adaptativos (en tiempo de ejecución), de manera que un nivel es el metanivel del inmediatamente inferior. Los niveles que se distinguen son:

- **Nivel 1:** el ingeniero de requisitos debe capturar y elaborar la información disponible sobre el dominio. Debe decidir la funcionalidad que proporcionará el sistema, las alternativas existentes y realizar su especificación. Es decir, en este nivel se deben determinar cuáles son los comportamientos S_i del sistema, su dominio asociado D_i, y las salidas que se producen.

- **Nivel 2:** es el único realizado por el sistema dinámico. En este nivel, el sistema determina si se debe realizar una adaptación o no. Para ello, ante la última entrada recibida I, debe comprobar si pertenece o no a D_i, el dominio de las posibles entradas del sistema en el estado S_i. En caso negativo, hay que determinar el dominio D_{i+1}, y su correspondiente S_{i+1}, para modificar el comportamiento del sistema y adoptarlo.

- **Nivel 3:** el ingeniero de requisitos debe estudiar tres aspectos fundamentales: (i) cómo se determina D_{i+1} a partir de la entrada I, (ii) cómo obtener S_{i+1} a partir de D_{i+1}, y (iii) cómo se modifica el comportamiento del sistema para adoptar S_{i+1}. Para ello, debe explorar los posibles dominos y relacionarlos con las condiciones del entorno; estudiar las posibles adaptaciones ante nuevas entradas; y analizar las condiciones bajo las que la adaptación debe aplicarse.

- **Nivel 4:** corresponde a la investigación en técnicas y mecanismos de adaptación generales que den soporte al nivel 3.

3.2 *Goal-based Requirements Engineering*

La Ingeniería de Requisitos basada en Objetivos (*Goal-based Requirements Engineering*) propone el concepto de objetivo (*goal*) como concepto central en torno al cual se desarrolla el proceso de adquisición, elicitación y modelado de requisitos. Es un concepto que se ha ido incorporando a algunos métodos y técnicas de Ingeniería de Requisitos de manera complementaria, hasta dar lugar a ser una pieza clave para otros [YM98].

Ventajas

Entre las ventajas que se pueden destacar de los métodos y técnicas basados en objetivos, se encuentran:

- Guía el **proceso de adquisición** de requisitos, permitiendo entender el porqué de los mismos, así como las posibles alternativas con las que se cuenta.

- Relaciona los requisitos con el **contexto organizacional y de negocio**.
- Permite **analizar los requisitos**, estudiando su descomposición sucesiva y su refinamiento.
- Muestra las **relaciones conflictivas** entre requisitos, permitiendo analizar las posibles soluciones a estos problemas en una etapa temprana del desarrollo.
- Sirven como **guía del diseño**.

En las siguientes subsecciones estudiaremos con mayor detalle algunos métodos y técnicas concretos que emplean los objetivos para realizar el proceso de Ingeniería de Requisitos: GBRAM, KAOS e i*.

3.2.1 GBRAM: *Goal-based Requirements Analysis Method*

GBRAM (*Goal-based Requirements Analysis Method*) es un método de análisis de requisitos basado en objetivos propuesto por Antón *et al.* [AP98]. Parte del supuesto de que los objetivos no se han documentado previamente, sino que deben obtenerse y, posteriormente, analizarse y refinarse. Para la obtención de los requisitos, los autores proponen el uso de técnicas conocidas y generalizadas, como la realización de un desarrollo incremental, gestión de riesgos, diseño participativo, prototipado rápido o paseos por escenarios (*scenario-walkthrough*).

Conceptos de GBRAM

El concepto fundamental de GBRAM es el de **objetivo**; sin embargo, este concepto se ve respaldado por otros dos que son necesarios y recurrentes en la ingeniería de requisitos. Dichos conceptos son ***stakeholder*** (aquella persona que tiene un interés en la consecución de un determinado objetivo) y **agente** (ente que es responsable de la consecución de un objetivo).

El método GBRAM consta fundamentalmente de dos actividades: **análisis de objetivos** y **refinamiento de objetivos** (Figura 3.2).

Análisis de objetivos

En el análisis de objetivos se deben realizar tres tareas básicas. La primera de ellas consiste en realizar una **exploración de actividades**. En esta tarea se debe examinar la información que se dispone sobre el sistema. Posteriormente, se debe realizar una **identificación de actividades**, dando lugar a una especificación inicial de los requisitos. Finalmente, debe realizarse una **organización de actividades** para clasificar los objetivos y detectar las dependencias entre ellos.

Refinamiento de objetivos

En la etapa de refinamiento de objetivos se deben realizar a su

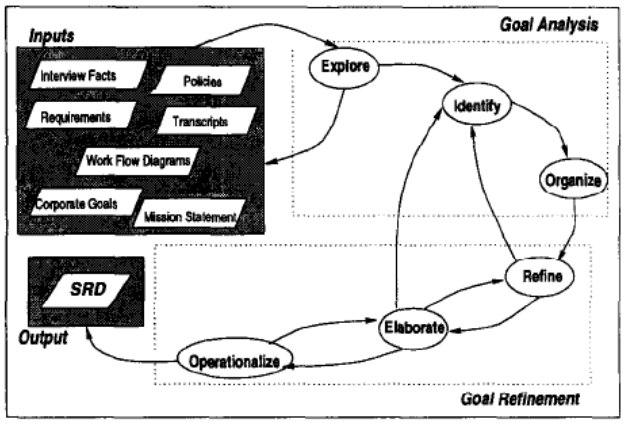

Figura 3.2: Actividades llevadas a cabo en el método GBRAM. Tomado de [AP98].

vez tres tareas complementarias. En primer lugar deben **refinarse los objetivos** para reducir su tamaño y eliminar las posibles redundancias existentes. A continuación, los autores plantean la **elaboración de los objetivos** para analizar los posibles obstáculos para su consecución y la construcción de escenarios para descubrir objetivos y requisitos ocultos. Por último, se deben **operacionalizar los objetivos**, es decir, tomar decisiones que traten de satisfacerlos, para obtener una especificación final de los requisitos del sistema.

3.2.2 KAOS: *Goal-driven Requirements Engineering*

La metodología **KAOS**, propuesta por A. van Lamsweerde [DDMvL97] [DvLF93], es otra metodología de Ingeniería de Requisitos basada en objetivos, la cual ha servido de inspiración a numerosos trabajos en la bibliografía. La metodología contiene un lenguaje de especificación y un método de elaboración de requisitos que permite analizar numerosos sistemas.

Conceptos de KAOS

KAOS propone una ontología con una mayor riqueza conceptual que GBRAM. A los conceptos de **objetivo** y **agente**, añade los conceptos **objeto** y **acción**. Además, establece un conjunto de relaciones entre dichos conceptos, entre los que destacan las siguientes relaciones:**refinamiento** (relación entre un objetivo de alto nivel y un conjunto de objetivos más detallados), **conflicto** (entre dos objetivos o requisitos que no pueden satisfacerse simultáneamente), **operacionalización** (entre un objetivo y un conjunto de acciones) y **asignación de responsabilidad** (entre un objetivo y el agente encargado de su satisfacción).

Método de elaboración

El proceso propuesto en KAOS consiste en la realización iterativa de los siguientes pasos:

1. Identificar y refinar objetivos hasta que se puedan asignar a agentes. Los objetivos se organizan empleando un grafo AND/OR que muestra su refinamiento y sus relaciones.
2. Identificar objetos y acciones de manera progresiva.
3. Derivar los requisitos en objetos y acciones para que cumplan las restricciones.
4. Asignar restricciones, objetos y acciones a los agentes que van a satisfacer los requisitos.

Perfil UML

La notación propuesta originalmente por los autores de KAOS no es estándar (Figura 3.3). Aunque existe una herramienta para poder modelar requisitos con KAOS, su uso está bastante limitado. Por tanto, en [HF04] se presenta un perfil UML que permite representar los conceptos existentes en KAOS mediante un conjunto de estereotipos y valores etiquetados (Figura 3.4). Ello permite el uso de una notación estandarizada y para la cual existen numerosas herramientas.

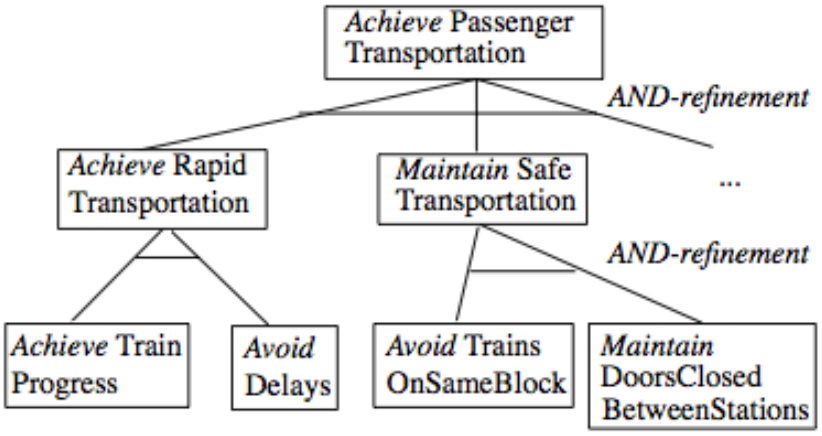

Figura 3.3: Notación no estándar de KAOS. Tomado de [DDMvL97].

3.2.3 i*: *Early-phase Requirements Engineering*

Yu *et al.* [Yu97] proponen un método de Ingeniería de Requisitos basado en objetivos llamado **i***. Este método plantea la elaboración de dos modelos: el ***Strategic Dependency Model*** y el ***Strategic Rationale Model***.

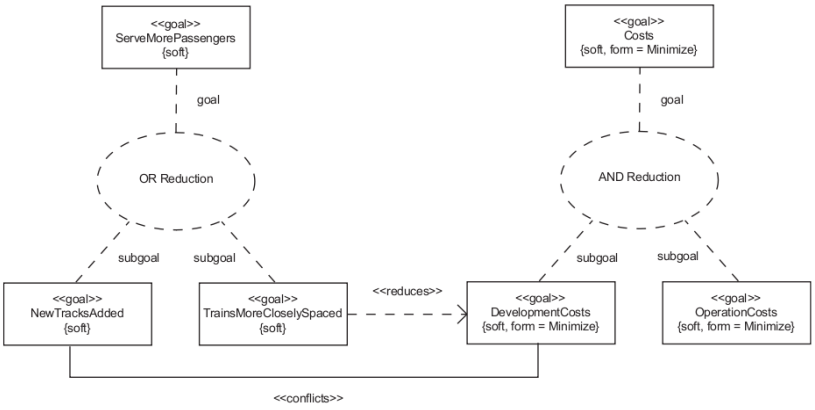

Figura 3.4: Ejemplo de utilización del perfil UML de KAOS. Tomado de [HF04].

*Conceptos de I**

i* presenta como concepto fundamental el de **actor intencional**. Los actores del sistema tienen objetivos, creencias y habilidades, y dependen de otros por los objetivos a lograr y las tareas que deben realizar. Los actores se comportan de manera estratégica, ya que buscan reconfiguraciones del sistema para que éste sirva a sus necesidades.

Strategic Dependency Model

El *Strategic Dependency Model* es un modelo que contiene las relaciones de dependencia entre los actores en un contexto organizacional. Ayuda a entender las configuraciones organizacionales tal como existen, o como se proponen. En el modelo se distinguen fundamentalmente cuatro conceptos, **objetivo**, **recurso**, **tarea** y ***softgoal***, que se relacionan mediante relaciones de dependencia y se organizan en torno a los actores del sistema. Presenta una mayor riqueza semántica que los anteriores, ya que permite representar propiedades de calidad del sistema mediante los *softgoals* (objetivos para los que no existe un criterio claro para definir su satisfación).

Strategic Rationale Model

Por otra parte, el *Strategic Rationale Model* describe los intereses de los *stakeholders* y cómo éstos se abordan desde las diferentes configuraciones del sistema. Para ello, partiendo del *Strategic Dependency Model*, deben desarrollarse con mayor detalle cada uno de los actores del sistema. Asimismo, deben añadirse relaciones positivas o negativas de la contribución de cada elemento a la satisfacción de los softgoals, que permiten evaluar y obtener diferentes alternativas del sistema. Por último, debe obtenerse una representación jerárquica de las tareas que deben realizarse.

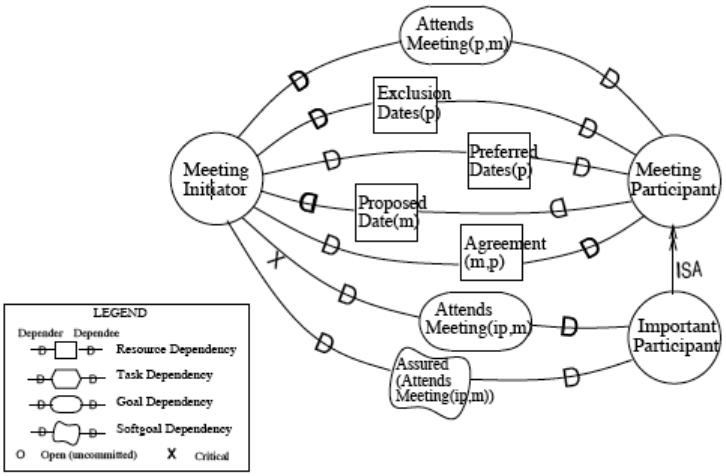

Figura 3.5: Ejemplo de utilización del *Strategic Dependency Model* en i*. Tomado de [Yu97].

3.3 Métodos basados en Relajación de objetivos

Los métodos vistos en la sección anterior especifican los requisitos del sistema en términos de sucesivos refinamientos de objetivos, jerarquizados según su especificidad. Sin embargo, estos métodos solamente contemplan la satisfacción (o no) de los objetivos que se especifican; es decir, no se refleja el hecho de que pueda obtenerse una satisfacción parcial de algún objetivo. Más aún, existen algunos objetivos cuya satisfacción no puede determinarse de manera objetiva, y otros que admiten diferentes graduaciones en su satisfacción.

Para tratar de incorporar estos hechos a los modelos de Ingeniería de Requisitos basada en objetivos, se plantean dos soluciones que, sobre la base de la metodología KAOS, añaden lógica difusa para reflejar la satisfacción parcial de los objetivos: FLAGS y RELAX.

3.3.1 FLAGS: *Fuzzy Live Adaptive Goals for Self-adaptive Systems*

El método **FLAGS** (*Fuzzy Live Adaptive Goals for Self-Adaptive Systems*), propuesto por Baresi *et al.* [BPS10], es una extensión de la metodología KAOS para incorporar la posibilidad de satisfacciones parciales de objetivos. Emplea la lógica difusa para «suavizar» los objetivos y permitir pequeñas violaciones en su satisfacción. Ello es particularmente útil en sistemas adaptativos,

como son los sistemas ubicuos, cuando se encuentran en proceso de transición entre un estado y otro.

Método FLAGS

El proceso propuesto por los autores consta de las siguientes actividades. En primer lugar, los autores proponen la creación de un **modelo de objetivos** siguiendo la notación tradicional de KAOS. En dicho modelo, deben identificarse las **relaciones conflictivas** entre los diferentes objetivos, y **asignar prioridades** a la satisfacción de los objetivos en conflicto. A continuación, se deben formalizar los objetivos empleando **Lógica Temporal Lineal**.

Una vez obtenido este modelo (similar al proceso tradicional de KAOS), se deben **suavizar los objetivos** en conflicto, permitiendo pequeñas violaciones de su satisfacción por períodos de tiempo reducidos o permitiendo su satisfacción parcial, entre otros. Para realizar este proceso, los autores proponen el uso de lógica difusa, y distinguen entre dos tipos de objetivos: ***crisp***, aquéllos cuya satisfacción es un valor *booleano*, y ***fuzzy***, los que toman un valor en el intervalo [0,1].

Por último, se debe especificar la **adaptación en ejecución**. Para ello, Baresi *et al.* proponen el concepto de **objetivo adaptativo**. Un objetivo adaptativo consta de un disparador, que determina cuando se ha de llevar a cabo dicho objetivo, una condición para su activación y un objetivo a lograr, que puede ser forzar la satisfacción de un objetivo hoja (aquél que no tiene descendientes) sin alterar su definición, forzar una versión relajada de un objetivo, o prevenir una situación de baja satisfacción en un objetivo. Adicionalmente, se especifican un conjunto de acciones a llevar a cabo, como pueden ser añadir o quitar objetivos del modelo, modificar un objetivo hoja, añadir o quitar operaciones, modificar las pre y post–condiciones de un objetivo, o añadir o quitar entidades, eventos y agentes.

Así, en la Figura 3.6 pueden verse ejemplos de los objetivos adaptativos. El objetivo $G1{,}2$ (izquierda), *mantener un bajo consumo de energía*, presenta una posible adaptación, $AG1{,}2$, que consiste en quitar la operación *seleccionar programa* y añadir la operación *seleccionar ecoprograma*. Para el caso del objetivo $G1{,}3$ (centro), *mantenerse sin ropa sucia*, existen dos posibilidades: *relajar la satisfacción de* $G1.2$, es decir, permitir una satisfacción parcial del objetivo, o bien añadir un nuevo objetivo. Por último, en el objetivo $G1{,}4$, *lograr realizar el lavado*, la adaptación consiste en añadir un objetivo, y añadir la operación *encender la lavadora*.

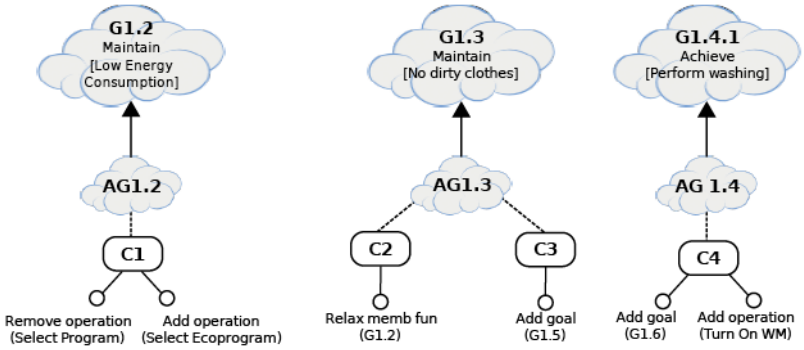

Figura 3.6: Ejemplo de objetivos adaptativos en un sistema de gestión de lavanderías en FLAGS. Tomado de [BPS10].

3.3.2 RELAX

El método propuesto por Cheng *et al.* pretende extender KAOS para obtener un método de Ingeniería de Requisitos para sistemas adaptativos que incorpore la incerteza del ambiente de aplicación [CSBW09]. Para ello, los autores se sirven de un lenguaje, RELAX [WSB+09], que permite relajar temporalmente los objetivos o requisitos para realizar una adaptación, si es necesario, mediante el empleo de lógica difusa (Figura 3.7).

Ámbito de incerteza

En primer lugar, se debe determinar el **objetivo general** del sistema. Adicionalmente, se elabora un **modelo conceptual** con los actores del sistema (tanto humanos como agentes software) que sirven de apoyo para identificar las condiciones del entorno y la incerteza que debe manejar el sistema. Se incorporan al sistema, por tanto, las **fuentes de incerteza** y **los monitorizadores del entorno**.

Modelado del Sistema Objetivo

A continuación se deben **refinar los objetivos generales** a objetivos de alto nivel, y finalmente a objetivos de bajo nivel, lo cual servirá para determinar los servicios clave que debe proporcionar el sistema. Para este propósito, los autores proponen el uso de la notación de la metodología KAOS.

Identificar incerteza

Tras realizar el modelado de los objetivos, debe realizarse una **búsqueda de las fuentes de incerteza** potenciales[1], siguiendo una estrategia *bottom-up*. Dichas fuentes se reflejan como obstáculos en un grafo AND/OR similar al grafo de objetivos, y ligado a éste; es decir, existen relaciones entre los obstáculos existentes y los objetivos a los que afectan.

Mitigar incerteza

Por último, debe **mitigarse la incerteza**. Para ello, debe anali-

1 Cuando se habla de fuentes de incerteza, los autores se refieren a obstáculos para la satisfacción de los objetivos

zarse el grafo AND/OR de los obstáculos potenciales y determinar si es necesario mitigar dicha incerteza. En caso afirmativo, existen diferentes estrategias a seguir, ordenadas de menor a mayor coste:

- Definir nuevo comportamiento como subobjetivos que manejen la condición excepcional.
- Si se admite la satisfacción parcial, relajar el objetivo empleando el lenguaje RELAX.
- Si no se puede relajar el objetivo, se da por fallado. En este caso, debe crearse un nuevo objetivo encargado de corregir el fallo causado.

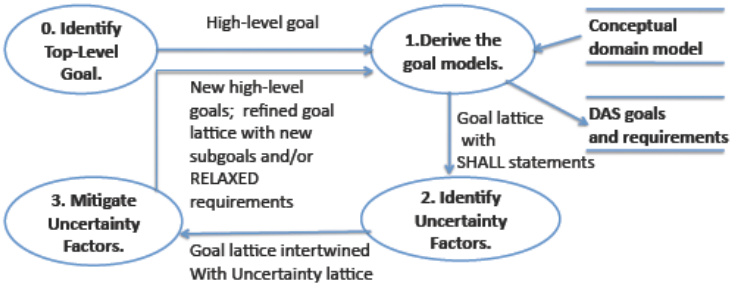

Figura 3.7: Etapas propuestas por el método RELAX. Tomado de [CSBW09].

Los autores proponen realizar una iteración sucesiva entre cada una de las etapas hasta conseguir tener un modelo de objetivos y requisitos completo.

3.4 *Scenario-based Requirements Engineering*

Los métodos de **Ingeniería de Requisitos basada en escenarios** [SM98] plantean el uso de descripciones de las secuencias de actividades que ocurren en el sistema para, a partir de ellas, determinar los requisitos del futuro sistema que se pretende desarrollar. Los escenarios presentan una visión futura del sistema mediante secuencias de comportamientos y descripciones contextuales. Son particularmente útiles, ya que los *stakeholders* tienen a expresar sus necesidades como actividades a realizar, en lugar de objetivos a alcanzar.

Ventajas e inconvenientes

Los métodos basados en escenarios presentan numerosas ventajas; a saber, son métodos centrados en la realidad, que sirven

como una base de razonamiento sobre el sistema futuro, permiten validar el sistema desarrollado haciendo un seguimiento del comportamiento descrito, y sirven como guía del proceso de modelado. Sin embargo, también presentan algunos inconvenientes: en ocasiones, centrarse en escenarios concretos de aplicación del sistema puede causar pérdida de generalidad de la solución. Los *stakeholders*, por lo general, ofrecen visiones personales del sistema, no globales, las cuales pueden ser conflictivas. Además, pueden no guiar hacia el modelo correcto, sino hacia la satisfacción de los intereses de un subconjunto de *stakeholders*. Por último, existe una tendencia a buscar los escenarios positivos, mientras que se olvidan las situaciones de excepción. Para paliar este hecho, no sólo deben elaborarse casos de uso del sistema, sino también casos de mal-uso, o cursos alternativos en los casos de uso que prevean esas situaciones no deseadas.

En esta sección se presentan dos métodos de Ingeniería de Requisitos basada en escenarios, SCRAM y ScenIC.

3.4.1 SCRAM: *Scenario-based Requirements Analysis Method*

Motivación

La motivación de **SCRAM** (*Scenario-based Requirements Analysis Method*) [Suto3] es doble. Por una parte, se pretende que los usuarios del sistema futuro se impliquen en el proceso de adquisición y refinamiento de los requisitos. Por otra parte, el uso de escenarios sirve para contextualizar la discusión sobre el sistema. Su metamodelo se muestra en la Figura 3.8.

Método

El método SCRAM propone la realización de las siguientes actividades:

- **Captura inicial de los requisitos y familiarización con el dominio**: a través de un proceso de entrevistas, se debe recopilar la información primaria sobre el sistema a desarrollar, modelada en texto plano o formularios.
- ***Storyboarding* y primeras versiones del diseño**: en esta fase se construyen las primeras versiones del sistema futuro.
- **Exploración de requisitos**: se construye un conjunto de prototipos básicos del sistema, que sirven para realizar una crítica del diseño y una validación inicial de los requisitos obtenidos.
- **Prototipado y validación de requisitos**: se construyen prototipos completamente funcionales y se refinan los requisitos hasta que se satisfacen teniendo en cuenta a todos los *stakeholders*.

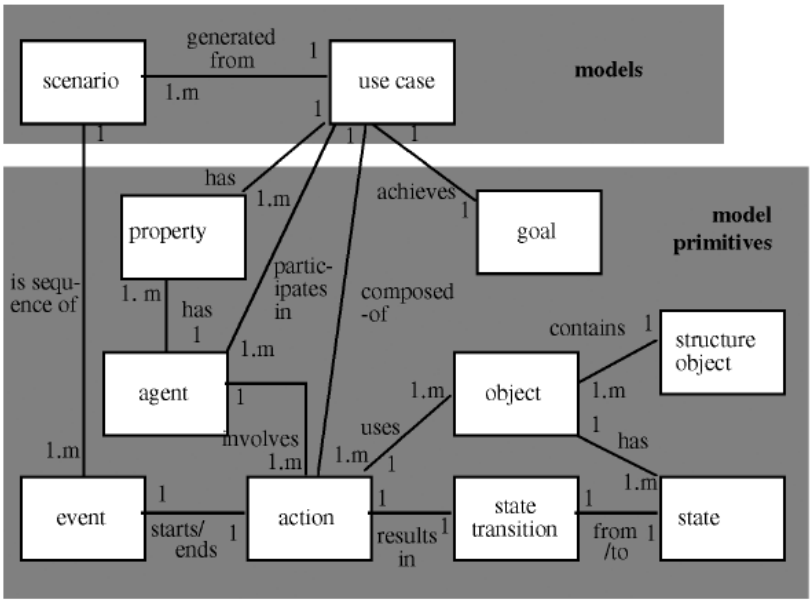

Figura 3.8: Metamodelo de escenarios en SCRAM. Tomado de [Suto3].

3.4.2 ScenIC: *Scenario-based Requirements Engineering*

ScenIC es un método de Ingeniería de Requisitos basada en escenarios, propuesto por Potts [Pot99], que se sustenta en el método ***Inquiry Cycle*** [PTA94]. Este método establece una analogía entre el funcionamiento de la memoria humana y las etapas que se llevan a cabo en la Ingeniería de Requisitos.

Semantic Memory Schema

El método propone la elaboración de tres modelos o esquemas, los cuales son una metáfora de una de las partes de la memoria humana. El primero de ellos, el ***Semantic Memory Schema***, trata de representar la información del sistema (con visiones tanto del *hardware/software*, como del entorno). Para ello, sigue un enfoque basado en metas, en el que el concepto fundamental es el de ***goal***. Un *goal* puede ser un **objetivo** (una propiedad del sistema que se debe lograr o mantener) o una **tarea** (una serie de acciones que se realizan para llegar a un estado del sistema). Las tareas son realizadas por **actores**. Por otra parte, el modelo representa también los obstáculos que entorpecen la consecución de *goals* y los relaciona con las tareas que ayudan a mitigar su efecto.

Episodic Memory Schema

El segundo modelo, ***Episodic Memory Schema***, recoge un conjunto de **escenarios** relacionados con la descripción semántica del sistema para el entendimiento de los usuarios. Los escenarios contienen **episodios**, y éstos a su vez contienen secuencias de **acciones**. Los episodios se relacionan con los obstáculos y *goals* del modelo anterior, y las acciones con instancias concretas de

los actores y tareas. De esta manera se potencia la trazabilidad entre los diferentes modelos.

Working Memory Schema

El tercer y último modelo, ***Working Memory Schema***, se utiliza para anotar los asuntos pendientes o no tratados aún en el sistema, y sirve como apoyo para el posterior refinamiento de los requisitos. El concepto fundamental manejado en este modelo es el de **recordatorio** (*reminder*). Un recordatorio puede ser una decisión que se debe tomar o un asunto que se debe tratar sobre algún aspecto del sistema, y se relaciona con elementos del *Semantic* o *Episodic Memory Schema*.

Estrategia del método

Los autores proponen la siguiente estrategia a modo de guías de diseño para utilizar el método ScenIC:

- Identificar el **ámbito del sistema** y los **actores** involucrados en el mismo.
- Identificar los ***goals*** y refinarlos. Se propone un conjunto reducido de verbos para expresarlos (p.ej. mantener, maximizar, evitar...).
- Establecer las **dependencias existentes entre tareas** (temporales, condicionales...).
- Realizar la **asignación de tareas a actores**.
- Identificar la presencia de **obstáculos** siguiendo estrategias *top-down* y *bottom-up*.
- **Elaborar** los objetivos y tareas.
- **Identificar y elaborar los escenarios**. Debe existir uno o varios casos de uso normales (con alternativas), y casos de excepción, hasta que se cubran todos los objetivos del sistema.
- **Componer los escenarios** más simples para dar lugar a otros de más alto nivel.

3.5 Métodos para la gestión de Requisitos No Funcionales

Los **Requisitos No Funcionales** (NFR), a menudo llamados atributos de calidad o, coloquialmente «*-ilities*», son cualidades globales de un sistema software, tales como la flexibilidad, la usabilidad, la seguridad, etc. Habitualmente, dichos requisitos se especifican de manera informal, y suelen ser controvertidos, dado

que involucran intereses conflictivos entre los *stakeholders*, por lo que son difíciles de tener en cuenta durante el análisis y el diseño, y complicados de validar [MCY99]. Una mala especificación de los requisitos no funcionales da lugar a un software de baja calidad, y al fracaso en el desarrollo del producto.

En esta sección estudiaremos un método que puede encontrarse en la bibliografía para tratar los requisitos no funcionales de manera adecuada en el proceso de desarrollo de software en general, y de la Ingeniería de Requisitos en particular.

3.5.1 *NFR Framework*

Softgoal

Con el objetivo de tratar de manera sistemática los requisitos no funcionales en el proceso de diseño y desarrollo de software, Chung *et al.* [CNYM00] [MCY99] proponen el ***NFR Framework***. Este *framework* gira en torno al concepto de ***softgoal***, que se define como un objetivo del sistema futuro para el cual no existe una manera de determinar claramente si está satisfecho o no, sino que se considera que se cumple cuando hay suficiente evidencia positiva y poca negativa para que se satisfaga. Este concepto es particularmente apropiado para representar las propiedades de calidad del software (requisitos no funcionales), ya que habitualmente no puede determinarse de manera clara su cumplimiento o no, sino que existe una graduación en su satisfacción.

Tipos de Softgoals

El *framework* distingue tres tipos de *softgoals*:

- ***NFR Softgoals*:** son representaciones de los requisitos no funcionales del sistema futuro, como precisión o seguridad. Se parte de *softgoals* de alto nivel, los cuales son descompuestos en otros más simples y concretos.
- ***Operationalizing Softgoals*:** una vez descompuestos los *NFR Softgoals*, es necesario buscar soluciones que contribuyan a su satisfacción. Dichas soluciones se denominan operacionalizaciones, y comprenden operaciones, procesos, representación de datos, estructuras...
- ***Claim Softgoals*:** sirven para proporcionar justificación a las descomposiciones y operacionalizaciones realizadas por el diseñador.

Softgoal Interdependency Graphs

Los *softgoals* son organizados en lo que se denomina **Grafos de Interdependecias entre *Softgoals*** (*Softgoal Interdependency Graphs*, SIG) (Figura 3.9). Dichos grafos muestran las relaciones entre los distintos tipos de *softgoals*. Existen diferentes relaciones de interdependencia entre *softgoals*: **descomposición**, que refinan un

softgoal en otros del mismo tipo; **operacionalización**, que muestran la contribución de una solución a la consecución de un *softgoal*; **argumentación**, que proporcionan justificación a las decisiones de diseño; y **priorización** que muestran qué *softgoals* deben recibir mayor consideración.

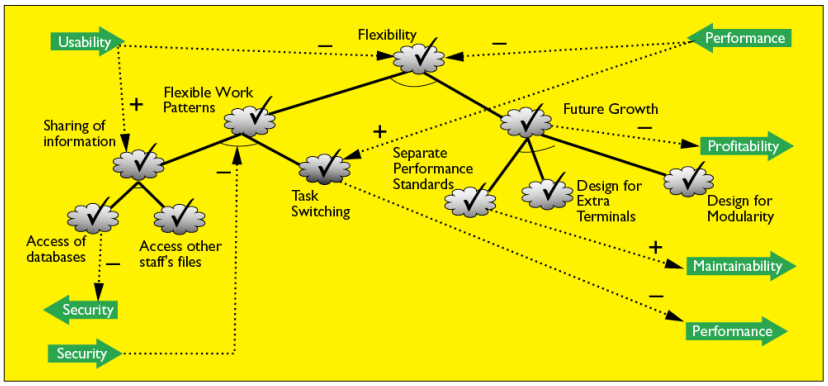

Figura 3.9: Ejemplo de un *Softgoal Interdependency Graph*. Tomado de [CNYMoo].

Contribuciones

Las contribuciones que pueden darse entre unos *softgoals* y otros pueden ser de diversos tipos:

- **AND/OR:** una relación tipo AND implica que el *softgoal* padre es satisfecho si y sólo si todos sus hijos se satisfacen; una relación de tipo OR implica que el *softgoal* padre se satisface si alguno de sus hijos lo hace.
- **Positiva:** una contribución positiva implica que si el *softgoal* hijo se satisface, el padre también podría satisfacerse. Existen dos relaciones positivas: *help* y *make*, siendo la segunda de mayor contribución que la primera.
- **Negativa:** una contribución negativa implica que si el *softgoal* hijo se satisface, el padre podría no satisfacerse. Existen dos relaciones negativas: *hurt* y *break*, siendo la segunda de mayor impacto que la primera.
- **Desconocida:** una contribución desconocida indica que existe alguna relación entre los *softgoals*, pero no se sabe en qué medida.

Procedimiento de Evaluación

Teniendo en cuenta estas contribuciones, se propone un **procedimiento de evaluación**, el cual, de manera semi–automática (con ayuda del diseñador cuando se produce una situación de

conflicto), permite determinar si la elección de operacionalizaciones realizada permite satisfacer los *softgoals*. Dichas operacionalizaciones son organizadas de manera que se compruebe que satisfacen los requisitos funcionales y no funcionales del sistema.

Catálogos

Por último, se propone la creación de diferentes catálogos que organicen los tipos de requisitos no funcionales, métodos de descomposición, operacionalización y argumentación, y las correlaciones entre los diferentes requisitos no funcionales, para que puedan ser reutilizados y sirvan de apoyo al proceso de Ingeniería de Requisitos.

3.6 Otras propuestas de interés para la especificación y tratamiento de requisitos

3.6.1 PORE: *Procurement-Oriented Requirements Engineering*

PORE (*Procurement-Oriented Requirements Engineering*), es un método de Ingeniería de Requisitos orientado a la adquisición de componentes software, creado por Ncube *et al.* [NM99]. Los autores proponen el método en el marco del **diseño de software basado en componentes**. Consiste en un conjunto de procesos que se llevan a cabo de manera iterativa, y que están especificados a tres niveles:

Niveles PORE

Universal: procesos que sirven como guía general a los actores del proceso.

***Wordly*:** procesos relevantes a la adquisición iterativa de requisitos, y a la evaluación y selección de productos software que satisfagan los requisitos.

Atómico: métodos, procedimientos y técnicas que habilitan los procesos especificados en el nivel *Wordly*.

Actividades en PORE

La adquisición de requisitos en PORE se realiza mediante la evaluación iterativa de diferentes componentes software, de tal manera que, tras sucesivas evaluaciones de productos, el conjunto de requisitos adquiridos es cada vez mayor, y el conjunto de componentes a estudiar se va reduciendo. Las actividades que deben realizarse en PORE son:

- **Gestión del Sistema de Contratación:** planear y gestionar el proceso de contratación para satisfacer las necesidades del contratador a tiempo y a un coste razonable.

- **Adquisición de Requisitos:** adquirir y validar los requisitos del cliente.
- **Selección de Proveedores:** establecer los criterios de selección, evaluar los proveedores y priorizarlos.
- **Selección de Paquetes Software:** identificar los productos candidatos, establecer los criterios de selección, evaluar los componentes y priorizarlos.
- **Producción del Contrato:** negociar el contrato con el proveedor.
- **Aceptación del Paquete:** contrastar el paquete adquirido con los requisitos finales del cliente.

Modelos PORE

Para determinar la idoneidad de un determinado componente para uno o varios requisitos del cliente, los autores proponen la creación de varios modelos. El primero de ellos es el **modelo de productos**. En él se refleja el comportamiento observable del producto (empleando casos de uso), los objetivos del producto (empleando algún modelo de Ingeniería de Requisitos basada en objetivos) y la arquitectura del producto. Por otra parte, el **modelo de requisitos** recoge tanto los requisitos funcionales y no funcionales del cliente, como la información que se dispone de los proveedores. Por último, se establece un **modelo de conformidad** que liga cada producto evaluado con los requisitos según la idoneidad de los primeros para satisfacer estos últimos.

3.6.2 SORE: *Service-Oriented Requirements Engineering*

En el marco de la Arquitectura Orientada a Servicios (*Service Oriented Architecture*, SOA [OAS06]) se encuadra **SORE** (*Service Oriented Requirements Engineering*) [TJWW07]. El método asume el empleo de algún *framework* de desarrollo para SOA, lo cual implica unas determinadas elecciones y decisiones de diseño, incluso antes de realizar el proceso de Ingeniería de Requisitos, diferenciándose así del enfoque tradicional en el que el diseño sigue a la fase de análisis de requisitos.

Orientación a la Reusabilidad

SORE es un *framework* **orientado a la reusabilidad** y **acumulativo**, en el que se pretende maximizar la reutilización de elementos entre iteraciones de un mismo proyecto, entre proyectos, y hasta del propio proceso de desarrollo. Para los autores, no sólo los servicios son reusables, sino también los *workflows*, las plantillas de aplicación, los esquemas de datos, las *policies*, los *script* de test y las interfaces de usuario. Este hecho pone de

manifiesto la gran proliferación de artefactos reusables que se pueden encontrar, dando lugar a la necesidad de proporcionar mecanismos de clasificación para estos recursos reusables.

Dependencia del Dominio

A diferencia de los métodos tradicionales de Ingeniería de Requisitos, SORE enfatiza la utilización de elementos **dependientes del dominio** concreto de la aplicación, como pueden ser ontologías que representen el dominio.

Basado en Evaluación

Dado que los servicios pueden ser descubiertos y usados incluso en tiempo de ejecución, los autores establecen que SORE debe ser **basado en evaluación**; es decir, deben proporcionarse mecanismos para realizar tests a los servicios en tres momentos fundamentales: (i) en la preparación, para pre-evaluar las características del servicio; (ii) en la composición, para comprobar completitud y consistencia; y (iii) en la ejecución, para confirmar el cumplimiento de políticas (*policies*) y realizar monitorización dinámica.

Análisis de políticas

Por último, de manera paralela al análisis y modelado de requisitos, debe realizarse un **modelado y análisis de políticas (*policies*)** para determinar aquéllas restricciones que se imponen sobre el sistema.

3.6.3 AORE: *Aspect-Oriented Requirements Engineering*

En el marco de la **Programación Orientada a Aspectos** [KLM+97], Rashid *et al.* [RSMA02] proponen el método de Ingeniería de Requisitos **AORE** (*Aspect-Oriented Requirements Engineering*). La motivación de este método es doble: por una parte, se pretende realizar una separación de *crosscutting concerns* para identificar y manejar conflictos; por otra, se debe facilitar la identificación de la relación entre requisitos y artefactos que surjan en etapas posteriores del desarrollo.

Método AORE

El método propone la realización de diferentes actividades para determinar los requisitos del sistema. En primer lugar, se deben identificar, por una parte, los **intereses (*concerns*)**[2] relativos al sistema, y por otra, los **puntos de vista (*viewpoints*)** del sistema, los **requisitos** que se encuentran relacionados a ellos, y vincularlos a los *concerns* (Figura 3.10).

Teniendo en cuenta lo obtenido anteriormente, se debe realizar una **especificación de *concerns***, y a partir de dicha especificación, identificar los **aspectos candidatos**. Se dice que un aspecto es candidato si afecta de manera transversal a varios requisitos. Posteriormente, mediante un proceso de refinamiento, se pasa

2 Un concern es «*cualquier cosa sobre el software sobre la que se desea poder pensar como una entidad relativamente bien definida.*» (Gregor Kiczales) [KLM+97]

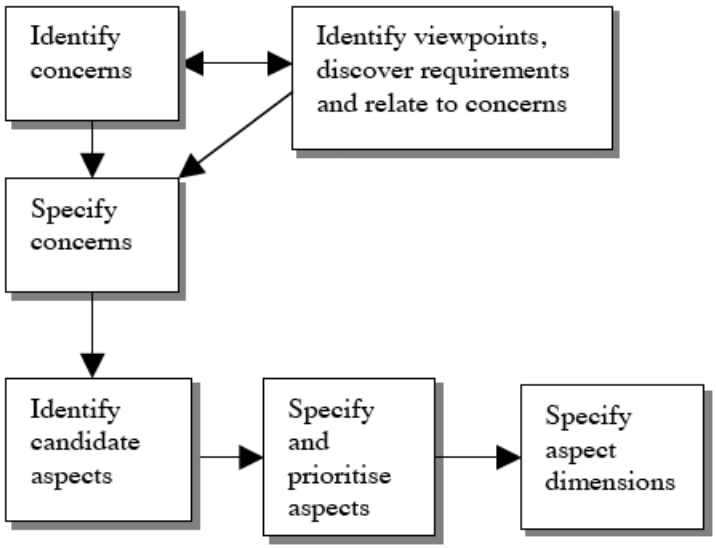

Figura 3.10: Ciclo de actividades en AORE. Tomado de [KLM+97].

a **especificar los aspectos candidatos y a priorizarlos**. En esta etapa pueden aparecer conflictos, que deberán ser resueltos.

Por último, para cada uno de los aspectos deben especificarse las dimensiones a las que afectan los aspectos. De acuerdo con los autores, al menos debe especificarse el **mapping** del aspecto (función del software, decisión de diseño, decisión de implementación...) y su **influencia** sobre las diferentes etapas del desarrollo.

3.6.4 PC-RE: *Personal and Contextual Requirements Engineering*

Necesidad de especificación personal de requisitos

Los métodos anteriores se centran en la adquisición y modelado de los requisitos de manera general para todos los usuarios de un sistema. Sin embargo, las anteriores propuestas no exploran la especificación de requisitos para individuos concretos. Según Sutcliffe *et al.* [SFS05], la Interacción Persona-Ordenador solo se ha centrado en las interfaces gráficas, dejando a un lado otros aspectos. Por otra parte, los requisitos de las aplicaciones son cambiantes respecto a diferentes individuos y entornos.

Framework PC-RE

Para tratar de reflejar estas necesidades en un método de Ingeniería de Requisitos, Sutcliffe *et al.* proponen **PC-RE** (*Personal and Contextual Requirements Engineering*), un *framework* que trata de relacionar los requisitos de una aplicación a necesidades individuales. El *framework* intenta estudiar, durante el proceso

de Ingeniería de Requisitos, cómo las necesidades individuales cambian con el tiempo, y cómo evolucionan los requisitos cuando la gente aprende a manejar el sistema y aumentan sus ambiciones. El *framework* se estructura en tres capas, cada una con dos dimensiones, espacial y temporal, que indican cómo cambian los requisitos ante cambios en dichas dimensiones (Figura 3.11).

Requisitos Generales de los Stakeholders

La primera de las capas se centra en los **Requisitos Generales de los *Stakeholders***. Esta capa recoge los requisitos más generales de la aplicación, y cómo varían en el tiempo (cambio de los procesos de negocio) y en el espacio (cambios culturales, internacionalización). Los requisitos a este nivel tienen implicaciones arquitectónicas, ya que se debe realizar un diseño orientado a la adaptación, incluir monitores del contexto de la aplicación, realizar interfaces personalizables o adaptativas, y, en general, contar con una arquitectura adaptable y flexible, capaz de incorporar los cambios que puedan ocurrir.

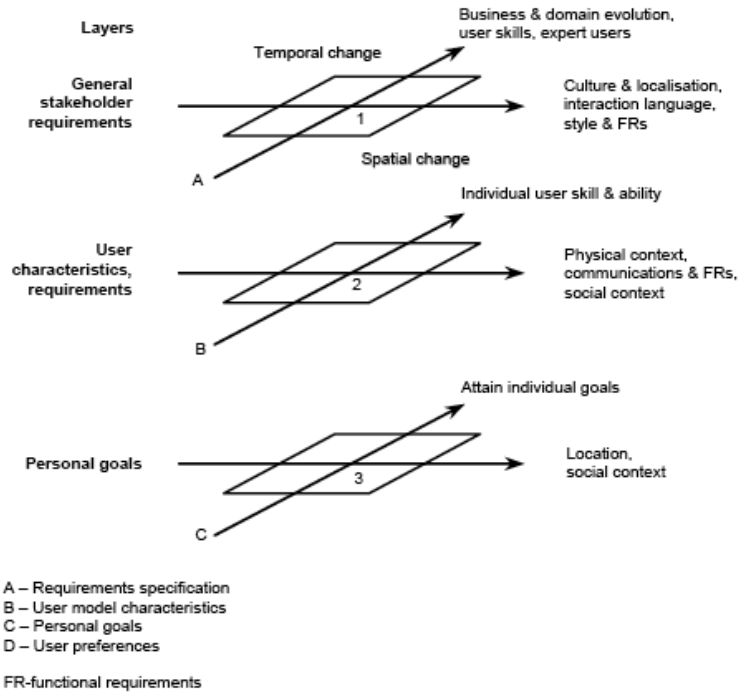

Figura 3.11: Capas del *framework* PC-RE junto con sus dimensiones de cambio. Tomado de [SFS05].

Requisitos y características de los usuarios

La segunda capa presenta los **requisitos y características de los usuarios**. En este nivel se recogen las necesidades personales de los usuarios, de manera parcialmente independiente del dominio de la aplicación. Sus habilidades son recopiladas en un perfil de usuario, que se emplea para elegir y adaptar la modali-

dad de comunicación. En este caso, la dimensión espacial se ve afectada por cambios en el contexto físico y social, mientras que la temporal es influenciada por los cambios en las habilidades del usuario a lo largo del tiempo.

Objetivos personales

Por último, la última capa recoge los **objetivos personales**. Este análisis es clave en aquellos sistemas en los que la personalización individual es clave para el sistema, por lo que los requisitos deben analizarse desde un punto de vista personal. A este nivel, el cambio en la dimensión temporal depende de la estabilidad de los deseos del usuario, mientras que el cambio espacial depende de cómo su ubicación afecta a sus objetivos.

Análisis Coste-Beneficio

La consideración de requisitos a nivel personal tiene como resultado la aparición de una amplia variedad de alternativas para satisfacer objetivos similares. Para discriminar y priorizar dichas alternativas para un posterior diseño e implementación, los autores proponen una **técnica de análisis coste-beneficio**. La técnica consiste en asignar un valor entre 1 y 100 al beneficio obtenido al satisfacer un objetivo con una determinada alternativa. A continuación, se reparten 100 puntos entre las categorías de costes o penalizaciones que se vayan a estudiar, y se puntúa cada categoría. Las categorías típicamente estudiadas son:

- Coste de cada alternativa para conseguir el objetivo. Debe estudiarse el coste del aprendizaje que supone al usuario utilizar dicha alternativa, el coste operacional impuesto al usuario, y el coste que se impone a otros (usuarios o no) por la imposición de dicha solución.
- Penalizaciones si la alternativa seleccionada no cumple con el objetivo.
- Penalización sobre los requisitos no funcionales si son infringidos, o beneficios si se contribuye a su satisfacción.

Tras obtener las puntuaciones de beneficios y costes de cada alternativa, se computa su porcentaje de satisfacción como la diferencia entre beneficios y penalizaciones, y se ordenan de manera decreciente. De esta forma, podemos observar las situaciones de compromiso o conflicto (*trade-offs*) que existen entre las alternativas posibles, y elegir aquella que maximice el porcentaje de satisfacción.

3.6.5 RE-CAWAR: *Req. Engineering for Context-Aware Systems*

RE-CAWAR (*Requirements Engineering for Context-Adaptive Systems*) es un método de **Ingeniería de Requisitos orientado a**

sistemas capaces de adaptarse dependiendo del contexto de aplicación, propuesto por Sitou *et al.* [SS07].

Integrated model of usage context

El método plantea la elaboración de un modelo integrado de uso del sistema en relación con el contexto (***integrated model of usage context***). En dicho modelo se consideran diferentes dimensiones que pueden afectar al sistema. Por una parte, los participantes en el sistema pueden cambiar; dichos cambios son típicamente relativos a su localización, pero también se refieren a sus propiedades personales, mentales, fisiológicas, etc. Por otra parte, las actividades a realizar en el sistema pueden sufrir cambios debido a que las tareas y objetivos pueden verse influenciados por eventos ocurridos en el entorno. Por último, el propio entorno puede cambiar (redes de comunicaciones, dispositivos, factores físicos...). Sumado a todo esto, se debe considerar el cambio de cada una de estas dimensiones a lo largo del tiempo.

El modelo integrado de uso contiene diferentes submodelos que se centran en aspectos concretos del sistema:

- **Modelo de Usuario:** contiene información acerca de usuarios y grupos en el sistema.
- **Modelo de Tareas:** contiene las actividades del sistema y las interacciones entre ellas.
- **Modelo de Dominio:** representa los aspectos operacionales del entorno, es decir, los objetos accesibles y directamente manipulables por el usuario.
- **Modelo de Plataforma:** modela la infraestructura física y las relaciones existentes entre los dispositivos con los que se cuenta.
- **Modelo de Diálogo:** refleja las posibles interacciones entre el usuario y el sistema.
- **Modelo de Presentación:** contiene los elementos necesarios para la interacción entre el usuario y el sistema.

Núcleo de RE-CAWAR

El **núcleo de la metodología** se basa en técnicas de Ingeniería de Requisitos basada en escenarios y objetivos, enriquecidas con técnicas de Ingeniería de la Usabilidad y modelado de usuarios. La metodología cuenta con dos partes fundamentales (Figura 3.12):

- ***Stability check*:** el principal objetivo de esta fase es obtener los requisitos generales del sistema, tanto para su núcleo como para su interfaz de usuario. Para ello, se recurre

a la formulación de casos de uso basados en los modelos anteriores para determinar los requisitos. Además, se deben obtener aquellas necesidades que dependen del contexto. Los resultados de esta fase sirven como entrada para la siguiente.

- ***Identification check*:** en esta fase deben identificarse aquellas necesidades que pueden ser detectadas automáticamente, y convertirlas en requisitos de adaptación de la aplicación. Deben emplearse conjuntamente técnicas de Ingeniería de la Usabilidad, ya que la automatización de adaptaciones puede dar lugar a problemas de usabilidad. Asimismo, deben realizarse propotipos de la aplicación para validar los requisitos obtenidos.

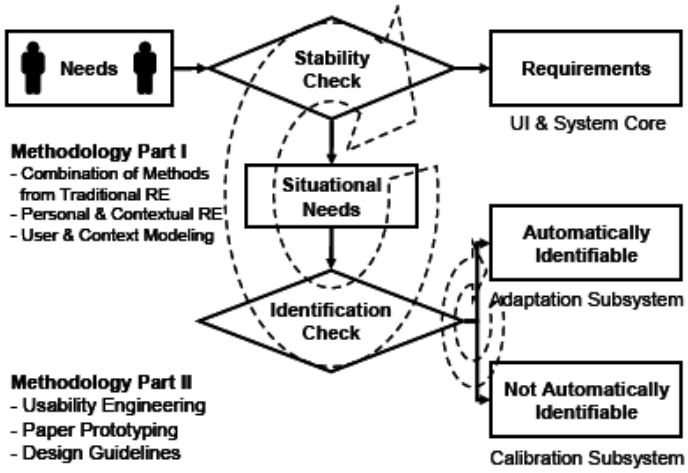

Figura 3.12: Iteración de las actividades en RE-CAWAR. Tomado de [SS07].

3.6.6 LoREM: *Levels of Requirements Engineering Method*

Basándose en lo expuesto en [BCZ05] sobre los niveles de Ingeniería de Requisitos en sistemas adaptativos dinámicamente, Goldsby *et al.* presentan **LoREM** (*Levels of Requirements Engineering Method*) [GSB+08], un método de **Ingeniería de Requisitos para sistemas dinámicos**, que se basa en el método basado en objetivos I*. En sistemas dinámicos existen tres aspectos fundamentales que deben tenerse en cuenta: (i) estudiar las condiciones a monitorizar para realizar la adaptación; (ii) determinar las adap-

taciones necesarias para obtener un nuevo comportamiento; y (iii) determinar el procedimiento de toma de decisiones.

Niveles LoREM

Los autores organizan las tareas a realizar dentro de LoREM en los cuatro niveles de Ingeniería de Requisitos, asignando las tareas a diferentes roles:

- **Nivel 1**: el responsable de realizar las tareas a este nivel se denomina **desarrollador de sistema**. En primer lugar, debe determinar los *goals* y *softgoals*, tanto del sistema dinámico como de las posibles adaptaciones. A continuación, para obtener los requisitos, debe identificar los posibles dominios; identificar las adaptaciones basándose en los *goals*, *softgoals* y dominios; y, para cada adaptación, crear un modelo de requisitos.
- **Nivel 2**: en esta etapa se continúa elaborando el modelo de requisitos. Para ello, deben identificarse las adaptaciones entre cada par de alternativas, así como la infraestructura que les da soporte. Se define un **escenario de adaptación** como una transición aceptable que cumple *goals* y *softgoals*. Por otra parte, se define un **modelo de adaptación** como los requisitos de los mecanismos de monitorización, toma de decisiones y de adaptación, y debe describir tanto los cambios funcionales como los no funcionales. El responsable de realizar estas tareas es el **desarrollador de escenarios de adaptación**.
- **Nivel 3:** llevado a cabo por el **desarrollador de infraestructura de adaptación**, debe llevar a cabo la selección de mecanismos de adaptación: identificar los mecanismos de monitorización, de toma de decisiones y de adaptación que satisfagan los *goals* y *softgoals*, dando lugar al **modelo de infraestructura de adaptación**.
- **Nivel 4:** corresponde a la investigación y desarrollo de mecanismos para infraestructuras de adaptación, y es llevado a cabo por la comunidad de investigación en sistemas adaptativos dinámicamente.

Procesos LoREM

En un proceso de desarrollo habitual, tan solo se siguen los tres primeros niveles (se parte de la hipótesis de que algunos métodos de adaptación han sido previamente estudiados). De esta manera, los autores proponen dos posibles procesos:

- **Proceso dirigido por Aplicación:** es un proceso *top-down* en el que el orden seguido en los niveles es $1 \rightarrow 2 \rightarrow 3$. Este enfoque es preferible cuando se cuenta con un conjunto

maduro de componentes de adaptación que cumplen los requisitos de la mayoría de las necesidades.

- **Proceso dirigido por la tecnología:** es un proceso *bottom-up* en el que el orden seguido en los niveles es $1 \rightarrow 3 \rightarrow 2$. Es aplicable cuando el soporte de los mecanismos de adaptación es limitado (para evitar modelar una adaptación que no es soportada), cuando el cliente fuerza una infraestructura de adaptación concreta, o cuando las adaptaciones son específicas de un dominio concreto.

3.6.7 EUC: *Executable Use Cases*

Motivación EUC

Con el objetivo de estrechar el hueco existente entre las ideas expresadas de manera informal en la especificación de requisitos y la formalización de la implementación, así como de estimular la comunicación entre usuarios y desarrolladores, Jrgensen *et al.* proponen los ***Executable Use Cases*** [JB04]. EUC es un método de extracción, elicitación y validación de requisitos basado en prototipado y descripción explícita del entorno.

Niveles EUC

EUC propone tres niveles, a modo de vistas, sobre el proceso de Ingeniería de Requisitos:

- **Nivel de Prosa:** en este nivel se realiza una especificación de requisitos en texto plano; es decir, se confecciona una descripción textual de los procesos de trabajo y el soporte que debe dar el sistema informático a dichos procesos.

- **Nivel Formal:** en este nivel se plantea el uso de lenguajes formales, entre los que los autores seleccionan el empleo de Redes de Petri. Deben modelarse los estados del sistema y las acciones que pueden realizarse en cada estado. Su función es reducir el salto existente entre la informalidad de los requisitos y la formalidad de una implementación. Los modelos obtenidos en este nivel sirven para verificar la corrección de los requisitos extraídos.

- **Nivel de Animación:** este nivel es una animación gráfica sobre el nivel formal. Se emplean conceptos gráficos que sean familiares a los usuarios, los cuales son animados disparando las transiciones correspondientes a las acciones en las Redes de Petri del nivel formal. Esto favorece la comunicación entre usuarios y desarrolladores, y contribuye a elicitar y validar los requisitos obtenidos.

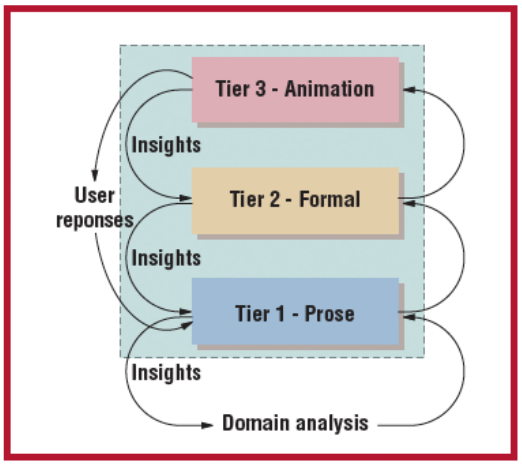

Figura 3.13: Niveles de especificación de EUC.

3.6.8 UWA: *Ubiquitous Web Applications*

UWA (*Ubiquitous Web Applications*) es una metodología para el desarrollo de aplicaciones Web ubicuas [Cono7a]. Emplea un modelo de requisitos basado en objetivos, inspirado en KAOS. Los conceptos fundamentales dentro de este método son:

Conceptos en UWA

- ***Goal*:** es un objetivo abstracto a largo plazo que debe realizarse mediante la cooperación de los agentes (software y humanos). Los *goals* deben **refinarse**, dando lugar a objetivos más concretos. Asimismo, deben identificarse los **conflictos** existentes entre los diferentes *goals*.
- **Requisito:** un requisito es un objetivo concreto, a corto plazo, que operacionalizan los *goals*.
- ***Stakeholder*:** un *stakeholder* es alguien o algo que tiene interés en el sistema. Mantienen una relación muchos a muchos con los *goals*, y en ocasiones tienen intereses conflictivos que deben ser solventados.
- **Valor:** es el *porqué* de la relación entre un *stakeholder* y un *goal*, y es extremadamente difícil de formalizar.
- **Dimensión:** una dimensión sirve para categorizar los requisitos y agruparlos.
- **Prioridad:** asignada a los requisitos, sirve para determinar la importancia de los requisitos y ayudar a resolver conflictos potenciales.

3.6.9 Patrones de Requisitos

La noción de patrón de diseño en Ingeniería del Software ha estado presente durante varios años desde la aparición del libro del *Gang of Four* con un conjunto de patrones para diseño orientado a objetos [**?**]. De manera similar, algunos autores han identificado patrones de diseño recurrentes en el ámbito de los Sistemas Ubicuos [LB03]. Otros trabajos existentes tratan de descubrir y definir patrones potenciales que se centran principalmente en una de las características esenciales de los Sistemas Ubicuos, la consciencia del contexto [RDFRR06] [RGL05].

El uso de patrones no está limitado al diseño; también se pueden definir y utilizar patrones de requisitos, aunque el tratamiento sistemático de los mismos (y en especial el tratamiento de los NFRs) mediante la aplicación de patrones aún no ha sido explorado ampliamente. Por ejemplo, Franch [Fra13] presenta un extenso catálogo de patrones de requisitos, agrupados en diferentes categorías, donde hay patrones específicos relativos a algunos NFRs, tales como rendimiento o usabilidad, junto con un metamodelo para patrones de requisitos [FPQ+10]. Supakkul *et al.* [SHC+10] propone un enfoque dirigido por patrones de NFRs, también para sistemas de propósito general. La búsqueda bibliográfica realizada no ha permitido encontrar ningún trabajo relacionado con patrones de requisitos en el ámbito de los Sistemas Ubicuos.

El uso de patrones tiene varias ventajas: acelera el desarrollo, incrementa la reutilización de artefactos software, proporciona soluciones generales que pueden aplicarse en diferentes ámbitos y establece una terminología común para referirse a problemas recurrentes, mejorando así la comunicación entre analistas y diseñadores. Sin embargo, deberían aplicarse cuidadosamente, dado que un mal uso de la aplicación de patrones puede conllevar un incremento innecesario de la complejidad del software [McC93].

3.7 Análisis

Tras realizar la presentación de diversos métodos y técnicas de Ingeniería de Requisitos, en esta sección se analizan sus ventajas e inconvenientes para su aplicabilidad en el proceso de diseño de Sistemas Ubicuos. Un resumen de este análisis se muestra en la Tabla 3.1.

3.7.1 Cobertura

La mayoría de técnicas examinadas se centran en el tratamiento y elaboración de requisitos una vez obtenidos, mientras que pocas de ellas plantean actividades para su adquisición. La técnica de *Executable Use Cases* (EUC) [JB04] es especialmente útil por estar enfocada en la elicitación de requisitos para Sistemas Ubicuos, aunque carece de técnicas que permitan elaborarlos para llegar a un diseño de software.

Otras técnicas de extracción de requisitos específicamente dedicadas para Sistemas Ubicuos se presentan en [Kolo5]. Estas técnicas permiten a grupos de usuarios dibujar el sistema futuro sin ninguna restricción. Posteriormente, en una sesión de tormenta de ideas (*brainstorming*) se seleccionan las dos mejores ideas. Los usuarios reciben dinero imaginario y deben realizar una inversión en las características del sistema que desearían. Ello permite una extracción de los requisitos del sistema, del entorno de su aplicación y de la priorización de dichos requisitos.

Relacionadas con el diseño basado en componentes y orientado a servicios respectivamente, los enfoques de PORE [NM99] y SORE [TJWW07] se centran en la evaluación y reutilización de recursos existentes para satisfacer los requisitos del sistema. Sin embargo, en estas técnicas el usuario permanece en un segundo plano, centrándose fundamentalmente en los artefactos software. Ello puede ser contraproducente dado que el usuario es un elemento central en los Sistemas Ubicuos.

PC-RE [SFS05] trata de paliar este efecto considerando los requisitos a varios niveles, llegando hasta el nivel personal, y teniendo en cuenta la evolución de los requisitos cuando las dimensiones contextuales varían. Este enfoque permite una mayor personalización del sistema al tener en cuenta las necesidades particulares de los individuos o grupos de estos, pero puede provocar una gran proliferación de requisitos y conflictos entre ellos si no se manejan de manera sistemática.

3.7.2 Objetivos vs. Escenarios

Tanto las técnicas basadas en objetivos [AP98] [DDMvL97] [Yu97] [BPS10] [CSBW09] [GSB+08] [Cono7a] como las basadas en escenarios [Suto3] [Pot99] [SS07] [JB04] presentan numerosas ventajas e inconvenientes. Las primeras permiten un análisis de los requisitos de manera que se pueden ir descomponiendo, refinando y estudiando su refinamiento. Las últimas constituyen un modelo más cercano al usuario, ya que habitualmente éstos

tienden a expresar sus necesidades en términos de acciones y procesos.

En este sentido, y para el ámbito que nos ocupa, ambas técnicas deben usarse de manera complementaria: por una parte, un método basado en objetivos presenta abstracciones sobre las intenciones de los usuarios que nos permite razonar sobre ellas, evaluarlas y eventualmente derivarlas a un diseño de software; por otra, los escenarios sirven como ejemplos concretos de operación del sistema, los cuales ayudan a que el usuario final esté involucrado en el proceso de extracción y elaboración de los requisitos, potenciando así la validez de los mismos.

3.7.3 Tratamiento de los Requisitos No Funcionales

La gran mayoría de métodos se centran en la adquisición de los requisitos no funcionales del sistema futuro, prestando poca o ninguna atención a las propiedades de calidad (requisitos no funcionales). Dichas propiedades resultan de vital importancia para el éxito del sistema que se pretende construir, y si no son tratadas adecuadamente durante la especificación de requisitos y diseño, son difíciles de incorporar a un sistema ya construido.

Las propuestas de Chung *et al.* [CNYM00] [MCY99] y Yu *et al.* [Yu97] incorporan el concepto de *softgoal*, que se define como un objetivo para el cual no existe un límite claro para determinar su satisfacción o no. Este concepto permite representar los atributos de calidad del sistema, ya que habitualmente son aspectos transversales a diferentes partes del sistema y no puede determinarse con claridad si son satisfechos o no. Además, el NFR Framework [CNYM00] presenta un método para tratar sistemáticamente los requisitos no funcionales en el proceso de Ingeniería del Software, asegurando su cumplimiento en el sistema futuro.

Por otra parte, AORE [RSMA02] distingue el concepto de *concern*, que permite representar los requisitos no funcionales. Eventualmente los *concerns* son refinados en aspectos, los cuales son considerados de manera transversal en el sistema y permiten incorporar código necesario para garantizar algunas propiedades de calidad. Sin embargo, existen requisitos no funcionales que no pueden reflejarse siguiendo este enfoque, como pueden ser portabilidad o escalabilidad, los cuales deben ser tratados tomando otro tipo de decisiones (arquitectónicas, tecnológicas...).

Otras aportaciones, como lo expuesto en [TNA09] tratan los requisitos no funcionales creando una vista especial para ellos que se relaciona con las vistas funcional, de topología y de infraestructura de red. Sin embargo, este enfoque solamente se centra

en los requisitos de rendimiento (tiempo de respuesta, ancho de banda...) dejando de lado otros que también son de vital importancia.

Adicionalmente, la prioridad de satisfacción de los requisitos no funcionales en los Sistemas Ubicuos es cambiante y dependiente del contexto en el que se encuentre el sistema. Así, es necesario poder reflejar esta prioridad cambiante en el modelo de requisitos, así como las condiciones contextuales que implican un cambio de prioridad. La propuesta del NFR Framework [CNYM00] presenta la posibilidad de priorización de los requisitos, pero dicha priorización es fija y no existe la posibilidad de expresar las condiciones contextuales que hacen que cambie.

3.7.4 Relajación de Objetivos

En Sistemas Ubicuos es habitual encontrar situaciones adversas que obstaculicen la satisfacción de los requisitos del sistema. Así, es necesario que el método de Ingeniería de Requisitos empleado para este tipo de sistemas permita reflejar tanto la presencia de obstáculos en la satisfacción de requisitos, como la posibilidad de satisfacción parcial de algunos requisitos mientras dure la condición de excepción o se produzca la transición del sistema a un nuevo estado.

FLAGS [BPS10] y RELAX [CSBW09] permiten, mediante el uso de lógica difusa, la relajación en el cumplimiento de ciertos objetivos mientras se ven afectados por una situación adversa. Por otra parte, el concepto de *softgoal* de [Yu97] y [CNYM00] permite representar este tipo de objetivos, aunque es necesaria una extensión que permita incorporar información de las condiciones excepcionales sobre las que se permite la satisfacción parcial del objetivo, así como el nivel de satisfacción mínimo que es admisible.

3.7.5 Adaptación y Consciencia del Contexto

Los Sistemas Ubicuos son, por naturaleza y definición, inherentemente adaptativos, de manera que, en función de las condiciones del contexto que los rodea y de las preferencias del usuario, son capaces de modificar su comportamiento para dar un mejor servicio. Por tanto, un método de Ingeniería de Requisitos para Sistemas Ubicuos debería ser capaz de representar las posibles adaptaciones que debe realizar el sistema, así como las condiciones contextuales que le llevan a realizar dichas adaptaciones.

Como se ha comentado anteriormente, FLAGS [BPS10] y RELAX [CSBW09] dan soporte a la posibilidad de incorporar objetivos que reflejan adaptaciones del sistema cuando se producen situaciones adversas para el mismo. La transición entre estos estados se especifica mediante el uso de lógica difusa. Sin embargo, no se especifican transiciones en el sistema como adaptaciones para personalizar el sistema de acuerdo a las prioridades del usuario o al contexto que le rodea.

PC-RE [SFS05] tiene en cuenta el contexto del usuario en la elicitación de requisitos, pero se limita a la dimensión espacio-temporal de los mismos, sin tener en cuenta otros aspectos del contexto que rodea al usuario.

LoREM [GSB+08] es un método para sistemas dinámicos que indica cuáles son las posibles adaptaciones y cómo se realizan estas, pero carece de soporte para mostrar la influencia del contexto sobre las adaptaciones.

Por último, RE-CAWAR [SS07] es un método de Ingeniería de Requisitos para sistemas conscientes del contexto. Presenta varios modelos (usuario, dominio, plataforma...) que se integran y relacionan entre sí, pero carece de un modelo de contexto que indique bajo qué condiciones se producen las adaptaciones.

3.8 Resumen

En este capítulo se han presentado numerosos métodos y técnicas de Ingeniería de Requisitos, así como se ha realizado un análisis de su aplicabilidad para modelar Sistemas Ubicuos. De los resultados del análisis se desprende que no existe un método de Ingeniería de Requisitos que presente muchas de las características deseables para poder modelar sistemas software de calidad, y de forma específica las peculiaridades de los Sistemas Ubicuos. El método propuesto debe permitir:

- El uso de técnicas de elicitación de requisitos que tengan al usuario en un primer plano para garantizar la validez de los requisitos obtenidos y la usabilidad del sistema futuro.
- Una representación exhaustiva de los requisitos del sistema de manera que se pueda derivar de ellos un diseño del sistema.
- Un tratamiento sistemático de los Requisitos No Funcionales del sistema, así como la representación de la variabilidad en su priorización.

- La representación de diferentes situaciones en el contexto.
- La representación de las adaptaciones del sistema, tanto ante situaciones adversas como ante cambios en el contexto.

Aspecto	GBRAM	KAOS	I*	FLAGS	RELAX	SCRAM	ScenIC	NFR Framework	PORE	SORE	AORE	PC-RE	RE-CAWAR	LoREM	EUC	UWA
Adquisición de Requisitos	✓		✓			✓	✓		✓	✓	✓	✓			✓	
Modelado de Requisitos	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓
Orientado a la Reutilización								✓	✓	✓						
Basado en objetivos	✓	✓	✓	✓	✓		✓						✓	✓		✓
Basado en escenarios						✓	✓						✓		✓	
Tratamiento específico de NFR			✓					✓			✓					
Relajación de objetivos			∼	✓	✓			∼								
Adaptatividad / Dinamicidad				✓	✓							✓	✓	✓		
Consciencia del contexto					∼							✓	✓			

Tabla 3.1: Comparativa de los diferentes métodos y técnicas de Ingeniería de Requisitos. ✓completamente abordado, ∼ parcialmente abordado

4 DISEÑO DE SOFTWARE PARA SISTEMAS UBICUOS

Índice

4.1 Introducción

Tras analizar los principales métodos y técnicas de Ingeniería de Requisitos para Sistemas Ubicuos, en este capítulo se presentan diversos enfoques relacionados con varios aspectos del diseño de este tipo de sistemas desde el punto de vista de la Ingeniería del Software. En particular, los principales aspectos que se analizarán en este capítulo son:

Aspectos de diseño en sistemas ubicuos

- **Ingeniería dirigida por Modelos:** en la primera sección se estudiarán las características de este paradigma en el que se propone el empleo de los modelos como entidades de primer orden y su transformación y refinamiento en artefactos más concretos hasta llegar a la implementación del sistema. Se estudiarán los diferentes tipos de transformaciones que pueden encontrarse en la bibliografía y las tecnologías que

dan soporte a su realización. Asimismo, se analizarán algunos trabajos que aplican este paradigma en la realización de Sistemas Ubicuos.

- **Ingeniería de Procesos:** en este apartado se estudiarán las posibilidades existentes para la definición de métodos y procesos. En particular, se presentará el estándar SPEM 2.0, el cual permite la definición y descripción de procesos, y que cuenta con artefactos específicos para la descripción de procesos de Ingeniería del Software. Se comparará este estándar con otras propuestas existentes.

- **Arquitectura Orientada a Servicios:** el paradigma de orientación a servicios es ampliamente utilizado en la bibliografía sobre Sistemas Ubicuos dados los beneficios que presenta en el desarrollo de software (e.g. alta reusabilidad, bajo acoplamiento). Se estudiarán algunas metodologías existentes para el diseño de servicios, así como algunas de las tecnologías estándares que existen para su implementación.

- **Técnicas de Adaptación:** dada la importancia de la adaptación en los Sistemas Ubicuos (principalmente basada en la ocurrencia de cambios en el contexto del sistema), en este apartado se estudiarán las posibles técnicas existentes para la detección de dichos cambios, la toma de decisiones sobre la adaptación que debe realizarse, y los mecanismos existentes para realizar la adaptación.

En cada una de estas secciones se presentarán diversas propuestas existentes en la bibliografía. Asimismo se analizarán sus principales ventajas e inconvenientes. De esta forma, se intenta resaltar cómo para llevar a cabo el diseño de sistemas ubicuos la combinación y extensión de diferentes propuestas existentes permitiría aportar ciertos avances y ventajas.

4.2 Ingeniería dirigida por Modelos

Model-driven Engineering

La **Ingeniería dirigida por Modelos** (*Model-driven Engineering*, MDE) es un paradigma de desarrollo que trata de elevar el nivel de abstracción en la especificación de sistemas e incrementar la automatización del desarrollo. La idea consiste en la definición de modelos a diferentes niveles de abstracción y su correspondiente transformación hasta la obtención de un modelo que puede ser ejecutado, bien mediante la generación de código, bien mediante la interpretación de modelos.

Una visión particular de este enfoque es el estándar MDA [Gro03b], propuesto por OMG. En esta sección se profundiza en dicho estándar, así como se presentan los diferentes tipos de transformaciones que pueden encontrarse en la bibliografía y algunos lenguajes de transformación existentes.

4.2.1 *Model-driven Architecture* (MDA)

Model-driven Architecture

La **Arquitectura dirigida por Modelos** (*Model-driven Architecture*, MDA [Gro03b]) es un enfoque para el desarrollo, integración e interoperabilidad de sistemas software. Se basa en los estándares MOF (*Meta Object Facility* [Gro05a]), UML (*Unified Modeling Language 2.0* [Gro05b]), CWM (*Common Warehouse Metamodel* [Gro03a]) y XMI (*XML Metadata Interchange* [Gro11c]). La estrategia general consiste en separar la especificación de la operación del sistema de cualquier consideración técnica. En particular, MDA define tres puntos de vista:

- **Computation Independent Viewpoint (CIV):** vista independiente de la computación, se centra en el entorno del sistema y sus requisitos.
- **Platform Independent Viewpoint (PIV):** vista independiente de la plataforma, se centra en la operación del sistema, ocultando los detalles necesarios de una plataforma particular.
- **Platform Specific Viewpoint (PSV):** vista específica de plataforma, se centra en el uso de una plataforma específica por parte de un sistema.

Niveles de abstracción

Estos tres puntos de vista tienen como resultado tres modelos del sistema a diferentes niveles de abstracción: el **Modelo Independiente de la Computación** (*Computation Independent Model*, CIM), el Modelo Independiente de Plataforma (*Platform Independent Model*, PIM), y el Modelo Específico de Plataforma (*Platform Specific Model*, PSM), respectivamente.

El ciclo de vida MDA es similar al ciclo de vida tradicional: requisitos, análisis, diseño, codificación, pruebas y despliegue. Sin embargo, una de las mayores diferencias subyace en la naturaleza de los artefactos que se crean durante el proceso como resultado de una fase y entrada para la siguiente. Estos artefactos son modelos semiformales y pueden ser (parcialmente) interpretados por los ordenadores. El concepto de transformación de modelos es crucial para la obtención de ciertos beneficios en este ciclo de vida, tales como portabilidad y mantenibilidad. Tal como

se define en las especificaciones del OMG, la transformación es *el proceso de convertir un modelo a otro modelo del mismo sistema* (por ejemplo, de PIM a PSM). En las siguientes secciones se profundizará en los tipos de transformaciones entre modelos existentes.

4.2.2 Transformaciones entre modelos

Existen diferentes enfoques para la realización de una transformación de modelos. Habitualmente, una transformación entre modelos conlleva la incorporación de nueva información. Este procedimiento es especialmente necesario en transformaciones verticales, donde cada modelo tiene un nivel de abstracción diferente. En otras palabras, el modelo origen, creado a un nivel de abstracción más alto, debe refinarse con información adicional significativa en un modelo objetivo más concreto.

Para la obtención de estos modelos más concretos mediante transformación, los diferentes enfoques [CH03] existentes son:

Enfoques para la transformación de modelos

- **Marcado:** el proceso de transformación consta de dos etapas. Primero, se anota el modelo origen con algunas etiquetas que incorporan información adicional; el marcado se realiza normalmente sobre el modelo. A continuación, se emplea un mapeo para transformar los elementos marcados en el modelo origen en nuevos elementos del modelo destino, generando un nuevo artefacto.

- **Basado en modelos:** la transformación se realiza mediante el establecimiento de una correspondencia entre tipos en los diferentes modelos. Los elementos de los modelos origen y destino son subtipos de los tipos especificados en la transformación. Normalmente, la transformación es uno a uno, por lo que el proceso es determinístico y, en la mayoría de los casos, bidireccional.

- **Basado en metamodelos:** los modelos origen y destino son instanciaciones de sus correspondientes metamodelos. La transformación se especifica como un mapeo entre conceptos en los metamodelos origen y destino, lo cual permite la correspondiente transformación entre los modelos instanciados.

- **Aplicación de patrones:** de manera similar a la transformación basada en modelos, el empleo de patrones relaciona grupos de elementos en el modelo origen a otros grupos en el modelo destino. En lugar de aplicar las reglas de

transformación al modelo completo, pueden aplicarse sólo a aquellos elementos que cumplen con las características del patrón.

- **Manipulación directa:** se presenta al ingeniero una representación interna de los modelos, junto con una API (implementada como un *framework* orientado a objetos) para manipular los modelos.
- **Enfoques relacionales:** en este tipo de transformación, se especifican relaciones matemáticas y declarativas entre conceptos, junto con algunas restricciones. Se añade una semántica de ejecución para realizar la conversión.
- **Basado en grafos:** los modelos se representan como grafos tipados y etiquetados, especialmente diseñados para representar modelos basados en UML. Las reglas de transformación operan en patrones de los grafos, los cuales relacionan los modelos origen y destino.
- **Dirigido por estructuras:** el proceso de transformación consta de dos etapas. Primero, se crea la estructura del modelo destino aplicando las reglas de transformación. Después, esta estructura es poblada con atributos y referencias para completar la conversión.
- **Enfoques híbridos:** combinan varios de los enfoques previos.

4.2.3 Lenguajes de transformación

En esta sección se presentan algunas de las soluciones tecnológicas existentes para la transformación entre modelos y la generación de representaciones textuales a partir de modelos, las cuales permiten la generación de código. Los enfoques presentados son *Query-View-Transformation* (QVT), *ATLAS Transformation Language* (ATL) y *MOF Model to Text* (MOFM2T).

Query-View-Transformation (QVT)

El estándar *Query-View-Transformation* (QVT) [Gro11b], propuesto por OMG, es un conjunto de lenguajes para la especificación de transformaciones en MDA. QVT se basa en el estándar MOF; es decir, las transformaciones definidas de acuerdo a este estándar pueden verse como modelos en sí mismas, permitiendo la posibilidad de que sean transformadas también. Por lo general, sigue un enfoque basado en metamodelos.

QVT define tres tipos de lenguajes:

Lenguajes propuestos en QVT

- **QVT Operational:** es un lenguaje imperativo que permite transformaciones unidireccionales entre modelos.
- **QVT Relations:** es un lenguaje declarativo que soporta transformaciones unidireccionales y bidireccionales. Proporciona una representación textual y otra gráfica para describir las reglas de transformación. Puede ejecutarse en dos modos: *check only*, para comprobar la consistencia entre modelos, y *enforce*, para modificar un modelo para conseguir la consistencia.
- **QVT Core:** es también un lenguaje declarativo similar a QVT Relations, pero no ha sido completamente implementado dado que es menos expresivo que el anterior.

QVT permite la incorporación de expresiones OCL para definir restricciones en las transformaciones. También, proporciona un mecanismo de caja negra que permite la invocación de funciones expresadas en otros lenguajes. Como inconveniente, el estándar QVT no cubre transformaciones desde modelos a texto plano, y viceversa. El estándar MOFM2T [Gro08a] se encarga de esto.

ATLAS Transformation Language (ATL)

El lenguaje de transformación ATL (*ATLAS Transformation Language* [INR08]), propuesto por los grupos de investigación INRIA y LINA, es un lenguaje definido en términos de un metamodelo y una sintaxis textual. Aunque es, en esencia, un lenguaje declarativo, incorpora elementos imperativos para facilitar la expresión de transformaciones más complejas.

Un programa ATL consta de un conjunto de reglas que definen cómo se transforma un modelo origen en un modelo destino. Sigue un enfoque basado en metamodelos (ver sección 4.2.2). Los metamodelos y modelos de origen y destino se definen conforme a la especificación EMF (*Eclipse Modeling Framework*).

Cada regla de transformación tiene las siguientes partes:

Estructura de una regla ATL

- ***from*:** especifica el patrón que deben cumplir los elementos del modelo origen que van a ser transformados.
- ***using*:** opcional, en este bloque se definen las variables auxiliares necesarias para la transformación.
- ***to*:** especifica los elementos del modelo destino que se crearán.

- ***do*:** opcional, contiene un conjunto de órdenes imperativas que se ejecutan tras la inicialización de los elementos generados en el bloque *to*.

Cada elemento del modelo origen debe ser seleccionado, como máximo, por una regla de transformación. Para realizar de manera más sencilla la definición de transformaciones, existe la posibilidad de definir algunas funciones auxiliares (*helpers*).

ATL está disponible como un plugin para Eclipse y cuenta con un editor, una máquina virtual para su ejecución y un depurador para facilitar el trabajo del programador.

Model to Text (M2T)

En las secciones anteriores se ha centrado la atención en las opciones disponibles para la transformación entre modelos. Sin embargo, un paso importante en MDA consiste en el paso de los modelos PSM a código o documentación textual. Para este propósito, OMG propone el estándar *MOF Model to Text* (MOFM2T [Gro08a]).

Este estándar aborda la transformación de un modelo en una representación textual lineal. Para ello, hace uso de la definición de plantillas. Estas plantillas contienen el texto fijo que debe contener el resultado, junto con un conjunto de etiquetas que permiten parametrizarlas. Dichas etiquetas, durante la ejecución de la transformación, obtienen información de los modelos que se proporcionan como entrada. Esta información se transforma en fragmentos de texto que se insertan en las partes correspondientes de la plantilla.

El estándar permite la composición de plantillas para permitir transformaciones complejas. Las transformaciones muy extensas pueden estructurarse en módulos, similares a los paquetes UML.

Una implementación de este estándar es *Acceleo* [Gro06]. Es un motor de transformación que ha sido implementado como un plugin de Eclipse, haciendo uso de las facilidades del editor para la transformación de modelos.

4.2.4 Propuestas basadas en MDA en Sistemas Ubicuos

Gestión del contexto en otros enfoques

En la bibliografía pueden encontrarse varias propuestas sobre cómo aplicar los principios de la Ingeniería Dirigida por Modelos, y más precisamente de la especificación MDA [Gro03b], al desarrollo de sistemas ubicuos. Principalmente, estos enfoques se centran en la especificación, adquisición, gestión y uso

del contexto en la aplicación, dejando de lado otras características importantes, tales como la heterogeneidad de soluciones tecnológicas o las variaciones en la priorización.

Carencias de otros enfoques

Estas propuestas tienen en común el uso de ontologías para representar y gestionar el contexto [OGA+06] [SVP10]. Un enfoque más abstracto y no ligado a una tecnología particular podría ser más apropiado. De esta manera, se podrían emplear modelos a nivel CIM, y permitir al desarrollador que elija la opción para gestionar el contexto más apropiada para el proyecto en el que se esté trabajando.

Aunque el estándar MDA distingue tres niveles de abstracción (CIM, PIM y PSM), las propuestas examinadas [CCAT12] [dOdPdSB09] [OGA+06] [SVP10] [VH08] se centran inicialmente en el nivel PIM. No se proporciona información sobre cómo se trata el nivel CIM o cómo se generan los modelos a nivel PIM. Además, la mayoría de los métodos que aseguran seguir un enfoque dirigido por modelos no proporcionan detalles específicos sobre cómo se realiza la transformación entre modelos. En su lugar, se centran en la generación de código como la única transformación desde un modelo PIM a diferentes modelos PSM para diferentes plataformas tecnológicas [CCAT12] [SVP10].

Estrategias de transformación de otras propuestas

Para aquellos trabajos que proporcionan información sobre la transformación de modelos, pueden encontrarse diferentes enfoques. En [VH08], se presenta un enfoque basado en marcado, donde se incorpora una pequeña modificación: los autores proponen el uso de reglas de transformación parametrizadas, para incorporar contexto en PIMs en un proceso iterativo de transformación de modelos. Emplean su propio motor de transformación basado en una extensión de OCL [Gro11a]. Aparte de este enfoque, en [OGA+06] se propone una transformación que hace uso de XMap.

4.3 Ingeniería de Procesos

Existen diferentes opciones para la definición de procesos, tanto en el ámbito de la ingeniería del software como de propósito general. En esta sección se presentará el metamodelo de Ingeniería de Procesos para Software y Sistemas (SPEM), estándar para la definición de este tipo de metodologías, que será la opción que se empleará en los capítulos siguientes para la especificación de la metodología. Adicionalmente, se presentarán algunos otros enfoques para este propósito.

4.3.1 *Software and Systems Process Engineering Metamodel* (SPEM)

Objetivo de SPEM

El **Metamodelo de Ingeniería de Procesos para Software y Sistemas** (*Software and Systems Process Engineering Metamodel*, SPEM [Gro08b]), propuesto por el OMG, es un modelo basado en MOF [Gro05a] que trata de definir los procesos de desarrollo de software y sistemas, y sus componentes. El objetivo es ser capaz de acomodar un amplio rango de métodos y procesos de desarrollo con el conjunto mínimo de elementos necesarios para definirlos.

SPEM 2.0 puede usarse junto con los diagramas de actividad de UML 2.0 o con diagramas BPMN (*Business Process Modeling Notation* [Gro13]) para describir el comportamiento de un proceso de desarrollo.

Las características principales de SPEM 2.0 son:

Características de SPEM

- Proporciona una manera estandarizada de representar y gestionar bibliotecas de contenido reutilizable para la definición de métodos y procesos.
- Da soporte al desarrollo sistemático, gestión y crecimiento de procesos de ingeniería.
- Da soporte a la instanciación de diferentes configuraciones de métodos y procesos bajo demanda.
- Permite la representación de un proceso para proyectos de desarrollo.
- Separa de manera clara la definición del método de su aplicación concreta en un proceso.
- Mantiene diferentes alternativas para procesos de desarrollo de manera consistente.
- Proporciona varios modelos para diferentes ciclos de vida.
- Ofrece un mecanismo de plug-ins que permite una variabilidad y extensibilidad de procesos flexible.
- Proporciona conceptos para la definición de patrones de procesos para el ensamblado rápido de procesos.
- Se basa en componentes de procesos reemplazables y reutilizables que cumplen los principios de encapsulamiento.

El metamodelo de SPEM 2.0 ha sido dividido en siete paquetes diferentes con distintas responsabilidades. Dichos paquetes son:

- **Core:** contiene las metaclases base y las abstracciones para construir el resto de conceptos presentes en otros paquetes. Principalmente, proporciona la capacidad de crear cualificaciones definidas por el usuario para distinguir diferentes tipos de instancias de clases SPEM y para definir procesos SPEM 2.0.

- **Process Structure:** establece la base para definir modelos de procesos simples y flexibles. Proporciona mecanismos para la reutilización de procesos, permitiendo el ensamblado dinámico de procesos, lo cual es ideal para el ensamblado ad-hoc de procesos, tal y como ocurre en metodologías ágiles o equipos auto-organizados.

- **Process Behavior:** extiende el paquete anterior añadiendo la posibilidad de incorporar modelos de comportamiento, proporcionando enlaces a modelos de comportamiento existentes, tales como los diagramas de actividad UML 2.0.

- **Managed Content:** proporciona conceptos para gestionar las representaciones textuales que no pueden ser formalizadas con modelos, tales como documentos de buenas prácticas, *whitepapers* o guías.

- **Method Content:** presenta conceptos para definir conocimiento sobre el desarrollo, independiente de ciclos de vida, proyectos y procesos. Estos elementos pueden ser fácilmente reutilizados como parte de diferentes proyectos y procesos, y constituyen una base de conocimiento valiosa donde las buenas prácticas del desarrollo de software pueden concretarse.

- **Process with Methods:** define nuevos conceptos y redefine algunos existentes para poder integrar modelos de procesos expresados con las metaclases de *Process Structure* con los métodos reutilizables descritos empleando *Method Content*.

- **Method Plugin:** introduce conceptos para diseñar y gestionar bibliotecas y repositorios de procesos y métodos de desarrollo, mantenibles, de gran escala, reutilizables y configurables.

Ademas de estos paquetes, la especificación de SPEM 2.0 contiene el **SPEM 2.0 Base Plug-in**, un plug-in que contiene instancias de metaclases para conceptos que se emplean de manera habitual en el dominio de la ingeniería del software. Proporciona una buena fuente de activos reutilizables para definir procesos de

desarrollo de software. Los conceptos principales presentados tanto en este plug-in como en los paquetes previos se encuentran compilados en la Tabla 4.3 (al final del capítulo), junto con una breve descripción del concepto y el icono que lo representa.

Descripción de procesos en SPEM

Una descripción y definición típica de un proceso en SPEM puede incluir diferentes diagramas, tales como **diagramas de flujo de trabajo** (*workflow diagrams*), que expresan el orden de ejecución de las tareas; **diagramas de detalle de actividad** (*activity detail diagrams*), para describir en profundidad el trabajo que debe realizarse con una menor granularidad; **diagramas de dependencia de work products** (*work product dependency diagrams*), para reflejar las relaciones entre los diferentes que son requeridos, producidos o transformados en las diferentes tareas de los procesos; **diagramas de transición de estados en work products** (*work product state transition diagrams*), para mostrar los estados posibles en los que se puede encontrar un *Work Product*; **diagrams de perfil del equipo** (*team profile diagrams*), para reflejar la estructura de los diferentes roles involucrados en el proceso y cómo se organizan en un equipo; o **diagramas de componentes del proceso** (*process component diagrams*), para proporcionar una vista general de alto nivel sobre cómo se coordinan las diferentes fases del proceso completo de desarrollo, entre otros.

SPEM ha sido concebido como un metamodelo, pero también ha sido especificado en términos de un perfil UML, organizado en diferentes paquetes que extienden los elementos del núcleo de la especificación de UML. Este perfil UML será empleado a lo largo de este documento para describir la propuesta.

4.3.2 Otros enfoques de Ingeniería de Procesos

Aunque este trabajo hará uso del estándar SPEM 2.0 por las razones que se expondrán posteriormente, en esta sección se presentarán otras notaciones y enfoques existentes en el campo de la ingeniería de procesos. La tabla 4.1 resume estos enfoques.

Otros enfoques de Ingeniería de Procesos

- **PSL:** *Process Specification Language* [SIK98], es una ontología que contiene conceptos para especificar formalmente componentes de procesos y sus relaciones empleando CLIF (*Common Logic Interchange Format*). Proporciona notaciones gráfica y textual. Es un estándar ISO.

- **ARIS:** *Architecture of Integrated Information Systems* [Sch99], es un enfoque para el modelado de negocios que proporciona una colección de métodos para analizar procesos.

Proporciona un *framework* y una herramienta para dar soporte al modelado de procesos.

- **XPDL:** *XML Process Definition Language* [Coa12], es un formato estandarizado para el intercambio de definiciones de procesos de negocio entre diferentes productos de un *workflow*. Ha sido diseñado para contener todos y cada uno de los conceptos presentes en BPMN (*Business Process Modeling Notation*).

- **oXPDL:** es una ontología para XPDL [HMOG08] definida en OWL [Con04b] y WSML [Ont04] que incorpora conceptos de otras ontologías.

- **SUPER:** *Semantics Utilized for Process Management within and between Enterprises* [(IS07b], es un proyecto cuyo objetivo es proporcionar definiciones semánticas de la gestión de procesos de negocio.

- **GFO:** *General Formal Ontology* [Her10], es una ontología que trata de integrar procesos y objetos. Proporciona un *framework* para construir ontologías personalizadas específicas de un dominio concreto.

- **m3po:** *Multi Metamodel Process Ontology*, es una ontología, escrita en WSML [Ont04], que trata de combinar definiciones de *workflows*, para procesos de negocio internos, con definiciones de coreografía, para procesos de negocio externos. Puede usarse como una ontología para el intercambio de procesos.

Comparación de enfoques de Ingeniería de Procesos

La mayoría de estos enfoques proporcionan una representación textual del proceso [HOK06] [Her10] [(IS07b] [Coa12], pero sólo unos cuántos de ellos emplean notaciones gráficas como complemento [Sch99] [SIK98]. Las notaciones gráficas permiten una mejor comprensión de la definición del proceso y permiten visualizar la definición del proceso de un vistazo [OFR+12]. Más aún, es posible establecer diferentes niveles de abstracción para ocultar o mostrar detalles de la descripción. Esto también es posible en SPEM, dado que se emplea una notación gráfica, basada en UML 2.0, y una descripción textual en XML.

Muchas de estas propuestas están basadas en ontologías [HMOG08] [HOK06] [Her10] [(IS07b] [SIK98]. Ello permite la descripción formal de procesos y la comprobación semántica para verificar la corrección y consistencia de la definición. También, pueden inferirse nuevas propiedades del proceso gracias al uso de motores de razonamiento. SPEM también proporciona definiciones

(semi-) formales en términos de conformidad de un modelo a un metamodelo, pero carece de este mecanismo para inferir nuevo conocimiento.

Otro aspecto deseable son las herramientas de soporte, dado que facilita en gran medida la aplicación del enfoque [GMFFGS10]. Sólo uno de estos enfoques ofrece herramientas de soporte [Sch99]. Gracias a su estandarización, múltiples herramientas implementan SPEM y permiten a los desarrolladores describir sus procesos con facilidad.

Finalmente, nos centramos en la presencia de soporte específico para la ingeniería del software. Ninguno de los enfoques presentados anteriormente se basa en UML 2.0 o proporciona conceptos específicos de la ingeniería del software. SPEM 2.0 es el único que lo hace; ha sido revisado desde su especificación anterior para incorporar nuevos conceptos de UML 2.0 y, aunque trata de ser de propósito general, proporciona *SPEM Base Plugin*, donde se incluyen conceptos del ámbito de la ingeniería del software. SPEM parece ser la opción más apropiada para la especificación de un proceso de ingeniería del software, especialmente en el ámbito de MDA.

Aspecto	PSL [SIK98]	ARIS [Sch99]	XPDL [Coa12]	oXPDL [HMOG08]	SUPER [(IS07b]	GFO [Her10]	m3po [HOK06]	SPEM 2.0 [Gro08b]
Estándar	✓		✓					✓
Notación textual	✓		✓	✓	✓	✓	✓	✓
Notación gráfica	✓	✓						✓
Ontología	✓			✓	✓	✓	✓	
Soporte de herramientas	✓	✓	✓	✓				✓
Basado en UML								✓
Conceptos Software								✓

Tabla 4.1: Comparativa de los diferentes enfoques de Ingeniería de Procesos.

4.4 Arquitecturas Orientadas a Servicios

La **Arquitectura Orientada a Servicios** (*Service-oriented Architecture*, SOA [Erlo8]), es un conjunto de principios y metodologías para el diseño y desarrollo de software en forma de servicios interoperables. Este paradigma sigue los siguientes principios:

Principios SOA

- **Contrato estandarizado:** los servicios expresan sus capacidades en un contrato que se expone a otros y que contiene su funcionalidad, modelos y tipos de datos manejados, o las restricciones que se aplican, entre otros.
- **Bajo acoplamiento:** los servicios deben diseñarse de manera que se reduzca el nivel de dependencia entre el contrato, su implementación y su uso por parte de otros servicios.
- **Abstracción:** los servicios deben ocultar al máximo los detalles subyacentes a su implementación.
- **Reusabilidad:** los servicios deben ser diseñados de manera que no sean conscientes del contexto funcional en el que van a ser usados para promover su reutilización en otros entornos.
- **Autonomía:** para que los servicios puedan desarrollar sus capacidades, deben tener un nivel de control significativo sobre su entorno y recursos.
- **Carencia de estado:** para garantizar la disponibilidad de los servicios, éstos deben diseñarse de manera que no mantengan un estado, salvo en los casos en los que sea completamente requerido.
- **Posibilidad de ser descubierto:** para que los servicios puedan ser reutilizados con facilidad, éstos deben poder ser fácilmente encontrados.
- **Composición:** para poder realizar tareas más complejas a partir de servicios básicos, es necesario que éstos puedan componerse.

SOA propone la creación de servicios altamente reutilizables que permitan su descubrimiento y composición en otros más complejos para poder solucionar una mayor cantidad de problemas. En esta sección se presentan algunas de las metodologías y tecnologías existentes para el desarrollo basado en servicios.

4.4.1 Metodologías de Diseño basadas en SOA

Algunas de las metodologías que pueden encontrarse en la bibliografía para el desarrollo de una Arquitectura Orientada a Servicios [RDS07] son:

Metodologías SOA

- **IBM Service-Oriented Analysis and Design (SOAD) [ZKG04]:** consiste en un *framework* abstracto que contiene los elementos básicos que deberían estar en una metodología orientada a servicios. Está fundamentada en técnicas existentes de análisis y diseño orientado a objetos, diseño basado en componentes, gestión de procesos de negocio y características especiales de SOA.

- **IBM Service Oriented Modeling and Architecture (SOMA) [Ars04]:** propone tres pasos básicos para concebir los servicios: identificación de servicios, especificación de servicios y realización de servicios. Es un proceso iterativo e incremental, propietario de IBM.

- **SOA Repeatable Quality (RQ) [Mic07]:** propuesto por *Sun Microsystems*, es una metodología iterativa e incremental. La metodología consta de cinco fases: origen (*inception*), elaboración (*elaboration*), construcción (*construction*), transición (*transition*) y concepción (*conception*). La metodología es conforme al estándar UML.

- **CBDI-SAE Process [(IS07a]:** este proceso distingue cuatro disciplinas: consumición (*consume*), provisión (*provide*), gestión (*manage*) y habilitación (*enable*). El proceso cubre el ciclo de vida completo, proponiendo la integración entre la capa de negocio y la de IT mediante el análisis *top-down*, junto con el análisis *bottom-up* de sistemas legados.

- **Service-Oriented Architecture Framework (SOAF) [EAK06]:** consta de las siguientes fases: elicitación de la información, identificación de servicios, definición de servicios, realización de servicios, y gestión y planificación. Propone dos tipos de actividades: *to-be*, siguiendo un enfoque *top-down* analizando los requisitos de los procesos de negocio futuros, y *as-is*, siguiendo un enfoque *bottom-up* analizando los recursos existentes.

- **Service-Oriented Unified Process (SOUP) [Mit]:** es una metodología basada en el *Rational Unified Process*. Consta de las siguientes fases: concepción, definición, diseño, construcción, despliegue y soporte. Presenta algunas variaciones,

siguiendo el RUP para las fases iniciales, y una combinación de RUP + XP (*Extreme Programming*) para mantenimiento de SOA existentes.

- **Papazoglou [PVDH06]:** propone una metodología desde el punto de vista tanto del proveedor como del consumidor de servicios, que trata de cubrir el ciclo de vida completo. Parcialmente, se basa en modelos de desarrollo establecidos, como el RUP, el diseño basado en componentes y la gestión de procesos de negocio. Es un procesos iterativo e incremental, con una fase preparatoria y otras ocho fases principales.
- **Thomas Erl [Erl08]:** propone una metodología para el análisis y diseño de servicios. Cuenta con tres fases: identificación de servicios candidatos, especificación de servicios y realización de servicios como *Web Services*.
- **BPMN to BPEL [EWA06]:** en este enfoque se expresan los procesos de negocio empleando BPMN (*Business Process Modeling Notation*) y, empleando un conjunto de reglas de transformación, se transforma al lenguaje de ejecución BPEL (*Business Process Execution Language*).
- **Service Architectures [Jon05]:** es una metodología *top-down* consistente en los primeros pasos de un proyecto para asegurar propiedades SOA verdaderas en el desarrollo final. Es un proceso independiente de la tecnología.
- **Semantic and Syntactic Services [PB05]:** permite el modelado de servicios semánticos y sensibles al contexto. Propone, en primer lugar, un análisis de los requisitos funcionales para, a continuación, especificar las entidades funcionales de un servicio (componentes, conectores, puertos e interfaz del servicio). Además, incorpora aspectos no funcionales del servicio, tales como restricciones no funcionales sobre los componentes, interfaces para lidiar con el contexto o bloques de control del contexto.
- **Service Component Architecture (SCA) [BBB+05]:** según SCA, SOA no proporciona suficientes detalles sobre cómo desarrollar, reutilizar e integrar servicios. Esta metodología propone la creación de Servicios mediante módulos en los que se ensamblan componentes para realizar la implementación del servicio. De esta forma, se propone un alto acoplamiento entre los componentes que conforman un servicio, y un bajo acoplamiento entre los servicios.

Aspecto	SOAD [ZKG04]	SOMA [Ars04]	RQ [Mic07]	CBDI-SAE [(ISo7a]	SOAF [EAK06]	SOUP [Mit]	Papazoglou [PVDH06]	Erl [Erl08]	BPMN to BPEL [EWA06]	Service Arch. [Jono5]	Semantic Serv. [PBo5]	SCA [BBB+05]
Análisis de servicios	✓		✓	✓	✓	✓	✓	✓		✓	✓	
Diseño de servicios	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
Propietario	✓	✓	✓									
Basado en RUP						✓	✓					
Uso de estándares			✓					✓	✓			
Tratamiento de NFR											✓	
Basado en componentes							✓					✓

Tabla 4.2: Comparativa de las diferentes metodologías de diseño de servicios.

4.4.2 Tecnologías para SOA

En esta sección se presentan algunas de las soluciones tecnológicas existentes para la implementación de una Arquitectura Orientada a Servicios, entre los que se encuentran *Web Services* y *Web Ontology Language for Services*.

Web Services

Los *Web Services* son un conjunto de estándares y protocolos propuestos por el *World Wide Web Consortium* (W3C) basados en el estándar XML. De acuerdo a la definición del W3C, un servicio web es:

Definición de Servicio Web

> «*Un servicio web es un sistema software diseñado para dar soporte a la interacción interoperable entre dos máquinas por medio de una red de comunicaciones. Tiene una interfaz descrita en un formato procesable por una máquina. Otros sistemas interactúan con el servicio web mediante la manera prescrita en su descripción, empleando mensajes SOAP, habitualmente mediante una conexión HTTP con serialización XML en conjunción con otros estándares relacionados con la web.*»

Los principales estándares vinculados a los servicios web son:

Estándares para Web Services

- ***Simple Object Access Protocol* (SOAP) [Con07b]:** permite el intercambio de mensajes entre proveedor y consumidor.
- ***Web Service Description Language* (WSDL) [Con01]:** permite la definición de la interfaz del servicio, los tipos de datos usados y otros datos sobre los protocolos empleados para realizar las llamadas a las funciones expuestas.
- ***Universal Description, Discovery and Integration* (UDDI) [OAS02]:** permite la publicación de servicios para su descubrimiento y uso por parte de terceras partes.

Web Ontology Language for Services (OWL-S)

OWL-S [Con04a] es una ontología construida sobre el lenguaje de ontologías OWL que reemplaza la ontología previa DAML-S. Permite la descripción de servicios web semánticos. Permite a los usuarios y a los agentes software descubrir, invocar, componer y monitorizar automáticamente recursos web que ofrecen servicios, bajo unas restricciones especificadas.

La ontología tiene tres partes:

Partes de la ontología OWL-S

- ***Service profile:*** describe lo que hace el servicio de manera que sea legible por una persona, incluyendo el nombre y descripción del servicio, sus limitaciones, calidad de servicio o información de contacto.
- ***Process model:*** describe la interacción de un cliente con el servicio, incluyendo entradas, salidas, pre-condiciones y resultados de la ejecución del servicio.
- ***Service grounding:*** indica los detalles que necesita un cliente para interactuar con el servicio, tales como protocolos, formatos de mensajes o puertos. Requiere que se emplee conjuntamente el estándar WSDL para la descripción completa de estos aspectos.

4.5 Adaptación de Software en Sistemas Ubicuos

Podemos definir la adaptación de software como [KPP10]:

Definición de Adaptación

> «*La adaptación de software (en Sistemas Ubicuos) es un proceso reactivo disparado por un evento específico o un conjunto de eventos en el contexto, con el fin último de mejorar la calidad de servicio (QoS) percibida por el usuario final.* »

> «*La adaptación de software es la habilidad de la aplicación para alterar y reconfigurarse a sí misma como resultado de cambios en el contexto para ofrecer el mismo servicio de maneras diferentes cuando se demanda en diferentes contextos y en diferentes puntos en el tiempo.* »

De estas definiciones podemos extraer la importancia del contexto a la hora de realizar una adaptación de software. Existen múltiples definiciones de lo que se entiende por **contexto** en la bibliografía, pero una de las más aceptadas es la siguiente [Dey01]:

Definición de Contexto

> «*Cualquier información que pueda emplearse para caracterizar la situación de una entidad; una entidad es una persona, lugar u objeto que se considere relevante para la interacción entre un usuario y una aplicación, incluyendo al usuario y la aplicación mismos.* »

Dependiendo de las dimensiones consideradas, podemos establecer diferentes clasificaciones de los tipos de adaptaciones existentes [KPP10]:

Tipos de adaptaciones

- Según la funcionalidad:
  - **Funcional:** la adaptación modifica o corrige la funcionalidad del sistema.
  - **Extra-funcional:** la adaptación se centra en modificar propiedades de calidad del sistema, tales como eficiencia, precisión o seguridad.
- Según el objetivo:
  - **Correctiva:** se detecta un mal funcionamiento del sistema y se reemplaza la parte errónea por otra cuyo funcionamiento sea correcto.
  - **Adaptativa:** adapta el sistema en respuesta a cambios en el contexto que pueden afectar a su comportamiento.
  - **Extensiva:** añade nuevas funcionalidades o servicios que no existían previamente en el sistema.
  - **Perfectiva:** mejora una aplicación o sistema, incluso aunque su funcionamiento fuera correcto.
- Según el tiempo de decisión/aplicación:
  - **Estática:** se distinguen dos tipos:
    - **Adaptación de requisitos:** ocurre antes de la ejecución, cuando deben modificarse los requisitos del sistema.
    - **Adaptación de diseño:** ocurre antes de la ejecución, cuando existe un desajuste entre diferentes componentes del sistema durante el análisis del mismo.
  - **Dinámica:** ocurre en tiempo de ejecución. Se distinguen tres tipos:
    - **Reactiva:** la adaptación se produce como reacción a un cambio.
    - **Proactiva:** la adaptación se produce anticipándose a un cambio.
    - **Híbrida:** una combinación de los anteriores.
- Según la granularidad:
  - **Basada en parámetros:** la adaptación consiste en el ajuste de ciertas variables que afectan a ciertas partes del sistema.
  - **Composicional:** la adaptación consiste en cambios de partes completas del sistema.

De las definiciones anteriores se desprende que un proceso genérico de adaptación consiste, pues, en tres pasos:

Proceso genérico de adaptación en Sistemas Ubicuos

1. **Percepción y procesamiento del contexto**: se extraen datos sobre el contexto del usuario y del sistema y se procesan.
2. **Razonamiento:** con la información extraída, se trata de tomar una decisión sobre las adaptaciones que deberán realizarse.
3. **Reacción:** se aplican las adaptaciones que se hayan decidido en el sistema.

En las siguientes secciones se presentarán diferentes técnicas existentes en la bibliografía para la realización de cada uno de estos pasos. Para cada alternativa se evaluarán sus pros y contras.

4.5.1 Técnicas de razonamiento

El proceso de razonamiento en el ámbito de la adaptación consiste en decidir cuándo es necesario realizar una adaptación, qué alternativa es la que mejor satisface el objetivo global de la adaptación, y qué operaciones son necesarias para llevar al sistema al siguiente estado en su configuración. En este ámbito distinguimos cuatro categorías: técnicas basadas en acciones, técnicas basadas en objetivos, funciones de utilidad y adaptación por aprendizaje.

Técnicas basadas en acciones

Técnicas basadas en acciones o reglas

Las **técnicas de razonamiento basadas en acciones**, o basadas en reglas, consisten en la especificación de un conjunto de reglas que aplican una acción cuando se cumplen unas condiciones [KD07]. Es decir, habitualmente se cuenta con un conjunto de reglas del tipo "SI (condición) ENTONCES (acción)", las cuales son disparadas cuando ocurre un evento.

El principal problema de este tipo de reglas es que su evaluación es dicotómica (verdadero o falso), lo cual no es adecuado para entornos altamente dinámicos tales como los entornos de computación ubicua. Para paliar este efecto, puede hacerse uso de reglas difusas (*fuzzy rules*) [Wan96], que introducen ciertos niveles en la satisfacción de las condiciones.

Estos enfoques son apropiados ya que no consumen una gran cantidad de recursos para realizar el razonamiento, y el proceso de decisión es altamente trazable. Como inconveniente, puede destacarse que implican un gran esfuerzo de desarrollo dado que

hay que especificar todas las opciones posibles en términos de un conjunto de reglas, las cuales son difíciles de modificar de manera dinámica para hacerlas evolucionar con el sistema.

Técnicas basadas en objetivos

Técnicas basadas en objetivos

Las **técnicas de razonamiento basadas en objetivos** consisten en la especificación de un estado deseable para el sistema, o un criterio que caracteriza un conjunto de estados deseables [KW04]. De esta forma, el sistema decide qué adaptaciones tomar de manera que se alcance el estado deseado.

Al igual que los enfoques basados en reglas, este tipo de técnicas tienen la ventaja de ser eficientes en términos de consumo de recursos, y poseen una alta trazabilidad del proceso de razonamiento. Por el contrario, requieren algoritmos sofisticados de planificación y modelado de los objetivos. Este enfoque no permite capturar la relación existente entre los objetivos y las adaptaciones, y además implica la necesidad de introducir mecanismos de resolución para arbitrar en los conflictos entre diferentes objetivos.

Funciones de utilidad

Técnicas basadas en Funciones de Utilidad

Las **técnicas de razonamiento basadas en funciones de utilidad** son una extensión natural de las técnicas basadas en objetivos. Estas funciones proporcionan una medida de las preferencias del sistema en cuanto a los diferentes objetivos, asignando una puntuación a cada estado deseable [NS99] [GCH+04]. De esta manera, teniendo en cuenta dichos valores, se toma una decisión tratando de optimizar estas funciones.

Estas técnicas permiten una mayor posibilidad de modificación de los mecanismos de razonamiento simplemente cambiando las funciones de utilidad. Por el contrario, presentan problemas de trazabilidad en el razonamiento, así como su uso es difícil para los desarrolladores en sistemas complejos, dada la dificultad de expresar todas las preferencias de adaptación en términos de una función.

Adaptación por aprendizaje

Técnicas de adaptación por aprendizaje

Las **técnicas de razonamiento por aprendizaje** son las más complejas. Requieren la construcción de una base de conocimiento previa sobre la que se realiza el razonamiento, la cual va cambiando con la experiencia adquirida por el sistema a lo largo de su ejecución. Distinguimos dos tipos de técnicas:

- **Razonamiento basado en casos:** cuentan con una base de conocimiento en la que se expresan casos típicos que pueden encontrarse a lo largo de la ejecución del sistema, junto con las posibles soluciones que se dan a dichos casos [Lea96] [PCN06]. El proceso habitual seguido en la aplicación de estas técnicas consiste en consultar la base de conocimiento, recuperar el/los caso/s más similar/es, y realizar una adaptación de la solución para adecuarse a la situación que ocupa.

  Para realizar la adaptación de la solución existen varias alternativas:

  - **Null:** no se realiza adaptación, sino que se emplea un algoritmo para decidir cuál es la solución más adecuada.
  - **Transformación:** se realizan cambios estructurales en la solución basándose en un conjunto de reglas.
  - **Generación:** se genera una nueva solución desde cero, haciendo uso de las soluciones existentes.

- ***Reinforcement Learning*:** realizan aprendizaje no supervisado mediante ensayo y error. Se seleccionan acciones para maximizar un objetivo, otorgando una recompensa a largo plazo si se realiza la adaptación de manera correcta [DCCC05]. El proceso de razonamiento se basa en un problema de decisión de Markov, donde S_t es el estado del sistema en el instante t, $A(S_t)$ es el conjunto de acciones disponibles que pueden realizarse en este estado, y r_t es la recompensa que se obtiene. De esta forma:

$$S_t \xrightarrow{a \in A(S_t)} S_{t+1}, r_{t+1} \tag{4.1}$$

  Es decir, hay que elegir una acción $a \in A(S_t)$ que maximice la recompensa.

4.5.2 Mecanismos de adaptación

Una vez decidida la acción que se debe tomar, hay que aplicar la adaptación correspondiente. En esta sección se detallan algunas de las alternativas existentes, tales como código móvil, adaptación de parámetros, adaptación por composición y *Aspect Weaving*.

Código móvil

Las técnicas basadas en **movilidad de código** consisten, como su nombre indica, en migrar o mover instancias del código en ejecución entre diferentes dispositivos. Dependiendo del tipo de movilidad, se distingue:

Tipos de movilidad de código

- **Movilidad fuerte (*strong*):** se mueve el código y el estado de ejecución.
- **Movilidad débil (*weak*):** se mueve el código y un conjunto de datos de inicialización.

Por otra parte, dependiendo de quién invoque la movilidad, existen dos opciones: *pushing*, donde una entidad envía el código que debe ejecutarse en la unidad de destino; y *pulling*, donde una entidad solicita el código que debe ejecutar [ZME03]. En base a estos criterios, se distinguen los siguientes paradigmas:

Paradigmas de movilidad de código

- **Cliente-servidor:** es el enfoque más utilizado, en el que no existe una migración real del código, sino que se realiza una petición de ejecución de un código ubicado en otro dispositivo.
- ***Code on Demand (COD)*:** la entidad solicita el código que debe ejecutar. Es un enfoque apropiado cuando los recursos del dispositivo son limitados y no puede cargarse toda la funcionalidad del sistema.
- **Evaluación remota:** la entidad envía el código que debe ejecutarse en un entorno de ejecución remoto. Si se acepta, se despliega el código y se ejecuta.
- **Agentes móviles:** el agente es colocado en la red y se traslada entre los diferentes nodos para ejecutarse donde sea más conveniente.

Adaptación de parámetros

Adaptación por parámetrización

Las técnicas basadas en **adaptación de parámetros** han sido ampliamente utilizadas en múltiples sistemas para permitir variaciones en la ejecución de los mismos [KR12]. Consiste en la modificación de ciertas variables de control del sistema para configurar diferentes partes del mismo.

Como inconveniente, hay que destacar que es un enfoque poco eficiente, o incluso no factible, cuando el conjunto de adaptaciones es grande, siendo más apropiado para adaptaciones de granularidad fina.

Adaptación por composición

Las técnicas basadas en **adaptación por composición** consisten en la sustitución, adición o eliminación de determinados componentes del sistema dependiendo de las necesidades. Tienen su base en la separación de responsabilidades propia de la programación orientada a aspectos, diseño basado en componentes, reflexión computacional (la habilidad de un sistema para razonar sobre sí mismo) y los modelos de arquitectura [AC03] [MSKC04].

Adaptación por composición

Según cuando se realiza la composición, distinguimos entre composición estática y dinámica. La composición estática es aquella que se realiza en tiempo de diseño, compilación o despliegue. La composición dinámica es la que se realiza en tiempo de ejecución, y a su vez podemos distinguir entre software ajustable (*tunable softwtare*), donde se ajustan ciertos componentes, o software mutable (*mutable software*), que permite la recomposición, alterando la lógica de la aplicación.

Tipos de composición

Aspect weaving

Por último, las técnicas basadas en ***aspect weaving*** han sido propuestas por la programación orientada a aspectos. Consisten en la implementación individual de los *cross-cutting concerns* como aspectos. Estos aspectos son ejecutados en los llamados *join points*, que representan condiciones en tiempo de ejecución o puntos en la estructura del código [RBE08].

Aspect weaving

El proceso de inserción o eliminación de los aspectos en el código puede ocurrir en tiempo de ejecución, carga o compilación.

4.6 Resumen

En este capítulo se han analizado diferentes aspectos relacionados con la fase de diseño de software.

El paradigma de la Ingeniería Dirigida por Modelos ha sido ampliamente utilizado tanto en otros ámbitos como en Sistemas Ubicuos con éxito. Su aplicación proporciona numerosas ventajas, tales como una alta productividad o una reducción del número de fallos. El estudio realizado revela que existen, además de desarrollos particulares que lo emplean, ciertas metodologías que abogan por su uso, fomentando la realización de modelos y su transformación hasta la obtención de modelos que sean ejecutables, o bien, a los que se les puedan aplicar técnicas de generación de código. No obstante, del estudio realizado se desprende que mayoritariamente estas metodologías se centran en la transforma-

ción entre los niveles PIM y PSM, y en la generación de código, mientras que la transformación entre el nivel CIM y PIM raramente se realiza. Por tanto, y con el objetivo de proporcionar un soporte metodológico completo que abarque desde el nivel CIM hasta el PSM, en este trabajo se explorará la posibilidad de realizar transformaciones entre todos los niveles.

La comparación del estándar SPEM 2.0 para Ingeniería de Procesos con otras propuestas existentes ha revelado que es la opción más adecuada para la definición de un proceso o método de Ingeniería del Software puesto que cuenta con artefactos específicos para modelar las tareas que habitualmente se encuentran en este ámbito. Por tanto, se empleará este estándar para modelar, definir y describir los métodos de Ingeniería de Requisitos y Diseño que se propondrán como parte de esta tesis.

El estudio de la bibliografía también revela que existe una amplia variedad de metodologías para el diseño de servicios, ya que éstos son particularmente apropiados para el desarrollo de Sistemas Ubicuos puesto que pueden ser descubiertos dinámicamente en diferentes entornos donde se encuentre el usuario. Sin embargo, la mayoría de ellas proponen un conjunto de guías generales y carecen de detalles claros sobre cómo desarrollar, reutilizar e integrar servicios. Además, estas metodologías son de propósito general y no incorporan algunas de las características particulares de los Sistemas Ubicuos, tales como la consciencia del contexto o adaptación. Por tanto, con el objetivo de dar soporte al diseño de servicios para Sistemas Ubicuos, se realizara una propuesta de un método de diseño de servicios que tenga en cuenta sus características especiales aprovechando las ventajas existentes en las metodologías analizadas.

Por último, las técnicas de razonamiento y los mecanismos de adaptación analizados proporcionan un amplio conjunto de técnicas disponibles, cada una con sus ventajas e inconvenientes. De manera general, y dadas las características de los Sistemas Ubicuos, no es posible determinar cuál es mejor sobre las demás, sino que dependiendo de ciertas restricciones (poder de computación, almacenamiento, granularidad de las adaptaciones) será necesario aplicar unas técnicas u otras.

Una vez presentada esta revisión de propuestas existentes, en los siguientes capítulos se realizará la propuesta metodológica que trata de cubrir las carencias detectadas tras este análisis.

Concept	Description	Icon
Deliverable	*Work Product* que proporciona una descripción sobre el empaquetado de otros *Work Products*, y que puede ser entregado a una tercera parte	
Delivery Process	Proceso que describe un ciclo de vida completo de principio a fin y que puede emplearse como plantilla para planificar y ejecutar un proyecto similar	
Guidance	Elemento que proporciona información adicional sobre otros elementos; ejemplos de esto son *Guidelines*, *Checklists*, *Reusable Assets* o *Whitepapers*, entre otros	
Iteration	Grupo de tareas que son realizadas más de una vez, organizando el trabajo en ciclos repetitivos	
Milestone	Elemento de trabajo que representa un evento significativo en el proceso de desarrollo	
Package	Elemento que permite la agrupación de diferentes instancias con un aspecto común	
Process Component	Paquete que contiene un proceso que aplica los principios de encapsulamiento. Requiere y produce *Work Products* como resultado de su ejecución	
Role	Conjunto de habilidades, competencias y responsabilidades relacionadas. Realiza tareas y es responsable de *Work Products*	
Step	Sub-unidad de trabajo en la que se divide una tarea	
Task	Definición de trabajo realizada por un *Role*. Requiere y produce *Work Products* durante su ejecución	
Work Definition	Clasificador abstracto que generaliza todas las representaciones de trabajo en SPEM 2.0	
Work Product	Elemento que se usa, modifica o produce como resultado de la ejecución de una tarea	

Tabla 4.3: Conceptos principales de SPEM 2.0 usados en este documento, junto con sus correspondientes iconos

Parte III

Propuesta metodológica para el desarrollo de Sistemas Ubicuos

5 REUBI: MÉTODO DE INGENIERÍA DE REQUISITOS PARA SISTEMAS UBICUOS

Índice

5.1 Introducción

En el capítulo 3 se ha presentado, a modo de revisión del estado de la técnica, un conjunto amplio de métodos y técnicas que comprende la Ingeniería de Requisitos, tanto de propósito general (orientados al análisis de cualquier tipo de sistemas software), como de propósito específico (orientado a sistemas software dinámicos, adaptativos y ubicuos). Dichos métodos no cubren muchas de las características específicas presentes en los Sistemas Ubicuos, tal y como se deriva del análisis presentado en la sección 3.7 del mencionado capítulo.

De este modo, y de acuerdo a los objetivos planteados para este trabajo de investigación, en este capítulo se presenta un **Método de Ingeniería de Requisitos para Sistemas Ubicuos**, denominado REUBI, cuyo propósito es cubrir aquellas carencias que se han detectado en Capítulo 3 y proporcionar un método basado en un conjunto de modelos definidos formalmente que

permita tratar de manera sistemática los requisitos en este tipo de sistemas. Más concretamente, el método:

- Se centra en el concepto de objetivo, permitiendo su descomposición jerárquica en sub-objetivos de más bajo nivel y mayor concreción.
- Proporciona guías para el descubrimiento de los objetivos iniciales del sistema.
- Permite realizar un análisis de las situaciones adversas que debe afrontar el sistema, proporcionando un conjunto de guías para mostrar cómo solventarlas.
- Permite modelar las contribuciones de diferentes decisiones arquitectónicas y de diseño, y sus combinaciones, a la consecución de los objetivos.
- Recoge las justificaciones que dan soporte a las decisiones tomadas durante el análisis de los objetivos.
- Muestra el impacto del contexto en los objetivos que deben ser satisfechos[1] en cada momento.
- Permite realizar una priorización variable de la satisfacción de los objetivos, mostrando la influencia del contexto en los cambios de prioridades.
- Presenta un procedimiento de evaluación detallado que permite determinar qué decisiones son las más adecuadas para la satisfacción de los objetivos con unas condiciones de contexto determinadas.

La Figura 5.1 muestra las etapas generales propuestas para el método REUBI. En las siguientes secciones se definen formalmente los modelos propuestos y se describen en detalle las actividades comprendidas en dichas etapas. En el capítulo 9 se pretende ilustrar y validar este método por medio de su aplicación a un caso real.

1 Aunque cuando nos referimos a objetivos se suele emplear el término «alcanzado» cuando han sido conseguidos, en lo sucesivo se emplea el término «satisfecho» dado que en el ámbito de la Ingeniería de Requisitos se emplean los términos *satisfied* y *satisficed* para este propósito.

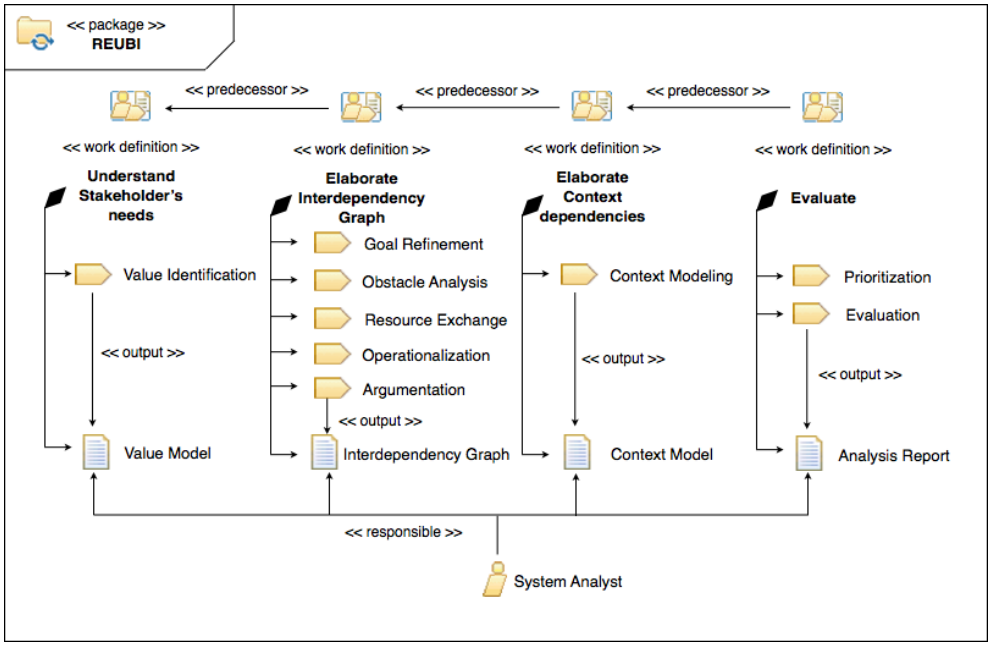

Figura 5.1: Etapas en el método de Ingeniería de Requisitos para Sistemas Ubicuos (REUBI), especificado en notación SPEM (Véase leyenda en Tabla 4.3).

5.2 Modelo de Valores

La fase inicial de REUBI consiste en la elaboración de un **Modelo de Valores**. Se define como:

Definición 5.1. Modelo de Valores
Es una representación de los actores involucrados en el Sistema Ubicuo que se ha de construir, sus intereses y las formas en las que éstos son intercambiados en el sistema.

Metamodelo para un Modelo de Valores

Un Modelo de Valores se construye conforme a un metamodelo (mostrado en la Figura 5.2) en el que los principales conceptos son ***actor***, ***value*** y ***value enhancer***.

Definición 5.2. Actor
Un Actor es cualquier individuo, humano o software que tiene interés en el sistema.

Definición 5.3. Value
Un *Value*[2] es cualquier bien (objeto material), servicio o información que está sujeto a un intercambio entre actores del sistema.

2 Valor

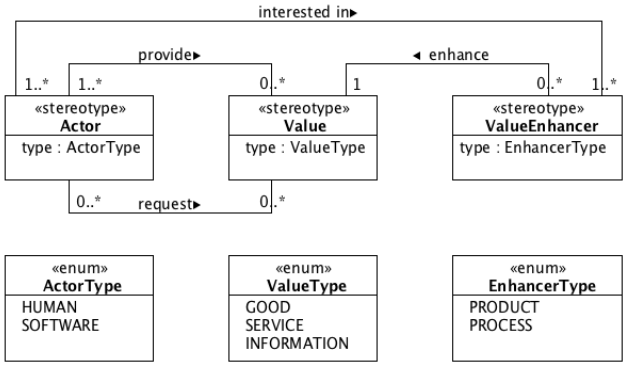

Figura 5.2: Metamodelo para el Modelo de Valores.

Definición 5.4. Value enhancer
Un *Value enhancer*[3] es cualquier propiedad de calidad que mejora o potencia un *value* o el proceso de su adquisición.

Identificación de Actores

En primer lugar, los actores deben ser determinados. Éstos pueden ser representados por un **rol** o una **posición**, y pueden organizarse jerárquicamente de manera similar a como se realiza en el modelado de los casos de uso (Figura 5.3).

Identificación de Valores

Los actores presentan relaciones de **provisión** y **demanda** de los valores. Así, para cada valor, debe existir al menos un actor que lo demanda y uno que lo ofrece. De igual manera, los **value enhancers** deben estar ligados al valor que potencian, y al actor que tiene interés en dicha mejora. Los valores y sus potenciadores representan las metas y objetivos de más alto nivel del sistema que se debe conseguir.

Guías de modelado

Las siguientes guías pueden contribuir al descubrimiento de los elementos del Modelo de Valores:

Modelado de Actores

- ¿Quién ofrece/produce un valor?
- ¿Quién demanda/consume un valor?
- ¿Quién está interesado en la calidad de un valor?
- ¿Quién está interesado en la calidad de la adquisición de un valor?

3 Potenciador de valor

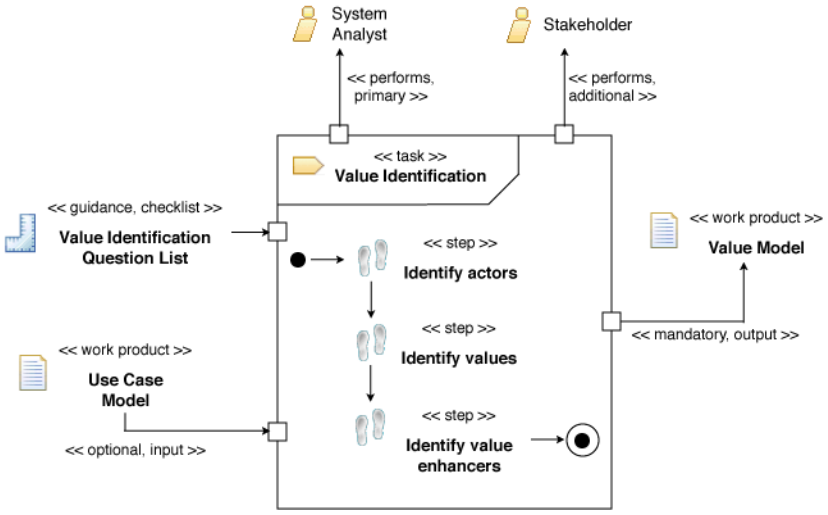

Figura 5.3: Definición de la tarea de Definición de Valores en notación de SPEM (Véase leyenda en Tabla 4.3).

- ¿Qué rol debe desempeñarse para obtener/producir un valor?
- ¿Puede especializarse este actor?

Modelado de Valores

- ¿Qué se intercambia en el sistema?
- ¿Qué tipo de valores se intercambian? ¿Bienes, servicios o información?
- ¿Quién proporciona este valor?
- ¿Quién demanda este valor?

Modelado de Potenciadores

- ¿Cómo puede mejorar este valor?
- ¿Cómo puede mejorar el proceso de adquisición de este valor?
- ¿Cuál es la calidad que se espera del valor?
- ¿Cuál es la calidad que se espera del proceso de adquisición del valor?
- ¿Existen restricciones temporales en el intercambio del valor?

- ¿Existen restricciones de seguridad en el intercambio del valor?
- ¿Existen restricciones de coste en el intercambio del valor?
- ¿Existen restricciones de precisión/fiabilidad en el intercambio del valor?
- ¿Es flexible el proceso de adquisición del valor?
- ¿Puede adquirirse el valor de formas diferentes?
- ¿Pueden adquirirse diferentes versiones del valor?
- ¿Puede decidirse la forma en la que se adquiere el valor?
- ¿Quién tiene interés en que la calidad del valor sea de esta forma?
- ¿Quién tiene interés en que la calidad del proceso de adquisición sea de esta forma?

5.3 Refinamiento de *Goals* y *Softgoals*

Grafo de Interdependencia

El Modelo de Valores sirve para identificar los objetivos generales del sistema e identificar los límites del mismo. Una vez obtenidos dichos objetivos generales, es necesario derivar objetivos más concretos y refinarlos progresivamente para ganar un mayor conocimiento del sistema que se ha de desarrollar. Dicho proceso es modelado en un **Grafo de Interdependencia**, cuyo metamodelo se muestra en la Figura 5.4. Dicho metamodelo ha sido definido haciendo uso de elementos que están habitualmente presentes en métodos de Ingeniería de Requisitos, a los que se les han añadido nuevos elementos que permiten modelar las características propias de los Sistemas Ubicuos en base a las carencias detectadas en la revisión bibliográfica.

Goals y Softgoals

Los valores y sus potenciadores recogidos en el Modelo de Valores se corresponden, respectivamente, con ***goals*** y ***softgoals***, los cuales se definen como:

Definición 5.5. Goal
Un *Goal* es un objetivo del sistema para el cual existe un criterio claro y bien definido que permite determinar su satisfacción o no. Es decir:

$$\forall \alpha : \alpha \in Goal \Leftrightarrow \exists f : f(\alpha) \mapsto \{0, 1\} \tag{5.1}$$

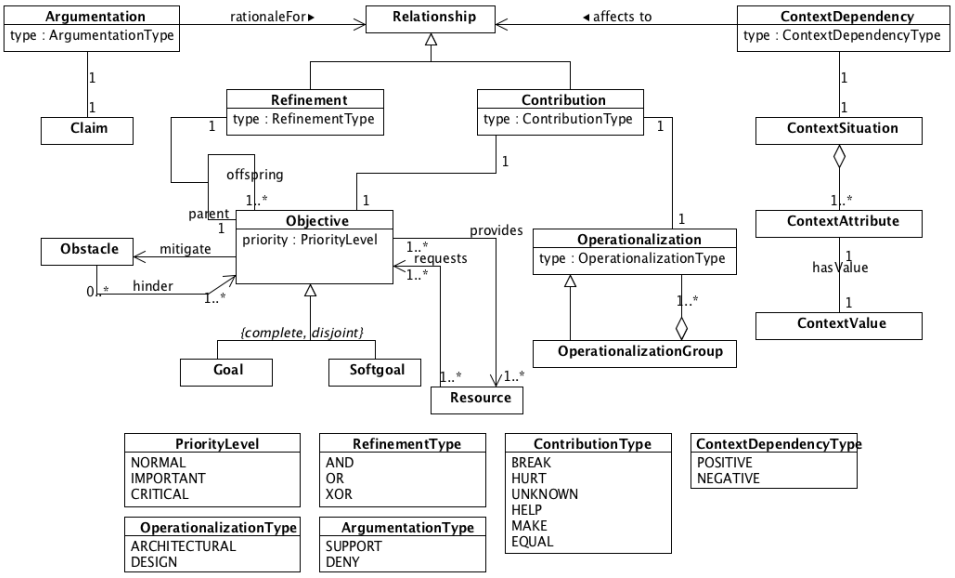

Figura 5.4: Metamodelo para el Grafo de Interdependencia.

Definición 5.6. Softgoal
Un *Softgoal* es un objetivo del sistema para el cual no existe un criterio que permite determinar sin ambigüedad su satisfacción, sino que se dice que está satisfecho cuando existe suficiente evidencia positiva y poca negativa para afirmarlo. Es decir:

$$\forall \alpha : \alpha \in Softgoal \Leftrightarrow \neg \exists f : f(\alpha) \mapsto \{0,1\} \tag{5.2}$$

Definición 5.7. Objetivo
Un objetivo del sistema es un *goal* o un *softgoal*[4]:

$$\forall \alpha : \alpha \in Objective \Leftrightarrow \alpha \in Goal \vee \alpha \in Softgoal \tag{5.3}$$

Catálogos

No todos los *goals* y *softgoals* pueden ser derivados del Modelo de Valores, sino que es necesario realizar refinamientos que conduzcan a una recopilación completa y lo más exhaustiva posible de los objetivos del sistema. Un enfoque posible consiste

4 En lo sucesivo, cuando se use el término *objetivo* se referirá indistintamente a *goal* o *softgoal*

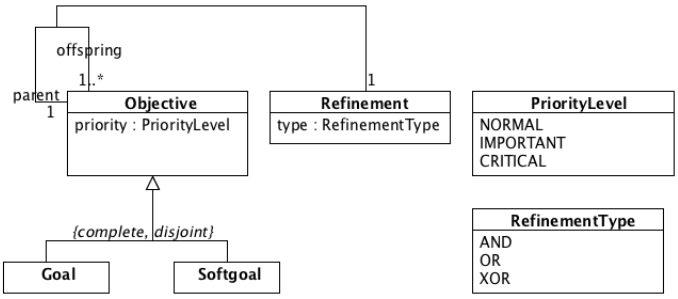

Figura 5.5: Metamodelo de objetivos.

en emplear catálogos o patrones, como se verá en el capítulo siguiente.

Definición 5.8. **Catálogo**
Un Catálogo es un conjunto de patrones que recoge relaciones habituales entre *goals* y *softgoals*.

Tipo y tema de los objetivos

Los objetivos tienen un **tipo** y un **tema**. El tipo está relacionado con lo que debe realizarse, mientras que el tema indica cuál es el sujeto de la aplicación.

Como ejemplo, consideremos el caso del diseño de un Sistema de Posicionamiento; un posible *softgoal* podría ser *Lograr Escalabilidad del Sistema de Posicionamiento*. En este caso, el tipo del *softgoal* sería *Lograr Escalabilidad* y el tema, *Sistema de Posicionamiento*.

Descomposición de objetivos

Los objetivos pueden descomponerse en otros de más bajo nivel teniendo en cuenta su tipo o su tema. Así, en el ejemplo anterior, una descomposición por tipo daría lugar a los *softgoals Lograr Escalabilidad Geográfica del Sistema de Posicionamiento* (mayor cobertura espacial) y *Lograr Escalabilidad en Densidad del Sistema de Posicionamiento* (mayor número de unidades localizadas). En cambio, una descomposición por tema daría lugar a *Lograr Escalabilidad en el Posicionamiento en Interiores* y *Lograr Escalabilidad en el Posicionamiento en Exteriores*.

De esta forma, podemos proceder a la descomposición de los *goals* y *softgoals* mostrando las contribuciones que los objetivos de más bajo nivel hacen a los objetivos de más alto nivel. Para la descomposición, es recomendable usar un conjunto reducido de verbos para especificar los objetivos con el propósito de evitar la aparición de ambigüedades. Distinguimos tres tipos de descomposiciones: **Inclusiva**, **Alternativa** y **Exclusiva**.

Definición 5.9. **Descomposición Inclusiva**

Descomposición tipo AND

Una descomposición es inclusiva (o de tipo AND) si para que

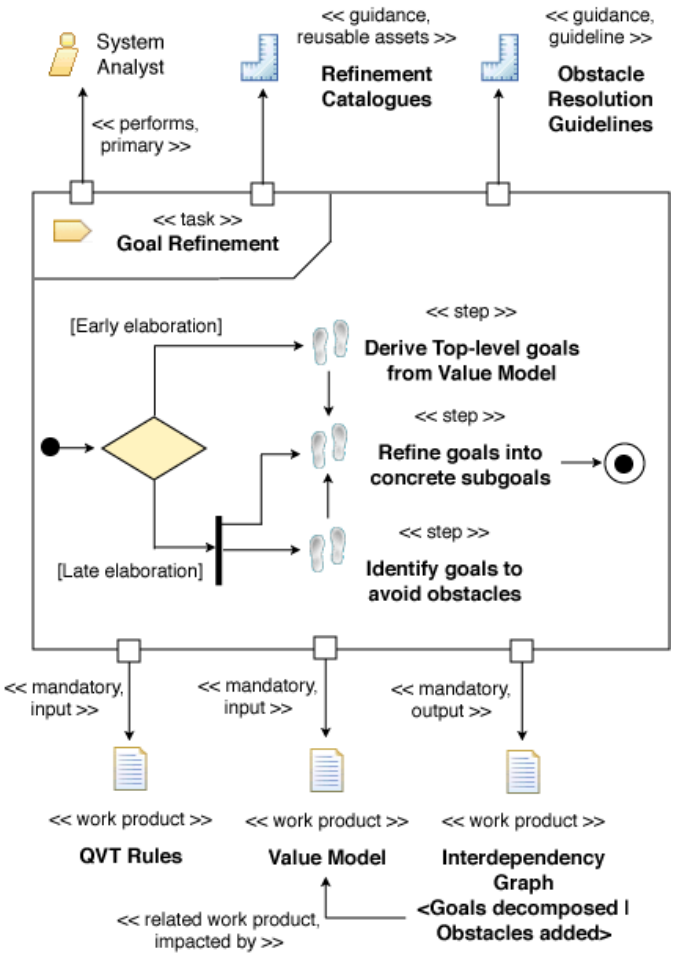

Figura 5.6: Definición de la tarea de Descomposición de Objetivos (Véase leyenda en Tabla 4.3)

el objetivo de alto nivel (padre) esté satisfecho, es necesario que todos los objetivos de más bajo nivel (hijos) sean satisfechos. Es decir, sea α un objetivo de alto nivel, $C = \{\alpha_1 \dots \alpha_n\}$ los objetivos de bajo nivel en los que se descompone, y sat una función booleana que determina la satisfacción o no del objetivo:

$$sat(\alpha) \Leftrightarrow \forall \alpha_i \in C : sat(\alpha_i) \tag{5.4}$$

Notaremos la descomposición por:

$$AND(\alpha, \{\alpha_1, \dots, \alpha_n\}) \tag{5.5}$$

Definición 5.10. Descomposición Alternativa
Una descomposición es alternativa (o de tipo OR) si para que el

Descomposición tipo OR

objetivo de alto nivel (padre) sea satisfecho, es necesario que al menos uno de los objetivos de más bajo nivel (hijos) sea satisfecho. Es decir, sea α un objetivo de alto nivel, $C = \{\alpha_1 \dots \alpha_n\}$ los objetivos de bajo nivel en los que se descompone, y *sat* una función booleana que determina la satisfacción o no del objetivo:

$$sat(\alpha) \Leftrightarrow \exists \alpha_i \in C : sat(\alpha_i) \tag{5.6}$$

Notaremos la descomposición por:

$$OR(\alpha, \{\alpha_1, \dots, \alpha_n\}) \tag{5.7}$$

Descomposición tipo XOR

Definición 5.11. Descomposición Exclusiva
Una descomposición es exclusiva (o de tipo XOR) si para que el objetivo de alto nivel (padre) sea satisfecho, es necesario que solamente uno de los objetivos de más bajo nivel (hijos) sea satisfecho. Es decir, sea α un objetivo de alto nivel, $C = \{\alpha_1 \dots \alpha_n\}$ los objetivos de bajo nivel en los que se descompone, y *sat* una función booleana que determina la satisfacción o no del objetivo:

$$sat(\alpha) \Leftrightarrow \exists \alpha_i \in C : sat(\alpha_i) \wedge \forall \alpha_j \in C : \alpha_i \neq \alpha_j \Rightarrow \neg sat(\alpha_j) \tag{5.8}$$

Notaremos la descomposición por:

$$XOR(\alpha\{\alpha_1, \dots, \alpha_n\}) \tag{5.9}$$

De esta forma, podemos construir un grafo con relaciones AND / OR / XOR, el Grafo de Interdependencia, que muestra las jerarquías de *goals* y *softgoals*, y cómo aquellos objetivos de más alto nivel pueden ser realizados a partir de la satisfacción de los objetivos de bajo nivel.

5.4 Análisis de Obstáculos

La naturaleza dinámica de los sistemas ubicuos está intimamente relacionada con la existencia de numerosas condiciones adversas que pueden dificultar la realización de los objetivos del sistema. Retomemos el ejemplo de un Sistema de Posicionamiento. En este tipo de sistemas hay varias unidades que transmiten señales, las cuales son medidas en otros dispositivos. Dichas mediciones son empleadas por un algoritmo de posicionamiento

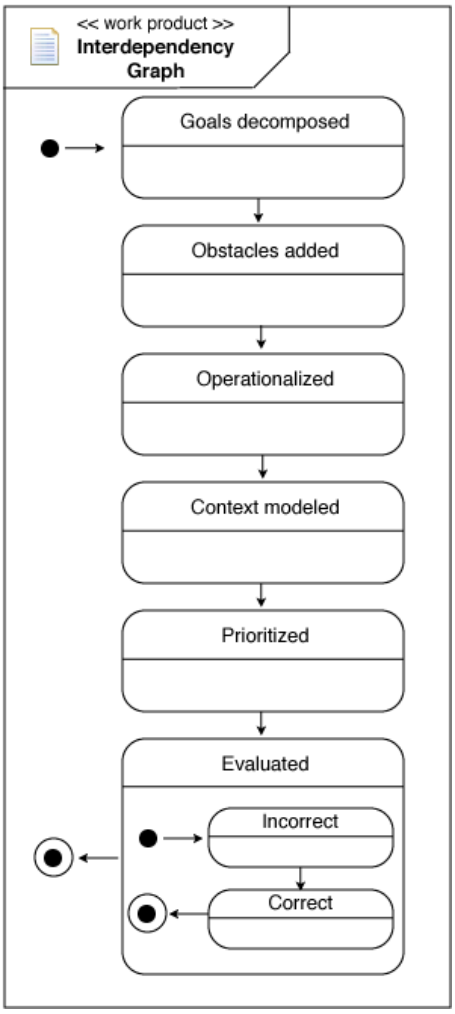

Figura 5.7: Diagrama de transición de estados que define los posibles estados de un Grafo de Interdependencia en notación SPEM (Véase leyenda en Tabla 4.3)

que realiza una estimación de la posición del dispositivo medidor de señal. Sin embargo, pueden ocurrir situaciones inesperadas: un transmisor de señal puede averiarse, puede que no todas las señales sean observables desde cualquier punto, etc.

Análisis de Obstáculos

Por lo tanto, en esta etapa se propone la realización de un **Análisis de Obstáculos**, con el objetivo de determinar las situaciones inconvenientes para la consecución de objetivos que pueden ocurrir con mayor probabilidad, a pesar de que la obtención de un conjunto completo de condiciones adversas puede ser difícil de conseguir. Definimos un **obstáculo** como:

Definición 5.12. Obstáculo
Un obstáculo es una condición, situación o evento en el entorno

del sistema que dificulta o impide la satisfacción de un objetivo. Notaremos esta relación por:

$$\alpha \in Objective, \Omega \in Obstacle : hinder(\Omega, \alpha) \tag{5.10}$$

Una vez determinados los obstáculos que entorpecen el funcionamiento del sistema ubicuo, es necesario encontrar soluciones que mantengan el correcto funcionamiento del sistema ante la aparición de dichos obstáculos. Es decir, para cada obstáculo que dificulta la consecución de un objetivo, debe existir al menos otro objetivo que lo mitigue:

$$\begin{aligned} \forall \alpha \in Objective, \Omega \in Obstacle : hinder(\Omega, \alpha) \Rightarrow \\ \exists \beta \in Objective : mitigate(\beta, \Omega) \end{aligned} \tag{5.11}$$

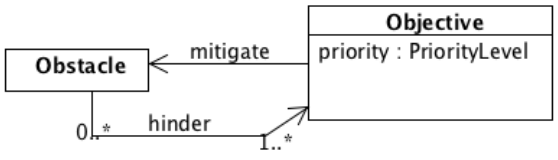

Figura 5.8: Metamodelo de obstáculos.

Heurísticas

De esta manera, se puede definir un conjunto de heurísticas para tratar de mitigar el efecto de los obstáculos en la consecución de los objetivos. Dichas heurísticas deben aplicarse de manera secuencial, y solamente si las anteriores no pueden resolver el problema.

Adición de sub-objetivos

Heurística 5.1. **Adición de sub-objetivos**
Si un obstáculo entorpece la satisfacción de un objetivo α, añadir un sub-objetivo α_i como hijo de α que pueda mitigar los efectos del obstáculo.

Sustitución de goals

Heurística 5.2. **Sustitución de *goals***
Si un obstáculo entorpece la satisfacción de un *goal*, sustituir el *goal* por un *softgoal* equivalente, permitiendo su satisfacción parcial, y determinar el nivel mínimo aceptable para su satisfacción.

Adición de objetivos

Heurística 5.3. **Adición de objetivos**
Si un obstáculo entorpece la satisfacción de un objetivo, aceptar

el posible fallo en la satisfacción del objetivo, e introducir un nuevo objetivo que pueda satisfacerse en caso de fracaso en la consecución del objetivo inicial.

La introducción de nuevos *goals* y *softgoals* implica la necesidad de llevar a cabo un proceso iterativo de refinamiento de objetivos y análisis de obstáculos que puede ser realizado de manera incremental, hasta que se hayan explorado aquellos escenarios indeseados que pueden ocurrir con una mayor probabilidad, y tengamos un conjunto completo de *goals* y *softgoals*.

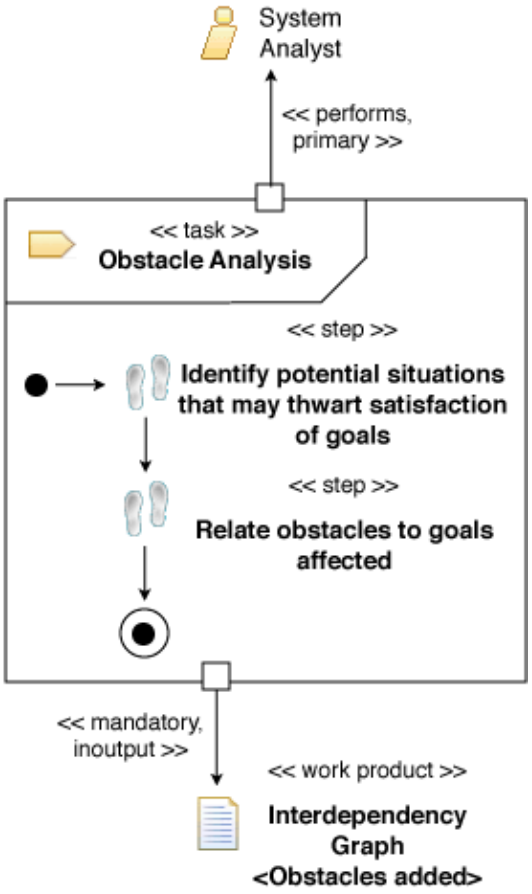

Figura 5.9: Definición de la tarea de Análisis de Obstáculos en notación SPEM (Véase leyenda en Tabla 4.3)

5.5 Intercambio de Recursos

Ordenación temporal de objetivos

En la sección 5.3 se han mostrado diferentes formas de descomponer un objetivo de alto nivel en otros de más bajo nivel.

Sin embargo, en ocasiones existen restricciones en el orden de ejecución de los sub-objetivos para la consecución del objetivo de más alto nivel. En efecto, existen objetivos que no pueden ser cumplidos hasta que otros no lo han sido, principalmente porque necesitan hacer uso de ciertos **recursos** que han sido generados como resultado de la consecución de otro/s objetivo/s.

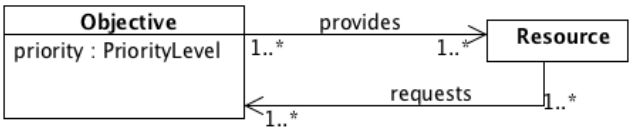

Figura 5.10: Metamodelo de recursos.

Definición 5.13. **Recurso**
Un recurso es un conjunto de datos que se origina por la realización de un objetivo, o que se requiere para la consecución de un objetivo.

Por tanto, existen dos relaciones fundamentales entre recursos y objetivos, la relación de provisión y la relación de demanda, las cuales notamos por:

Definición 5.14. **Relación de Provisión**
Relación provides
Una relación de provisión es una relación existente entre un recurso y un objetivo, de manera que el recurso se genera como resultado de la consecución del objetivo.

$$\alpha \in Objective, \rho \in Resources : provides(\alpha, \rho) \tag{5.12}$$

Definición 5.15. **Relación de Demanda**
Relación requests
Una relación de demanda es una relación existente entre un recurso y un objetivo, de manera que el recurso es necesario, pero no suficiente, para la consecución del objetivo.

$$\alpha \in Objective, \rho \in Resources : requests(\alpha, \rho) \tag{5.13}$$

5.6 Operacionalización

Operacionalización de objetivos

Una vez que el Grafo de Interdependencia ha sido obtenido, mostrando las relaciones entre *goals* y *softgoals*, junto con el impacto de los obstáculos sobre ellos, es necesario buscar alternativas que ayuden a satisfacer dichos objetivos; es decir, debemos proponer posibles **operacionalizaciones** y estudiar sus **contribuciones** al cumplimiento de los objetivos.

Definición 5.16. Operacionalización
Una operacionalización es una decisión que ayuda a cumplir uno o varios objetivos.

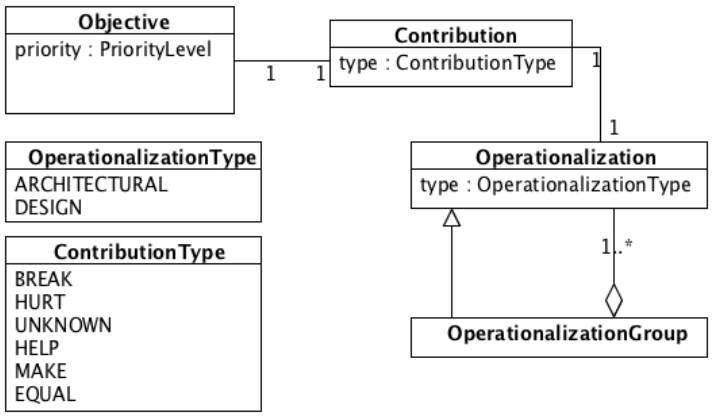

Figura 5.11: Metamodelo de operacionalizaciones.

Podemos clasificar las decisiones de operacionalización en dos categorías: operacionalizaciones **arquitectónicas** y operacionalizaciones de **diseño**.

Definición 5.17. Operacionalización Arquitectónica
Operacionalización Arquitectónica

Una operacionalización arquitectónica es una decisión que se refiere a la forma en la que el software está estructurado, los componentes que constituyen el sistema y los conectores entre ellos. López *et al.* [LA06] proponen organizar dichas decisiones en *architectural policies*, las cuales están ligadas a mecanismos arquitectónicos concretos que les dan soporte.

Definición 5.18. Operacionalización de Diseño
Operacionalización de Diseño

Una operacionalización de diseño es una definición detallada de los componentes y conectores software. Puede estar relacionada con los algoritmos empleados para realizar una tarea, la tecnología usada, la forma y medio en los que los datos son

presentados al usuario, los métodos de interacción propuestos para interactuar con el sistema, etc.

Adicionalmente, consideramos el estudio de la combinación de diferentes operacionalizaciones de manera conjunta, definiendo un **grupo de operacionalizaciones**.

Definición 5.19. **Grupo de Operacionalizaciones**
Un grupo de operacionalizaciones es un conjunto de decisiones distintas que sirven a un mismo propósito. Lo notamos por:

$$\gamma_1 \dots \gamma_n \in Operationalization, \Gamma \in Group : \\ group(\Gamma, \{\gamma_1 \dots \gamma_n\}) \tag{5.14}$$

Combinación de Operacionalizaciones

La combinación de operacionalizaciones tiene claras implicaciones en el diseño de software para sistemas ubicuos, ya que puede revelar nuevas contribuciones, tanto positivas como negativas, para el cumplimiento de los *goals* y *softgoals*.

Considerando el ejemplo del Sistema de Posicionamiento, podemos obtener el *Realizar una medición de la señal*, para el cual existen numerosas operacionalizaciones de diseño posibles: *Usar WiFi*, *Usar Bluetooth*, *Usar ZigBee* o *Usar RFID*, entre otras, pero además, usar varias de ellas y combinar sus mediciones. El uso de estas tecnologías de manera aislada puede presentar problemas si dicha tecnología no se encuentra disponible en el entorno en el que estamos, o bien, si algún transmisor de dicha señal se encuentra averiado. Sin embargo, su combinación es positiva para mantener la robustez del sistema, ya que no dependemos de una única tecnología, y también para la flexibilidad del sistema, ya que es capaz de emplear las tecnologías que se encuentren disponibles en el entorno de aplicación.

Contribuciones

Las decisiones de operacionalización tienen un impacto sobre los objetivos a los que dan soporte. Dicho impacto se modela mediante diferentes relaciones, llamadas contribuciones, que determinan la idoneidad de la operacionalización para la satisfacción del objetivo o no. Distinguimos cinco tipos de relaciones: **satisfacción**, **contribución positiva**, **contribución desconocida**, **contribución negativa** e **insatisfacción**.

Definición 5.20. **Relación de Satisfacción**

Relación make

Una relación de satisfacción entre una operacionalización y un objetivo implica que la elección de la operacionalización tiene

como resultado la consecución del objetivo asociado. Lo notamos como:

$$\alpha \in Objective, \gamma \in Operationalization : make(\gamma, \alpha) \quad (5.15)$$

Definición 5.21. **Relación de Contribución Positiva**
Una relación de contribución positiva entre una operacionalización y un objetivo implica que la elección de la operacionalización es prueba a favor de la satisfacción del objetivo, pero es necesaria una mayor evidencia positiva para concluir su satisfacción. Lo notamos por:

Relación help

$$\alpha \in Objective, \gamma \in Operationalization : help(\gamma, \alpha) \quad (5.16)$$

Definición 5.22. **Relación de Contribución Desconocida**
Una relación de contribución desconocida entre una operacionalización y un objetivo implica que la elección de la operacionalización tiene un impacto en la satisfacción del objetivo, pero no se conoce con certeza si es positiva o negativa. Lo notamos por:

Relación unknown

$$\alpha \in Objective, \gamma \in Operationalization : unknown(\gamma, \alpha) \quad (5.17)$$

Definición 5.23. **Relación de Contribución Negativa**
Una relación de contribución negativa entre una operacionalización y un objetivo implica que la elección de la operacionalización es prueba en contra de la satisfacción del objetivo, pero es necesaria una mayor evidencia negativa para concluir su insatisfacción. Lo notamos por:

Relación hurt

$$\alpha \in Objective, \gamma \in Operationalization : hurt(\gamma, \alpha) \quad (5.18)$$

Definición 5.24. **Relación de Insatisfacción**
Una relación de insatisfacción entre una operacionalización y un objetivo implica que la elección de la operacionalización resulta en la insatisfacción del objetivo asociado. Lo notamos por:

Relación break

$$\alpha \in Objective, \gamma \in Operationalization : break(\gamma, \alpha) \quad (5.19)$$

Adicionalmente, existe otra relación, la **relación de igualdad**, que puede ocurrir entre dos objetivos, o entre dos operacionalizaciones.

Igualdad entre Objetivos

Definición 5.25. **Relación de Igualdad entre Objetivos**
Una relación de igualdad entre objetivos implica que el grado de satisfacción de ambos es el mismo. Lo notamos por:

$$\alpha_1, \alpha_2 \in Objective : equal(\alpha_1, \alpha_2) \tag{5.20}$$

Igualdad entre Operacionalizaciones

Definición 5.26. **Relación de Igualdad entre Operacionalizaciones**
Una relación de igualdad entre operacionalizaciones implica que si una es aplicada, entonces la otra también debe aplicarse. Lo notamos por:

$$\gamma_1, \gamma_2 \in Operationalization : equal(\gamma_1, \gamma_2) \tag{5.21}$$

Mediante la representación de estas relaciones en el Grafo de Interdependencia, podemos encontrar los conflictos entre los diferentes *goals* y *softgoals*, así como las soluciones de compromiso (*trade-offs*) que puede ser necesario adoptar para así obtener un conjunto completo de objetivos satisfechos.

5.7 Justificación

Justificación de decisiones

Con objeto de proporcionar y recoger argumentación a las decisiones tomadas durante los pasos anteriores, REUBI proporciona el concepto de **justificación**.

Definición 5.27. **Justificación**
Una justificación es una razón que da soporte a las decisiones tomadas para descomponer u operacionalizar los *goals* y *softgoals* en un Grafo de Interdependencia.

Contribución de las justificaciones

Las justificaciones presentan relaciones positivas y negativas, llamadas relaciones de argumentación, con las relaciones de descomposición y operacionalización, de manera similar a lo

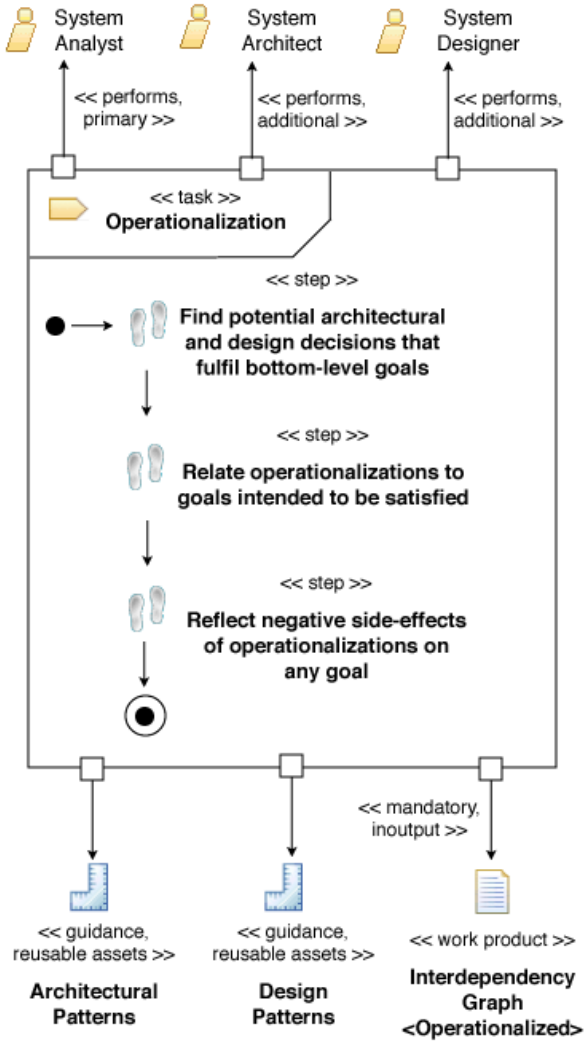

Figura 5.12: Definición de la tarea de Operacionalización en notación SPEM (Véase leyenda en Tabla 4.3)

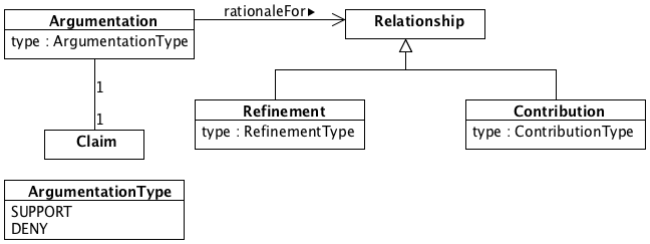

Figura 5.13: Metamodelo de justificaciones.

expuesto en la sección anterior. Así, distinguimos dos tipos de relaciones de justificación: de **soporte** y de **rechazo**.

Definición 5.28. Relación de Soporte
Una relación de soporte entre una justificación y una relación de descomposición o de operacionalización implica que ésta última es válida mientras la aseveración de la justificación sea cierta. Lo notamos por:

$$\lambda \in Claim, \sigma \in Relationship : support(\lambda, \sigma) \tag{5.22}$$

Definición 5.29. Relación de Rechazo
Una relación de rechazo entre una justificación y una relación de descomposición o de operacionalización implica que ésta última no es válida mientras la aseveración de la justificación sea cierta. Lo notamos por:

$$\lambda \in Claim, \sigma \in Relationship : deny(\lambda, \sigma) \tag{5.23}$$

Obtención de Justificaciones

Las relaciones de argumentación sirven para recopilar y justificar las decisiones que se toman durante el proceso de Ingeniería de Requisitos, así como pretenden potenciar la trazabilidad de las mismas. Dichas relaciones pueden ser obtenidas desde diferentes fuentes, tales como enfoques estándares, *frameworks*, autoridades, resultados de investigación o experiencias prácticas de implementación.

5.8 Modelado de Contexto

Consciencia del Contexto

Una de las características más relevantes de los Sistemas Ubicuos es la Consciencia del Contexto (*Context-awareness*). Dichos sistemas necesitan ser sensibles a los cambios que se producen en el contexto en el que se encuentran inmersos, con el propósito de adaptar su comportamiento para ajustarse a las necesidades y preferencias de los usuarios, y actuar de manera proactiva.

Notaciones para el contexto

Existen numerosos enfoques en la bibliografía que permiten modelar el contexto [BDR07], incluyendo notaciones gráficas, enfoques orientados a objetos o representaciones ontológicas. En esta etapa no es necesario contar con una representación del contexto tal y como se implementará en el sistema futuro. En

su lugar, necesitamos describir las situaciones contextuales que tienen un impacto en el Grafo de Interdependencia.

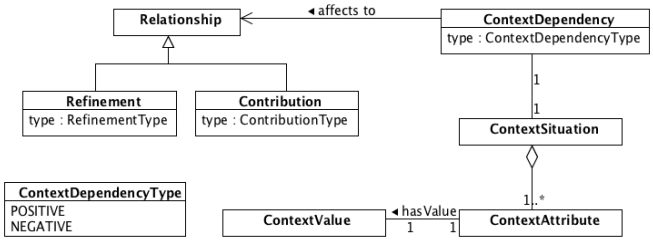

Figura 5.14: Metamodelo de contexto.

Conceptos para el Modelado de Contexto

Así, se propone una representación del contexto basada en tres conceptos fundamentales: **atributo de contexto**, **valor de contexto** y **situación de contexto**.

Definición 5.30. **Atributo de Contexto**
Un atributo de contexto es cualquier propiedad observable en el entorno del sistema que caracteriza una situación de interés para el mismo.

Definición 5.31. **Valor de Contexto**
Un valor de contexto es un posible valor para un atributo de contexto, sea puntual, un intervalo o un conjunto. Lo notamos por:

$$\phi \in ContextAttribute, v \in ContextValue : valueOf(\phi, v) \quad (5.24)$$

Definición 5.32. **Situación de Contexto**
Una situación de contexto es un conjunto de atributos de contexto que representan una situación de interés para el sistema y tiene un impacto en el Grafo de Interdependencia. Lo notamos por:

$$\Delta \in ContextSituation, \phi_1 \dots \phi_n \in ContextAttribute : situation(\Delta, \{\phi_1 \dots \phi_n\}) \quad (5.25)$$

Dependencias contextuales

Las situaciones de contexto poseen relaciones con las descomposiciones y operacionalizaciones vistas en las secciones 5.3 y 5.6. Para ello, definimos la relación de **dependencia de contexto**.

Definición 5.33. **Relación de Dependencia de Contexto**
Una relación de dependencia contextual entre una situación de

contexto, y una relación de descomposición u operacionalización implica que la ésta última es válida cuando la situación de contexto ocurre. Lo notamos por:

$$\Delta \in ContextSituation, \sigma \in Relationship : \\ contextDependency(\Delta, \sigma) \tag{5.26}$$

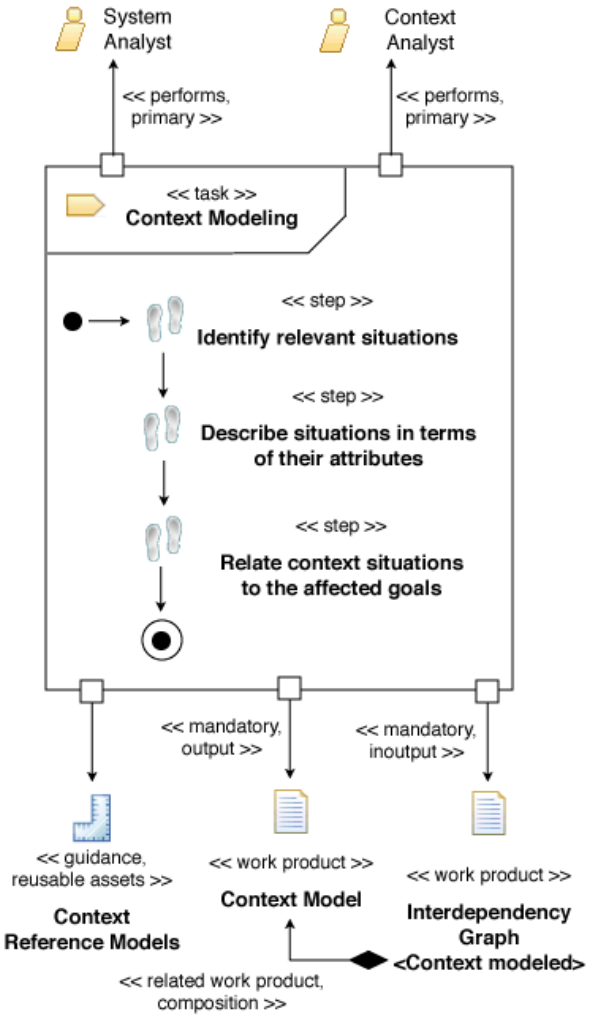

Figura 5.15: Definición de la tarea de Modelado de Contexto en notación SPEM (Véase leyenda en Tabla 4.3)

5.9 Priorización de Objetivos

Diferencias entre prioridades

No todos los objetivos deben ser abordados con el mismo esfuerzo; de hecho, hay algunos de ellos que son más importantes que otros, y en consecuencia deben ser **priorizados**. En un orden creciente de importancia, se distinguen tres niveles de prioridad: **normal**, **importante** y **crucial**. Estos últimos son vitales para el éxito del sistema, mientras que los primeros son prescindibles y habitualmente son sujeto de ser sacrificados o balanceados (*trade-offs*) si existen conflictos en su satisfacción.

Definición 5.34. **Priorización Crítica**
Prioridad Crítica

Un objetivo tiene prioridad crítica si su insatisfacción resulta en un fracaso para el sistema; es decir, su satisfacción es vital para el éxito del sistema. Lo notamos por:

$$\alpha \in Critical \subseteq Objective \tag{5.27}$$

Definición 5.35. **Priorización Importante**
Prioridad Importante

Un objetivo tiene prioridad importante si su satisfacción confiere un valor añadido al sistema. Lo notamos por:

$$\alpha \in Important \subseteq Objective \tag{5.28}$$

Definición 5.36. **Priorización Normal**
Prioridad Normal

Un objetivo tiene priorización normal si su satisfacción parcial no afecta de manera severa a la pérdida de calidad del sistema. Lo notamos por:

$$\alpha \in Normal \subseteq Objective \tag{5.29}$$

Sin embargo, a diferencia de otro tipo de sistemas, donde la prioridad en la satisfacción de los requisitos está fija durante su ejecución, los sistemas ubicuos se caracterizan por presentar cambios en la prioridad de satisfacción de requisitos, especialmente no funcionales, debido a cambios en el entorno del sistema.

Camios en la Priorización

Por ejemplo, es habitual que los sistemas ubicuos cuenten con un bajo tiempo de respuesta. Así, en el caso de un sistema de posicionamiento, ello implica la obtención de resultados menos

precisos, ya que un aumento de precisión involucra mayor cómputo, y por lo tanto, mayor tiempo de respuesta. No obstante, si ocurriese un accidente (por ejemplo, un incendio en un edificio), y existen personas que se encuentran inconscientes en su interior, es necesario localizarlas lo más precisamente posible. En este escenario, la precisión es crítica, incluso aunque involucre un mayor tiempo de respuesta. Por consiguiente, una reconfiguración del sistema es necesaria para afrontar el cambio de priorización de *goals* y *softgoals*.

Para modelar este cambio de prioridades dependiente del contexto en el Grafo de Interdependencia, debe hacerse uso de los conceptos definidos en la sección 5.8, concretamente de las situaciones de contexto y las dependencias de contexto. Las situaciones de contexto se relacionan con las relaciones de descomposición u operacionalización, a las que afectan por medio de dependencias de contexto, como se indica en la sección anterior.

Figura 5.16: Definición de la tarea de Priorización en notación SPEM (Véase leyenda en Tabla 4.3)

5.10 Procedimiento de Evaluación

Una vez que se ha modelado el Grafo de Interdependencia, se debe proceder a su evaluación para determinar si los *goals* y *softgoals* que contiene están satisfechos dadas las operacionaliza-

ciones que se proporcionan para los mismos. El procedimiento de evaluación tiene sus bases en el procedimiento expuesto en [CNYMoo], con ligeras variaciones para adaptarlo al modelo propuesto en REUBI, tal como se describe a continuación.

El Sistema Ubicuo es dinámico y su arquitectura está formada por componentes cuyo uso y distribución podrán variar en tiempo de ejecución dependiendo del contexto, del usuario, de la ocurrencia de situaciones anómalas y de cambios en la prioridad de satisfacción de los requisitos. Por todo ello, se necesita un procedimiento de evaluación que automatice y guíe al ingeniero para tomar la mejor decisión posible para que se cumplan los objetivos teniendo en cuenta todas las circunstancias anteriores.

5.10.1 Reglas de Evaluación

Con objeto de simplificar la explicación del procedimiento de evaluación, se define un conjunto de reglas de evaluación que determinan la satisfacción de cada uno de los objetivos. El procedimiento de evaluación toma como entrada un conjunto de situaciones de contexto, Δ, un conjunto de justificaciones, Λ, un conjunto de objetivos, α, y un conjunto de operacionalizaciones, Γ. Definimos el conjunto de **operacionalizaciones elegidas** como un subconjunto de las operacionalizaciones, $\Gamma' \subseteq \Gamma$, que serán las únicas tenidas en cuenta para la evaluación de los objetivos, asumiendo que solamente dichas operacionalizaciones serán incluidas en el sistema futuro. La salida del procedimiento será, para cada situación de contexto en Δ, las operacionalizaciones que satisfacen los objetivos propuestos.

Existen 10 reglas de evaluación que se indican a continuación.

Regla de Evaluación 5.1. Contribución de Satisfacción Única
Si un objetivo, $\alpha_i \in \alpha$, recibe al menos una contribución de satisfacción y no recibe ninguna contribución negativa o desconocida, entonces se encuentra satisfecho:

Contribución de Satisfacción Única

$$\begin{aligned} \exists \gamma_k \in \Gamma' \; make(\gamma_k, \alpha_i) \wedge \neg \exists \gamma_m \in \Gamma' \; (unknown(\gamma_m, \alpha_i) \vee \\ hurt(\gamma_m, \alpha_i) \vee break(\gamma_m, \alpha_i)) \Rightarrow \\ sat(\alpha_i) = SATISFIED \end{aligned} \tag{5.30}$$

Regla de Evaluación 5.2. Contribución de Insatisfacción Única
Si un objetivo, $\alpha_i \in \alpha$, recibe al menos una contribución de insatisfacción y no recibe ninguna contribución positiva o desconocida, entonces se encuentra insatisfecho:

Contribución de Insatisfacción Única

$$\exists \gamma_k \in \Gamma' \; break(\gamma_k, \alpha_i) \wedge \neg \exists \gamma_m \in \Gamma' \; (unknown(\gamma_m, \alpha_i) \vee help(\gamma_m, \alpha_i) \vee make(\gamma_m, \alpha_i)) \Rightarrow sat(\alpha_i) = UNSATISFIED \quad (5.31)$$

Contribución Desconocida

Regla de Evaluación 5.3. Contribución Desconocida
Si un objetivo, $\alpha_i \in \alpha$, recibe una contribución desconocida por parte de una operacionalización, entonces su satisfacción es desconocida:

$$\exists \gamma_k \in \Gamma' \; unknown(\gamma_k, \alpha_i) \Rightarrow sat(\alpha_i) = UNKNOWN \quad (5.32)$$

Contribución Contradictoria

Regla de Evaluación 5.4. Contribución Contradictoria
Si un objetivo, $\alpha_i \in \alpha$, recibe una contribución positiva o de satisfacción, y una contribución negativa o de insatisfacción, entonces su satisfacción es desconocida:

$$\exists \gamma_k \in \Gamma' \; (make(\gamma_k, \alpha_i) \vee help(\gamma_k, \alpha_i)) \wedge \exists \gamma_m \in \Gamma' \; (break(\gamma_m, \alpha_i) \vee hurt(\gamma_m, \alpha_i)) \Rightarrow sat(\alpha_i) = UNKNOWN \quad (5.33)$$

Contribución de Satisfacción Parcial

Regla de Evaluación 5.5. Contribución de Satisfacción Parcial Única
Si un objetivo, $\alpha_i \in \alpha$, recibe una contribución positiva, y no recibe ninguna contribución negativa, de insatisfacción o desconocida, entonces se encuentra parcialmente satisfecho:

$$\exists \gamma_k \in \Gamma' \; help(\gamma_k, \alpha_i) \wedge \neg \exists \gamma_m \in \Gamma' \; (hurt(\gamma_m, \alpha_i) \vee break(\gamma_m, \alpha_i) \vee unknown(\gamma_m, \alpha_i)) \Rightarrow sat(\alpha_i) = PARTIALLY\ SATISFIED \quad (5.34)$$

Contribución de Insatisfacción Parcial

Regla de Evaluación 5.6. Contribución de Insatisfacción Parcial Única
Si un objetivo, $\alpha_i \in \alpha$, recibe una contribución negativa, y no reci-

be ninguna contribución positiva, de satisfacción o desconocida, entonces se encuentra parcialmente insatisfecho:

$$\exists \gamma_k \in \Gamma' \; hurt(\gamma_k, \alpha_i) \wedge \neg \exists \gamma_m \in \Gamma' \; (help(\gamma_m, \alpha_i) \vee make(\gamma_m, \alpha_i) \vee unknown(\gamma_m, \alpha_i)) \Rightarrow sat(\alpha_i) = PARTIALLY\ INSATISFIED \tag{5.35}$$

Regla de Evaluación 5.7. Igualdad de satisfacción

Igualdad de Satisfacción

Si un objetivo, $\alpha_i \in \alpha$, ha sido evaluado, propagar su nivel de satisfacción a todos aquellos objetivos con los que tiene una relación de igualdad:

$$\alpha_i \in \alpha : \forall \alpha_k \in \alpha \backslash \{\alpha_i\} \; equals(\alpha_i, \alpha_k) \Rightarrow sat(\alpha_k) = sat(\alpha_i) \tag{5.36}$$

Regla de Evaluación 5.8. Evaluación de la Descomposición Inclusiva

Evaluación de la Descomposición Inclusiva

Sea $\{\beta_1 \dots \beta_n\} \subseteq \alpha$, un subconjunto de subobjetivos que refinan de manera inclusiva a un objetivo, $\alpha_i \in \alpha$.

Si existe un subobjetivo, β_k, que está insatisfecho, entonces el objetivo de alto nivel, α, está insatisfecho.

$$AND(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha \; sat(\beta_k) = UNSATISFIED \Rightarrow sat(\alpha_i) = UNSATISFIED \tag{5.37}$$

Si no, si existe un subobjetivo que está parcialmente satisfecho, parcialmente insatisfecho o cuya satisfacción se desconoce, entonces la satisfacción del objetivo de alto nivel es desconocida:

$$AND(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha \; (sat(\beta_k) = UNKNOWN \vee sat(\beta_k) = PARTIALLY\ SATISFIED \vee sat(\beta_k) = PARTIALLY\ UNSATISFIED) \Rightarrow sat(\alpha_i) = UNKNOWN \tag{5.38}$$

En cualquier otro caso, el objetivo está satisfecho:

$$AND(\alpha_i, \{\beta_1 \dots \beta_n\}) : \forall \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha$$
$$sat(\beta_k) = SATISFIED \Rightarrow sat(\alpha_i) = SATISFIED \quad (5.39)$$

Regla de Evaluación 5.9. Evaluación de la Descomposición Alternativa

Evaluación de la Descomposición Alternativa

Sea $\{\beta_1 \dots \beta_n\} \subseteq \alpha$, un subconjunto de objetivos que refinan de manera alternativa a un objetivo, $\alpha_i \in \alpha$.

Si existe un subobjetivo que está satisfecho, entonces el objetivo de alto nivel está satisfecho:

$$OR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha$$
$$sat(\beta_k) = SATISFIED \Rightarrow sat(\alpha_i) = SATISFIED \quad (5.40)$$

Si no, si un subobjetivo está parcialmente satisfecho, parcialmente insatisfecho o cuya satisfacción es desconocida, entonces la satisfacción del objetivo de alto nivel es desconocida:

$$\begin{gathered} OR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha \\ (sat(\beta_k) = UNKNOWN \vee \\ sat(\beta_k) = PARTIALLY\ SATISFIED \vee \\ sat(\beta_k) = PARTIALLY\ UNSATISFIED) \Rightarrow \\ sat(\alpha_i) = UNKNOWN \end{gathered} \quad (5.41)$$

En cualquier otro caso, el objetivo está insatisfecho:

$$\begin{gathered} OR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \forall \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha \\ sat(\beta_k) = UNSATISFIED \Rightarrow \\ sat(\alpha_i) = UNSATISFIED \end{gathered} \quad (5.42)$$

Regla de Evaluación 5.10. Evaluación de la Descomposición Exclusiva

Evaluación de la Descomposición Exclusiva

Sea $\{\beta_1 \dots \beta_n\} \subseteq \alpha$, un subconjunto de objetivos que refinan de manera exclusiva a un objetivo, $\alpha_i \in \alpha$.

Si existe un subobjetivo que está satisfecho y el resto de objetivos están insatisfechos, entonces el objetivo de más alto nivel está satisfecho:

$$XOR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha$$
$$sat(\beta_k) = SATISFIED \wedge$$
$$\forall \beta_m \in \{\beta_1 \dots \beta_n\} \backslash \{\beta_k\}\ sat(\beta_m) = UNSATISFIED \Rightarrow$$
$$sat(\alpha_i) = SATISFIED \tag{5.43}$$

Si no, si existe un objetivo que está parcialmente satisfecho, parcialmente insatisfecho o cuya satisfacción es desconocida, y el resto de objetivos están insatisfechos, entonces la satisfacción del objetivo de más alto nivel es desconocida:

$$XOR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k \in \{\beta_1 \dots \beta_n\} \subseteq \alpha$$
$$(sat(\beta_k) = PARTIALLY\ SATISFIED \vee$$
$$sat(\beta_k) = PARTIALLY\ UNSATISFIED \vee$$
$$sat(\beta_k) = UNKNOWN) \wedge$$
$$\forall \beta_m \in \{\beta_1 \dots \beta_n\} \backslash \{\beta_k\}\ sat(\beta_m) = UNSATISFIED \Rightarrow$$
$$sat(\alpha_i) = UNKNOWN \tag{5.44}$$

En cualquier otro caso, el objetivo de más alto nivel está insatisfecho.

$$XOR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \exists \beta_k, \beta_m \in \{\beta_1 \dots \beta_n\} \subseteq \alpha$$
$$sat(\beta_k) = sat(\beta_m) \wedge sat(\beta_k) \neq UNSATISFIED \wedge$$
$$sat(\beta_m) \neq UNSATISFIED \wedge \beta_k \neq \beta_m \Rightarrow$$
$$sat(\alpha_i) = UNSATISFIED \tag{5.45}$$

$$XOR(\alpha_i, \{\beta_1 \dots \beta_n\}) : \forall \beta_k \in \{\beta_1 \dots \beta_m\} \subseteq \alpha$$
$$sat(\beta_k) = UNSATISFIED \Rightarrow$$
$$sat(\alpha_i) = UNSATISFIED \tag{5.46}$$

5.10.2 Algoritmo de Evaluación

Una vez definidas las reglas de evaluación necesarias para el procedimiento de evaluación, definimos el algoritmo que determina la satisfacción de los objetivos para cada una de las situaciones de contexto que se hayan definido.

Entrada del algoritmo

El Algoritmo 1 toma como entrada un conjunto de objetivos

que deben satisfacerse, α; un conjunto de operacionalizaciones, Γ, que han sido definidas en el Grafo de Interdependencia y que contribuyen de diferentes maneras a la satisfacción de los objetivos; un conjunto de Justificaciones, Λ, y un conjunto de Situaciones de Contexto, Δ, los cuales determinan las relaciones de descomposición y operacionalización que son válidas en cada momento.

Procedimiento

El algoritmo trata de encontrar el conjunto de operacionalizaciones que satisface todos los objetivos, dada una determinada situación de contexto. Para ello, se deben examinar todas las situaciones de contexto. Dada una de ellas, se pide al Ingeniero de Requisitos que determine un subconjunto de las operacionalizaciones, $\Gamma'_0 \subseteq \Gamma$, que sirve para realizar una primera evaluación de la satisfacción de los objetivos.

A continuación, para cada uno de los objetivos, se evalúa su satisfacción teniendo en cuenta el subconjunto de operacionalizaciones y las 10 reglas de evaluación definidas en la subsección anterior. Si existe alguno de los objetivos cuyo grado de satisfacción no esté satisfecho completamente, se pide al Ingeniero de Requisitos que actualice el subconjunto de operacionalizaciones, $\Gamma'_{n+1} \subseteq \Gamma$, añadiendo o quitando operacionalizaciones de forma que se resuelva la situación de no satisfacción. Adicionalmente, se pide al Ingeniero de Requisitos que revise y complemente la evaluación realizada de manera automática mediante la evaluación de reglas. Esto es particularmente útil en casos donde se reciben diferentes contribuciones que satisfacen de manera parcial a un objetivo; en estos casos, las reglas de evaluación estimarán que el grado de satisfacción del objetivo será parcialmente satisfecho. El Ingeniero de Requisitos en este caso puede manualmente determinar que dicho objetivo se ha satisfecho. De manera similar se puede proceder cuando se detecta una insatisfacción parcial o un desconocimiento de la satisfacción.

Salida del algoritmo

El proceso anterior se repite de manera iterativa hasta que entre dos iteraciones no se produce ningún cambio en el grado de satisfacción entre los objetivos. Posteriormente, se debe determinar si el procedimiento de evaluación ha sido exitoso o no. Así, si existe algún objetivo, cuya prioridad es crítica para la situación de contexto, que no ha sido satisfecho, dicha situación no ha podido resolverse de manera satisfactoria con el Grafo de Interdependencia actual, y por tanto requiere un mayor trabajo del mismo. Si existe algún objetivo, cuya prioridad es importante, que no ha sido satisfecho, se informa al Ingeniero de Requisitos mediante un aviso, para que considere la posibilidad de realizar mejoras en el Grafo de Interdependencia para tratar de resolver este hecho. Por último, si no se da ninguna de las anteriores

circunstancias, se proporciona como salida el conjunto de operacionalizaciones que satisfacen los objetivos para la situación de contexto que se está considerando.

Algoritmo 1 Procedimiento de Evaluación de Objetivos

Require: Δ : {ContextSituation}
Require: Λ : {Claim}
Require: Γ : {Operationalization}
Require: α : {Objective}
1: **for all** $\Delta_i \in \Delta$ **do**
2: $\Gamma'_0 \leftarrow$ input from Req. Engineer
3: **repeat**
4: **for all** $\alpha_i \in \alpha$ **do**
5: update sat(α_i) according to evaluation rules 4.1 to 4.10
6: **end for**
7: **if** $\exists \alpha_i \in \alpha \; sat(\alpha_i) \neq SATISFIED$ **then**
8: $\Gamma'_{n+1} \leftarrow$ input from Req. Engineer
9: **end if**
10: ask Req. Engineer to check/correct satisfaction inferences
11: **until** no changes in satisfaction of α_i between iterations
12: **if** $\exists \alpha_k \in \alpha \; sat(\alpha_k) \neq SATISFIED \wedge \alpha_k \in Critical$ **then**
13: $output(\Delta_i, failure, \Gamma_n)$
14: **else if** $\exists \alpha_k \in \alpha \; sat(\alpha_k) \neq SATISFIED \wedge \alpha_k \in Important$ **then**
15: $output(\Delta_i, warning, \Gamma_n)$
16: **else**
17: $output(\Delta_i, success, \Gamma_n)$
18: **end if**
19: **end for**

En caso de producirse algún fallo o aviso en la evaluación de los objetivos para alguna de las situaciones de contexto, el Ingeniero de Requisitos debe realizar una iteración sobre las actividades propuestas a lo largo de este capítulo, con el objeto de conseguir satisfacer todos los objetivos para cada una de las situaciones de interés.

La salida del algoritmo relaciona las situaciones de contexto obtenidas con las operacionalizaciones que satisfacen los objetivos que deben cumplirse en cada una de ellas. Dicha información es proporcionada a los Ingenieros de Software, quienes se deben encargar de extraer las reglas de adaptación del sistema en cada una de las situaciones propuestas.

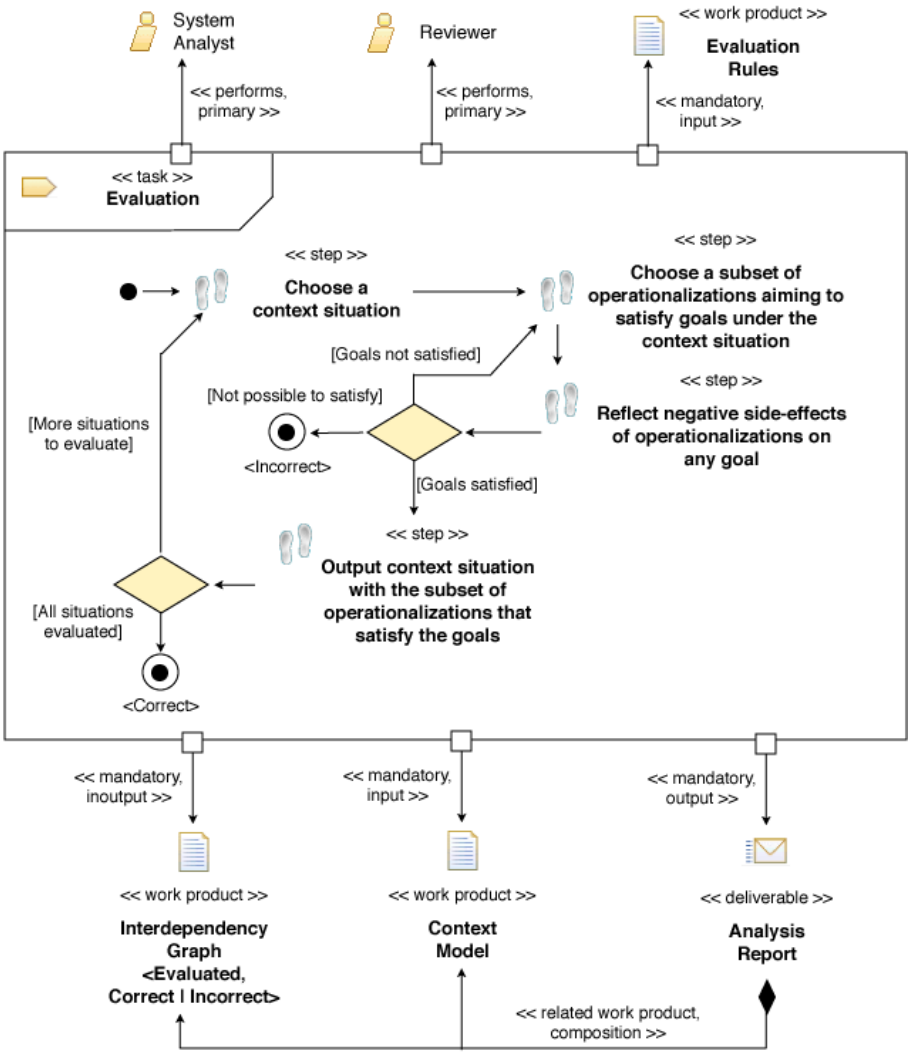

Figura 5.17: Definición de la tarea de Evaluación en notación SPEM (Véase leyenda en Tabla 4.3)

5.10.3 Heurísticas de Selección

En el Procedimiento de Evaluación expuesto anteriormente existe una etapa en la que se pide al Ingeniero de Requisitos realizar una selección de operacionalizaciones dirigidas a la satisfacción de los objetivos. Dicha selección puede ser un proceso costoso y difícil, ya que el proceso que debe realizarse es una optimización multiobjetivo.

Para tratar de paliar este efecto, y con objeto de dirigir al Ingeniero de Requisitos hacia buenas soluciones, en esta sección se presentan un conjunto de heurísticas que permitan realizar de ma-

nera exitosa la selección de operacionalizaciones que impliquen la satisfacción del mayor número posible de operacionalizaciones.

Heurística 5.4. Ordenación prioritaria
Ordenación prioritaria
Los objetivos deben ordenarse por prioridades, atendiendo en primer lugar a aquéllos cuya prioridad es crítica, seguidos de aquéllos cuya prioridad es importante, y, finalmente, aquéllos cuya prioridad es normal.

Heurística 5.5. Ordenación por profundidad
Ordenación por profundidad
Los objetivos deben ordenarse según su profundidad en el Grafo de Interdependencia, tratando de satisfacer en primer lugar aquéllos de más bajo nivel y, posteriormente, los de más alto nivel.

Heurística 5.6. Maximización de contribución positiva
Maximización de contribución positiva
Si una operacionalización contribuye positivamente a la satisfacción de varios objetivos, debe ser seleccionada.

Heurística 5.7. Minimización de contribución negativa
Minimización de contribución negativa
Si una operacionalización contribuye negativamente a la satisfacción de varios objetivos, debe ser rechazada.

Heurística 5.8. Minimización de contribución desconocida
Minimización de contribución desconocida
Si una operacionalización contribuye de manera desconocida a la satisfacción de varios objetivos, debe ser rechazada.

Heurística 5.9. Elección de mejor contribución de satisfacción
Elección de mejor contribución de satisfacción
Si un objetivo recibe varias contribuciones de satisfacción, éstas deben ordenarse teniendo en cuenta su idoneidad, y seleccionar tantas como sea posible sin que se afecte a la satisfacción de otros objetivos.

Heurística 5.10. Elección de múltiples contribuciones positivas
Elección de múltiples contribuciones positivas
Si un objetivo recibe varias contribuciones positivas, pero ninguna de ellas consigue satisfacer el objetivo de manera aislada, deben ordenarse teniendo en cuenta su idoneidad, y seleccionar tantas como sea posible sin que se afecte a la satisfacción de otros objetivos.

Heurística 5.11. Resolución de Conflictos
Resolución de Conflictos
Si un objetivo tiene un grado de satisfacción desconocido debido a que recibe contribuciones tanto positivas como negativas:

- Si las contribuciones positivas son superiores a las negativas, cambiar la satisfacción del objetivo a parcialmente satisfecho, o completamente satisfecho, según proceda.

- Si las contribuciones negativas son superiores a las positivas, cambiar la satisfacción del objetivo a parcialmente insatisfecho, o completamente insatisfecho, según proceda. Añadir operacionalizaciones que contribuyan positivamente a la satisfacción del objetivo, o eliminar operacionalizaciones que contribuyan negativamente a la satisfacción del objetivo.

Heurística 5.12. Satisfacción Parcial de Objetivos

Satisfacción Parcial de Objetivos

Si un objetivo está parcialmente satisfecho y recibe múltiples contribuciones positivas, pero ninguna de ellas consigue satisfacer el objetivo de manera aislada, el objetivo puede considerarse completamente satisfecho.

Heurística 5.13. Insatisfacción Parcial de Objetivos

Insatisfacción Parcial de Objetivos

Si un objetivo está parcialmente insatisfecho y recibe múltiples contribuciones negativas, pero ninguna de ellas consigue que el objetivo esté completamente insatisfecho de manera aislada, el objetivo puede considerarse completamente insatisfecho.

5.10.4 Matrices de contribución y refinamiento

Una forma alternativa de calcular la satisfacción de los objetivos consiste en representar matricialmente las contribuciones de las operacionalizaciones a los objetivos, y los refinamientos de estos en otros de más bajo nivel. Para ello, definimos los siguientes conceptos:

Definición 5.37. Matriz de contribución

Matriz de contribución

Sea $\mathcal{O} = \{\nu_1 \dots \nu_n\}$ el conjunto de operacionalizaciones, y $\mathcal{A} = \{\alpha_1 \dots \alpha_m\}$ el conjunto de objetivos que reciben una contribución directa de una operacionalización. Una matriz de contribución es una matriz $\mathcal{M}_{n,m}$ en la que cada componente (i, j) indica la contribución de la operacionalización ν_i a la satisfacción del objetivo α_j.

Dominio de la matriz de contribución

La matriz de contribución toma valores en el conjunto $\{\bigtriangledown, -, ?, +, \bigtriangleup, \emptyset\}$, que representan respectivamente las contribuciones de insatisfacción, negativa, desconocida, positiva, de satisfacción, y ninguna contribución.

Definición 5.38. Vector de selección

Vector de selección

Sea $\mathcal{O} = \{\nu_1 \dots \nu_n\}$ el conjunto de operacionalizaciones. Un vector de selección es una matriz $\mathcal{S}_{1,n}$ en la que cada componente $(1, i)$ indica si la operacionalización ν_i debe ser considerada o no. El vector de selección toma valores en el conjunto $\{\checkmark, \times\}$, que

representan respectivamente la elección o no de la operacionalización v_i.

Definición 5.39. Operación de selección
Sean los conjuntos $\mathcal{E} = \{\checkmark, \times\}$ y $\mathcal{C} = \{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$. Definimos el operador de selección, $\odot$, como un operador binario cuyo dominio es $(\mathcal{E}, \mathcal{C})$ y su recorrido es $\mathcal{C}$. La semántica del operador se encuentra en la Tabla 5.1.

Operador de selección

$\odot$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$
$\checkmark$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$
$\times$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Tabla 5.1: Operador de Selección

Definición 5.40. Operador de contribución
Sea el conjunto $\mathcal{C} = \{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$. Se define el operador de contribución, $\oplus$, como un operador binario cuyo dominio es $(\mathcal{C}, \mathcal{C})$ y cuyo recorrido es $\mathcal{C}$. La semántica del operador se encuentra en la Tabla 5.2.

Operador de contribución

$\oplus$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$
$\bigtriangledown$	$\bigtriangledown$	$\bigtriangledown$	?	-	?	$\bigtriangledown$
-	$\bigtriangledown$	$\bigtriangledown$	?	?	+	-
?	?	?	?	?	?	?
+	-	?	?	$\bigtriangleup$	$\bigtriangleup$	+
$\bigtriangleup$	?	+	?	$\bigtriangleup$	$\bigtriangleup$	$\bigtriangleup$
$\varnothing$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$

Tabla 5.2: Operador de Contribución

Una vez definidos estos conceptos, podemos calcular la satisfacción de los objetivos que reciben contribución directa de la siguiente manera:

Definición 5.41. Vector de satisfacción
Sea $\mathcal{M}_{n,m}$ la matriz de contribución de n operacionalizaciones a m objetivos, y $\mathcal{O}_{1,n}$ el vector de selección de las operacionalizaciones.

Vector de satisfacción

Se define el vector de satisfacción, $\mathcal{S}_{1,m}$, como el resultado de la operación:

$$\mathcal{S}_{1,m} = \mathcal{O}_{1,n} \bigodot \mathcal{M}_{n,m} \tag{5.47}$$

Donde cada elemento de $\mathcal{S}$ se calcula como:

$$\mathcal{S}_{(1,i)} = \bigoplus_{k=1}^{n} (\mathcal{O}_{(1,k)} \odot \mathcal{M}_{(k,i)}) \tag{5.48}$$

Dominio del vector de satisfacción

El vector de satisfacción toma valores en el conjunto $\{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$, donde la posición i-ésima representa la satisfacción del objetivo α_i.

Satisfacción inicial

De esta forma se consigue una estimación inicial de la satisfacción de aquellos objetivos que reciben una contribución directa por parte de una o varias operacionalizaciones. El vector de satisfacción debe ser analizado por parte del Ingeniero de Requisitos, ya que debe asegurarse de que, antes de proceder a la segunda etapa, no existen objetivos que no se encuentren satisfechos o parcialmente satisfechos. En caso contrario, debe repetirse el cálculo tomando un nuevo vector de selección.

A continuación, debe realizarse el cálculo de la satisfacción del resto de los objetivos (aquéllos que no reciben contribución directa por parte de las operacionalizaciones, sino que son refinados por objetivos de más bajo nivel). Para ello, se definen los siguientes conceptos:

Definición 5.42. **Matriz de Refinamiento**

Matriz de Refinamiento

Sea $\mathcal{A} = \{\alpha_1 \dots \alpha_n\}$ el conjunto de objetivos que reciben una contribución directa de una o varias operacionalizaciones, y $\mathcal{B} = \{\alpha_1 \dots \alpha_p\}, p \geq n$, el conjunto de todos los objetivos. Se define una matriz de refinamiento, $\mathcal{R}_p$, como una matriz cuadrada de orden p en la que la posición (i, j) representa un refinamiento del objetivo i en el objetivo j. La matriz de refinamiento toma valores en el conjunto $\{\checkmark, \times\}$, que representan, respectivamente, la presencia o ausencia de refinamiento entre ambos objetivos.

Definición 5.43. **Matriz de Refinamiento Inclusivo**

Matriz de Refinamiento Inclusivo

La matriz de refinamiento inclusivo, $\mathcal{R}_p^{AND}$, es una matriz de refinamiento que representa la presencia de refinamientos de tipo inclusivo entre objetivos.

Definición 5.44. **Matriz de Refinamiento Alternativo**

Matriz de Refinamiento Alternativo

La matriz de refinamiento alternativo, $\mathcal{R}_p^{OR}$, es una matriz de

refinamiento que representa la presencia de refinamientos de tipo alternativo entre objetivos.

Definición 5.45. Matriz de Refinamiento Exclusivo
La matriz de refinamiento exclusivo, $\mathcal{R}_p^{XOR}$, es una matriz de refinamiento que representa la presencia de refinamientos de tipo exclusivo entre objetivos.

Matriz de Refinamiento Exclusivo

Definición 5.46. Operador de Refinamiento Inclusivo
Sea el conjunto $\mathcal{C} = \{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$, se define el operador de refinamiento inclusivo, $\wedge$, como un operador binario cuyo dominio es $(\mathcal{C}, \mathcal{C})$ y su recorrido es $\mathcal{C}$. La semántica del operador se encuentra en la Tabla 5.3.

Operador de Refinamiento Inclusivo

$\wedge$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$
$\bigtriangledown$	$\bigtriangledown$	$\bigtriangledown$	?	$\bigtriangledown$	$\bigtriangledown$	$\bigtriangledown$
-	$\bigtriangledown$	-	?	-	-	-
?	?	?	?	?	?	?
+	$\bigtriangledown$	-	?	+	$\bigtriangleup$	+
$\bigtriangleup$	$\bigtriangledown$	-	?	$\bigtriangleup$	$\bigtriangleup$	$\bigtriangleup$
$\varnothing$	$\bigtriangledown$	-	?	+	$\bigtriangleup$	$\varnothing$

Tabla 5.3: Operador de Refinamiento Inclusivo

El operador de refinamiento inclusivo es un operador que posee las propiedades conmutativa y asociativa.

Definición 5.47. Operador de Refinamiento Alternativo
Sea el conjunto $\mathcal{C} = \{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$, se define el operador de refinamiento alternativo, $\vee$, como un operador binario cuyo dominio es $(\mathcal{C}, \mathcal{C})$ y su recorrido es $\mathcal{C}$. La semántica del operador se encuentra en la Tabla 5.4.

Operador de Refinamiento Alternativo

El operador de refinamiento alternativo es un operador que posee las propiedades conmutativa y asociativa.

Definición 5.48. Operador de Refinamiento Exclusivo
Sea el conjunto $\mathcal{C} = \{\bigtriangledown, -, ?, +, \bigtriangleup, \varnothing\}$, se define el operador de refinamiento exclusivo, $\bowtie$, como un operador binario cuyo dominio es $(\mathcal{C}, \mathcal{C})$ y su recorrido es $\mathcal{C}$. La semántica del operador se encuentra en la Tabla 5.5.

Operador de Refinamiento Exclusivo

El operador de refinamiento alternativo es conmutativo, pero

Definición recursiva del operador de refinamiento alternativo

$\vee$	∇	-	?	+	$\triangle$	$\emptyset$
∇	∇	-	?	+	$\triangle$	∇
-	-	-	?	+	$\triangle$	-
?	?	?	?	+	$\triangle$	?
+	+	+	+	+	$\triangle$	+
$\triangle$	$\triangle$	$\triangle$	$\triangle$	$\triangle$	$\triangle$	$\triangle$
$\emptyset$	∇	-	?	+	$\triangle$	$\emptyset$

Tabla 5.4: Operador de Refinamiento Alternativo

$\bowtie$	∇	-	?	+	$\triangle$	$\emptyset$
∇	∇	∇	?	+	$\triangle$	∇
-	∇	-	?	+	$\triangle$	-
?	?	?	?	?	?	?
+	+	+	?	∇	∇	+
$\triangle$	$\triangle$	$\triangle$	?	∇	∇	$\triangle$
$\emptyset$	∇	-	?	+	$\triangle$	$\emptyset$

Tabla 5.5: Operador de Refinamiento Exclusivo

no asociativo. De esta forma, cuando se debe aplicar el operador a un conjunto con un número mayor a dos elementos $C = \{\alpha_0 \dots \alpha_n\}$, la operación debe realizarse de manera recursiva como:

$$\begin{aligned} \bowtie_{k=0}^{n} \alpha_k &= \alpha_0 \bowtie \alpha_1 \bowtie \alpha_2 \dots \bowtie \alpha_n \\ &= (\alpha_0 \bowtie \alpha_1) \bowtie (\alpha_1 \bowtie \alpha_2) \bowtie \\ &\quad (\alpha_2 \bowtie \alpha_3) \bowtie \dots \bowtie (\alpha_{n-1} \bowtie \alpha_n) \end{aligned} \tag{5.49}$$

Satisfacción general

Una vez definidos estos operadores, el vector de satisfacción general se calcula como:

$$\mathcal{S}_{1,p} = \mathcal{S}_{1,p}^{AND} \oplus \mathcal{S}_{1,p}^{OR} \oplus \mathcal{S}_{1,p}^{XOR} \tag{5.50}$$

Donde:

$$\mathcal{S}_{1,p}^{AND} = (\mathcal{S}_{1,p}^{0} \wedge \mathcal{R}_{p}^{AND}) \tag{5.51}$$

$$\mathcal{S}_{1,p}^{OR} = (\mathcal{S}_{1,p}^{0} \vee \mathcal{R}_{p}^{OR}) \tag{5.52}$$

$$\mathcal{S}_{1,p}^{XOR} = (\mathcal{S}_{1,p}^{0} \bowtie \mathcal{R}_{p}^{XOR}) \tag{5.53}$$

Y cada componente se calcula como:

$$\mathcal{S}_{(1,i)}^{AND} = \bigwedge_{k=1}^{p} (\mathcal{S}_{(1,k)}^{0} \odot \mathcal{R}_{(k,i)}^{AND}) \tag{5.54}$$

$$\mathcal{S}_{(1,i)}^{OR} = \bigvee_{k=1}^{p} (\mathcal{S}_{(1,k)}^{0} \odot \mathcal{R}_{(k,i)}^{OR}) \tag{5.55}$$

$$\mathcal{S}_{(1,i)}^{XOR} = \bowtie_{k=1}^{p} (\mathcal{S}_{(1,k)}^{0} \odot \mathcal{R}_{(k,i)}^{XOR}) \tag{5.56}$$

Finalmente, para obtener el vector de satisfacción final, es necesario realizar un proceso de cálculo iterativo hasta obtener que la solución converja. Para ello, aplicamos el procedimiento que se muestra en el algoritmo 2.

Entrada del algoritmo

El algoritmo toma el conjunto de operacionalizaciones, Γ, y el conjunto de objetivos que deben satisfacerse, α. En primer lugar, se solicita al Ingeniero de Requisitos que seleccione aquellas operacionalizaciones que deben tenerse en cuenta para estudiar sus contribuciones. A continuación se rellena la matriz $\mathcal{M}$ con las contribuciones de las operacionalizaciones a los objetivos. Se obtiene el vector de satisfacción inicial $\mathcal{T}$ al aplicar la operación $\odot$ entre el vector de selección y la matriz de contribución.

Procedimiento

Posteriormente, se construyen las matrices $\mathcal{R}_{|\alpha|}^{AND}$, $\mathcal{R}_{|\alpha|}^{OR}$ y $\mathcal{R}_{|\alpha|}^{XOR}$ con las contribuciones presentes en el Grafo de Interdependencia. Adicionalmente, se inicializa el vector de satisfacción $\mathcal{S}_{1,|\alpha|}^{0}$, cuyas n componentes iniciales se toman del vector $\mathcal{T}$, calculadas en la etapa anterior por la contribución directa de las operacionalizaciones, y el resto de sus componentes se inicializan al valor desconocido, $\emptyset$. El algoritmo itera repetidamente calculando el vector de satisfacción general, $\mathcal{S}_{1,|\alpha|}^{i+1}$, y se detiene cuando la satisfacción computada entre 2 iteraciones sucesivas es la misma.

Algoritmo 2 Cálculo del Vector de Satisfacción de Objetivos

Require: Γ : {Operationalization}
Require: α : {Objective}
  $\mathcal{O}_{1,|\Gamma|} \leftarrow$ input from Req. Engineer
  $\mathcal{M}_{|\Gamma|,n} \leftarrow$ contributions from Interdependency Graph
  $\mathcal{T}^0_{1,n} \leftarrow \mathcal{O}_{1,|\Gamma|} \bigodot \mathcal{M}_{|\Gamma|,n}$
  $\mathcal{R}^{AND}_{|\alpha|} \leftarrow$ AND Refinements from Interdependency Graph
  $\mathcal{R}^{OR}_{|\alpha|} \leftarrow$ OR Refinements from Interdependency Graph
  $\mathcal{R}^{XOR}_{|\alpha|} \leftarrow$ XOR Refinements from Interdependency Graph
  $\mathcal{S}^0_{1,|\alpha|} \leftarrow \mathcal{T}_{1,n} \cup \{\emptyset\}_{|\alpha|-n}$
  repeat
    $\mathcal{S}^{AND}_{1,|\alpha|} \leftarrow (\mathcal{S}^i_{1,|\alpha|} \wedge \mathcal{R}^{AND}_{|\alpha|})$
    $\mathcal{S}^{OR}_{1,|\alpha|} \leftarrow (\mathcal{S}^i_{1,|\alpha|} \vee \mathcal{R}^{OR}_{|\alpha|})$
    $\mathcal{S}^{XOR}_{1,|\alpha|} \leftarrow (\mathcal{S}^i_{1,|\alpha|} \bowtie \mathcal{R}^{XOR}_{|\alpha|})$
    $\mathcal{S}^{i+1}_{1,|\alpha|} \leftarrow \mathcal{S}^{AND}_{1,p} \oplus \mathcal{S}^{OR}_{1,p} \oplus \mathcal{S}^{XOR}_{1,p}$
  until $\mathcal{S}^{i+1}_{1,|\alpha|} = \mathcal{S}^i_{1,|\alpha|}$
  return $\mathcal{S}^i_{1,|\alpha|}$

El último vector de satisfacción obtenido es el resultado de la evaluación.

Salida del algoritmo

Si el vector $\mathcal{S}^i_{1,|\alpha|}$ no contiene ningún elemento cuyos valores estén en el conjunto $\{\bigtriangledown, -, ?, \emptyset\}$, entonces la selección de operacionalizaciones satisface todos los objetivos. En caso contrario, debe realizarse un nuevo estudio y refinamiento para que se cumplan los objetivos.

5.11 Resumen

En este capítulo se ha propuesto un método de Ingeniería de Requisitos para Sistemas Ubicuos, llamado REUBI, para tratar sistemáticamente con los requisitos en este tipo de sistemas, teniendo en cuenta las características especiales del paradigma de la Computación Ubicua. A modo de resumen, las principales contribuciones de REUBI son:

- Representación de los requisitos para un sistema ubicuo en un Grafo de Interdependencia definido formalmente sobre la base de *goals* y *softgoals*, mostrando sus refinamientos.

- Proporcionar un medio para mostrar y analizar los posibles obstáculos que puedan aparecer y entorpecer la satisfacción de los objetivos a cumplir.
- Facilitar y guiar en el diseño de software de calidad explotando los beneficios del uso de técnicas de diseño que satisfagan completamente y de manera apropiada los *goals* y *softgoals* mediante el estudio de sus contribuciones.
- Mostrar el impacto del contexto del sistema en los requisitos.
- Abordar la importancia que tiene un cambio dinámico de prioridades dependiente del contexto en la satisfacción de ambos requisitos funcionales y no funcionales.
- Evaluar las decisiones tomadas con el objeto de determinar la satisfacción de los objetivos.

El resultado de la aplicación del método contiene información valiosa para la etapa de diseño del sistema. Esta información es especialmente relevante para identificar y derivar de forma justificada decisiones, tanto arquitectónicas del sistema como del diseño general de los subsistemas que forman parte de éste (todo ello a partir de las operalizaciones que permiten alcancar los objetivos), las adaptaciones que deben ocurrir en el sistema (dadas por los conjuntos de operacionalizaciones que se obtienen tras la evaluación para cada situación de contexto) y los eventos que disparan las adaptaciones que deben ocurrir (dados por los atributos que caracterizan cada situación de contexto, junto con sus valores asociados).

Además, el conocimiento generado por la aplicación del método puede ser reutilizado en otros desarrollos, o en futuras versiones del sistema. Así, la aparición de nuevos mecanismos o tecnologías que favorezcan la satisfacción de algunos requisitos pueden incorporarse al análisis realizado en forma de operacionalizaciones, las cuales se relacionan con los objetivos existentes. La ejecución del procedimiento de evaluación determinará la idoneidad o no de las nuevas soluciones propuestas para su incorporación en el sistema.

6 DEFINICIÓN DE PATRONES DE REQUISITOS PARA SISTEMAS UBICUOS

Índice

6.1 Introducción

En el capítulo anterior se ha presentado un método de Ingeniería de Requisitos para Sistemas Ubicuos, el cual tiene en cuenta las características principales de los Sistemas Ubicuos. Proporciona un metamodelo conforme al que modelar los requisitos y un método en el que se proporcionan guías para el modelado y elaboración de los requisitos. Sin embargo, frecuentemente, tratar con los NFRs en el análisis de Sistemas Ubicuos es una tarea difícil dado que requiere un amplio conocimiento y entendimiento, tanto de las características únicas del paradigma, como del entorno del sistema analizado.

Además, existen numerosas situaciones recurrentes y comunes a diferentes proyectos donde podrían aplicarse técnicas y hacer

uso de los mecanismos asociados para reutilizar el conocimiento generado en el modelado de requisitos. Sin duda, capturar el conocimiento relacionado con dichos problemas y las correspondientes situaciones para reutilizarlas posteriormente sería muy deseable.

Por este motivo, en este capítulo se presenta un conjunto de soluciones a situaciones recurrentes capturadas en términos de patrones de requisitos. El uso de patrones en la etapa de Ingeniería de Requisitos no es nuevo [SHC+10], pero ha recibido poca atención en general, y permanece inexplorado en el campo de los Sistemas Ubicuos. Este conjunto de patrones pueden ser aplicados de manera complementaria al método propuesto para simplificar la tarea de modelado de requisitos, o bien, pueden aplicarse junto con otros métodos existentes.

Como parte del trabajo realizado en esta tesis, se propone un conjunto de patrones, los cuales se han agrupado en cuatro categorías (descomposición, resolución, operacionalización y priorización) que están relacionadas con las etapas del método REUBI propuesto en el capítulo anterior (Figura 6.1), de manera que cada uno de los patrones puede aplicarse en la etapa correspondiente a su categoría. Además, se define un conjunto de operaciones para poder combinarlos o extenderlos, para dotar a la propuesta de mayor potencial en cuanto a aplicabilidad.

Para describir cada patrón se empleará una plantilla que describa los siguientes aspectos:

- Nombre del patrón.
- Propósito al que está orientado.
- Motivación para la existencia del patrón.
- Conceptos de REUBI involucrados en la definición del patrón.
- NFRs relacionados con la aplicación del patrón.
- Condiciones bajo las que el patrón es aplicable.
- Estructura del patrón.
- Patrones relacionados.

6.2 Patrones de descomposición

Patrones de descomposición

Este conjunto de patrones se enfoca a la descomposición de ob-

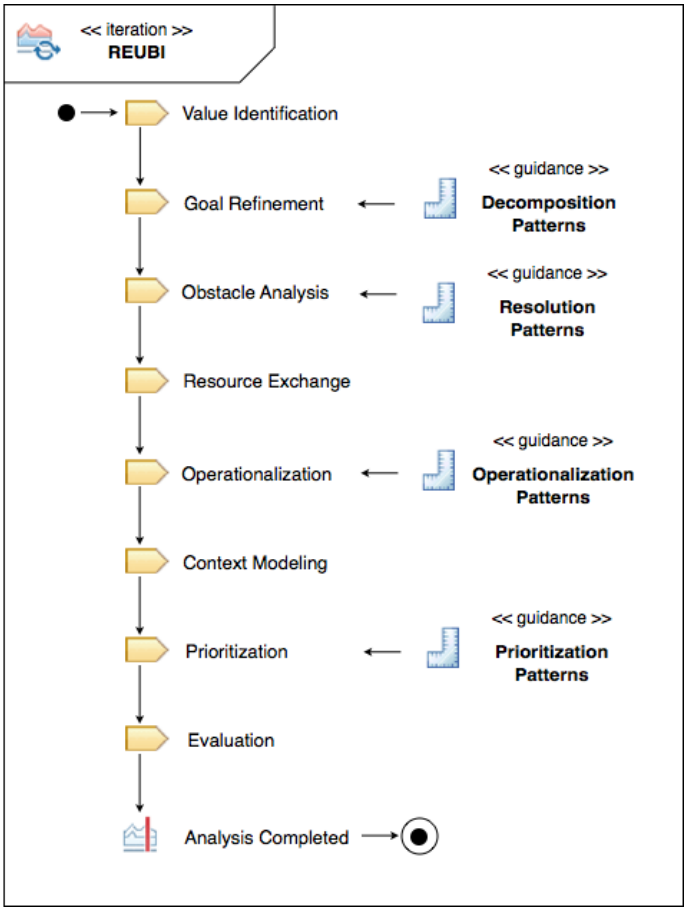

Figura 6.1: Correspondencia entre las fases de REUBI y las categorías de patrones propuestas.

jetivos generales en otros más concretos y de menor granularidad, los cuales puedan ser operacionalizados. Enfoques existentes sugieren la aplicación de descomposiciones basadas en el tipo o el tema del objetivo para dividirlo. También pueden emplearse catálogos de descomposición para reutilizar descomposiciones recurrentes [CNYMoo].

Los patrones definidos en esta categoría se centran en características que son esenciales en los sistemas ubicuos. El primero propone la definición precisa del concepto de ubicuidad en el ámbito del sistema en desarrollo; el segundo propone la descomposición de objetivos teniendo en cuenta la estructura del contexto en el que deben realizarse.

6.2.1 *Ubiquity definition*

Definición del concepto de Ubicuidad

Nombre del patrón: Ubiquity definition (Definición del concepto de Ubicuidad)

Propósito: Especificar de forma precisa el significado de la noción de *ubicuidad* en el sistema analizado, de manera que puedan encontrarse operacionalizaciones que satisfagan este requisito.

Motivación: Cuando los Analistas de Sistemas comienzan a definir un nuevo proyecto para un Sistema Ubicuo, una de las cuestiones principales que deben abordarse es cómo el sistema que se está analizando va a ser ubicuo. En otras palabras, existe la necesidad de definir lo que el concepto de *ubicuidad* significa en el contexto de dicho sistema.

A este respecto, este patrón propone la incorporación de un *softgoal*, titulado *Ubiquity*, el cual debe ser refinado de acuerdo a las necesidades específicas del sistema en cuestión. Por tanto, los *softgoals* de menor granularidad en los que ha sido descompuesto deben ser operacionalizados consecuentemente de manera que el objetivo inicial pueda ser satisfecho.

La Figura 6.2 muestra una descomposición general, aunque incompleta, de la noción de ubicuidad, teniendo en cuenta las características del paradigma de Computación Ubicua, las cuales pueden ser encontradas en la bibliografía. Dichas características incluyen:

- **Uso efectivo de ambientes inteligentes:** se basa en la detección de un usuario y sus necesidades, inferidas de su estado, independientemente de su localización. El espacio inteligente aparece cuando diferentes dispositivos inteligentes se encuentran en el mismo entorno físico y colaboran para dar soporte a las actividades del usuario que tienen lugar en dicho espacio físico.

- **Invisibilidad:** aunque la tecnología actual está lejos de la completa desaparición de la tecnología de la conciencia del usuario, es necesario tener en cuenta que los sistemas ubicuos deben ser lo menos intrusivos posible. El requisito de invisibilidad requiere cambios drásticos en el tipo de interfaces que se usan para comunicarse con los sistemas de computación, llevándonos a la necesidad de adoptar nuevas alternativas, tales como reconocimiento del habla y gestos; entendimiento del lenguaje natural oral y escrito; síntesis del habla; y representaciones gráficas.

- **Escalabilidad localizada:** el concepto de localidad de servicios es fundamental respecto a la universalidad de servicios en Internet. Se debe proporcionar a los usuarios ciertas funcionalidades asociadas a su contexto. Por tanto, no tiene sentido, por ejemplo, que las aplicaciones domóticas del hogar estén tratando de determinar las necesidades del usuario cuando éste está en la oficina.

- **Ocultación de los niveles de acondicionamiento:** dependiendo de la infraestructura y de los desarrollos tecnológicos disponibles en cada uno de los entornos específicos que conforman un entorno global, la distribución de servicios puede ser poco uniforme. En esta situación, el requisito de invisibilidad puede resultar insatisfecho, dado que el usuario detectaría transiciones entre entornos dada la ausencia de servicios. Una forma posible de superar esta situación sería definir un espacio personal con servicios propios del usuario, de manera que se compense la ausencia de servicios en ciertos entornos.

Conceptos de REUBI involucrados: Softgoal, *Goal*

Requisitos No Funcionales relacionados: Algunos de los NFRs más relevantes que pueden resultar afectados por la aplicación de este patrón son:

- **Invisibility:** como se comentó anteriormente, la invisivilidad es una característica central de la computación ubicua, de manera que el usuario no perciba la tecnología cuando la usa.

- **Intrusividad:** en relación con el requisito anterior, la tecnología invisible no debería interferir en la realización de las actividades diarias del usuario.

- **Aceptación del usuario:** en última instancia, el desconocimiento de la presencia de la tecnología podría mejorar la aceptación del usuario dado que realizan tareas asistidas por la tecnología de la misma manera en la que solían hacerlas.

- **Usabilidad:** los requisitos previos están directamente relacionados con la usabilidad del sistema. El uso de interfaces naturales (e.g., reconocimiento del habla, reconocimiento de gestos) ayuda a la realización de actividades de la misma manera en la que se realizan sin asistencia tecnológica.

- **Seguridad:** dado que estos sistemas tratan con datos personales importantes para proporcionar servicios personalizados, la seguridad es un requisito importante en el que los analistas deberían centrarse.
- **Precisión:** para proporcionar servicio personalizado al usuario, es importante obtener información contextual personalizada, de manera que el sistema no moleste o interrumpa al usuario.

Aplicabilidad: Este patrón permite ser aplicado de manera general. La mayoría de analistas deberían incorporar un *softgoal* de *Ubiquity* para definir claramente el significado de ubicuidad en el contexto de su proyecto. No hay restricción en el tipo de sistema que deba analizarse.

Estructura: La Figura 6.2 muestra la estructura del patrón usando un Grafo de Interdependencia con los conceptos propuestos en el metamodelo de REUBI.

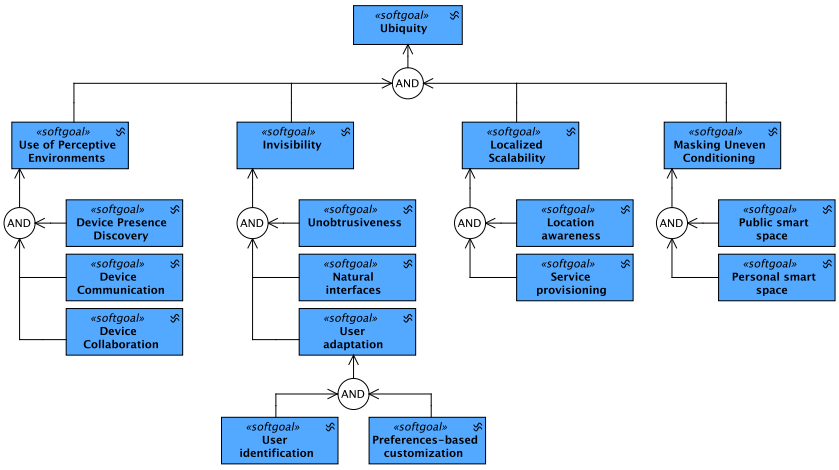

Figura 6.2: Ubiquity definition Pattern.

6.2.2 *Context-narrowing decomposition*

Descomposición basada en el contexto

Nombre del patrón: Context-narrowing decomposition (Descomposición basada en el contexto).

Propósito: Descomponer los objetivos en otros de menor granularidad teniendo en cuenta la estructura de las situaciones de contexto bajo las que tienen que ser satisfechos.

Motivación: Los objetivos de alto nivel sirven como un punto de partida para establecer el objetivo global del sistema, pero habitualmente son difíciles de operacionalizar; es decir, es difícil determinar con un nivel de granularidad aceptable qué decisiones pueden tomarse para satisfacerlos.

Para solucionar este problema, los objetivos de alto nivel se descomponen en otros más concretos y de menor granularidad, para los que es fácil encontrar decisiones alternativas que puedan satisfacerlos. Esta descomposición de objetivos puede hacerse teniendo en cuenta diferentes aspectos. En la bibliografía pueden encontrarse descomposiciones basadas en el tipo o el tema de los objetivos.

Sin embargo, una de las principales características de la Computación Ubicua es la consciencia del contexto. Como en la definición de objetivos, el contexto puede ser expresado con diferentes niveles de granularidad (e.g. en la escuela, en la primera planta, en la sección izquierda, en la habitación número 6). La realización de un objetivo puede deferir sustancialmente dependiendo del contexto; diferentes sub-objetivos pueden aparecer para diferentes situaciones, o distintos enfoques tecnológicos pueden aplicarse dependiendo de la situación.

Teniendo esto en mente, el patrón de descomposición basada en el contexto propone la descomposición de objetivos de alto nivel en otros de menor granularidad mediante la definición de un subobjetivo para cada subentorno en los que el contexto puede ser dividido. En otras palabras, asumiendo que la situación de contexto C puede dividirse en varios sub-escenarios disjuntos, $C_1 \dots C_n$, un *softgoal* de alto nivel S puede descomponerse en varios *sub-softgoals*, $S_1 \dots S_n$, los cuales son más fáciles de operacionalizar que el original.

Este patrón puede aplicarse recursivamente en un proceso que combina paralelamente descomposiciones de situaciones de contexto y de *softgoals*. El patrón puede usarse en combinación con otros patrones de descomposición existentes.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Context situation*

Requisitos No Funcionales relacionados: algunos de los NFRs más relevantes que pueden ser afectados por la aplicación de este patrón son:

- **Facilidad de desarrollo:** una descomposición efectiva de objetivos generales en otros más concretos facilita el desarrollo, gracias a una mejor separación de aspectos.

- **Facilidad de mantenimiento:** similarmente, si la descomposición se realiza adecuadamente, podría llevar a una mejor modularidad en el software, que en última instancia implica un mantenimiento más eficiente del software desarrollado.
- **Coste:** si el software es más sencillo de desarrollar y mantener, los costes asociados podrían ser menores.
- **Rendimiento:** una buena división de objetivos que deben ser satisfechos en un momento dado también contribuye positivamente a la eficiencia del sistema, el cual es capaz de identificar qué acciones o adaptaciones deben ser aplicadas en cada situación para satisfacer los requisitos.

Aplicabilidad: Este patrón es aplicable cuando los analistas tienen un *goal* o *softgoal* que es muy general para poder ser operacionalizado, y una situación de contexto donde este objetivo debe satisfacerse. Si la situación de contexto puede expresarse como la unión de diferentes situaciones con pequeñas variaciones en la satisfacción del objetivo, entonces puede descomponerse teniendo en cuenta estas sub-situaciones.

Estructura: La Figura 6.3 muestra la estructura del patrón haciendo uso de los conceptos propuestos en el metamodelo de REUBI.

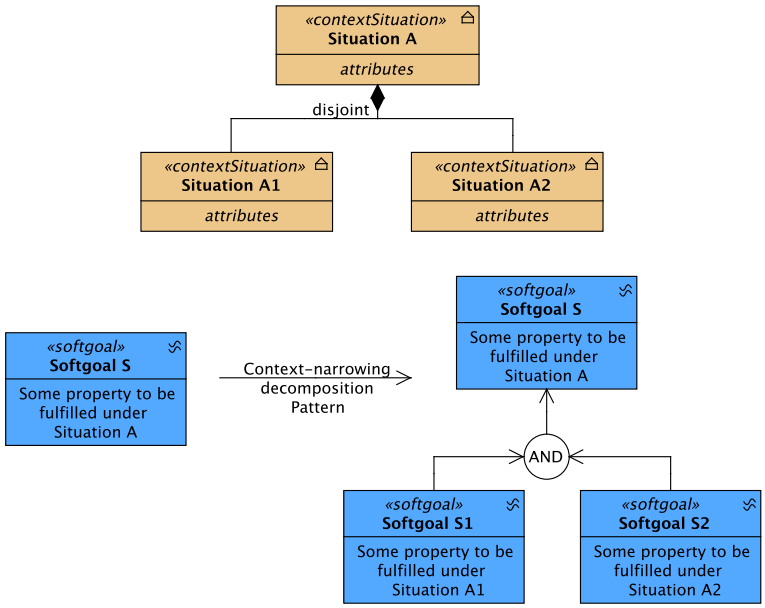

Figura 6.3: Context-narrowing decomposition Pattern.

Patrones relacionados: Topic-based decomposition, Type-based decomposition

6.3 Patrones de resolución

Patrones de resolución

Una estrategia de resolución de obstáculos efectiva tiene dos facetas: prevención y resolución. El conjunto de patrones que se identifican en esta categoría proporciona estrategias que pueden seguirse para identificar y resolver problemas causados por la aparición de situaciones inconvenientes que pueden perjudicar a los objetivos del sistema.

El primer patrón propone una estrategia que tiene en cuenta la probabilidad de ocurrencia de un obstáculo para determinar el protocolo que debe seguirse cuando ocurre. El segundo puede verse como una extensión del anterior, el cual también considera la gravedad de los daños que puede causar un obstáculo bajo ciertas situaciones de contexto.

6.3.1 *Obstacle-objective chain*

Cadena de obstáculos-objetivos

Nombre del patrón: Obstacle-objective chain (Cadena de obstáculos-objetivos).

Propósito: Establecer una estrategia de resolución de obstáculos teniendo en cuenta la probabilidad de ocurrencia de los obstáculos.

Motivación: Los entornos de Computación Ubicua son altamente dinámicos y difíciles de controlar; por tanto, pueden aparecer muchas situaciones inconvenientes. El método REUBI contempla esta situación e incorpora la noción de *obstáculo* en el metamodelo, con el propósito de modelar dichas situaciones que pueden suceder, entorpeciendo la satisfacción de un objetivo.

El método establece que cada obstáculo debería ser gestionado mediante la introducción de algún nuevo objetivo que evite la ocurrencia del obstáculo, mitigue sus efectos o corrija los fallos que pueda haber causado.

En algunos casos, pueden aparecer varios obstáculos que entorpecen la satisfacción de los objetivos que mitigan a otros obstáculos. El patrón de *cadena de obstáculos-objetivos* propone la representación de esta secuencia de obstáculos y objetivos que se entorpecen y mitigan respectivamente, como una cadena donde los obstáculos se ordenan por su probabilidad de ocurrencia y los objetivos se ordenan por algún otro criterio, como por ejemplo el coste o la facilidad de desarrollo.

De manera más precisa, sea $S_1 \dots S_n$ un conjunto de *softgoals*, y $O_1 \dots O_{n-1}$ un conjunto de obstáculos, donde:

$$S_i \; mitigates \; O_{i-1}, \forall i > 1 \tag{6.1}$$

$$O_i \; hinders \; S_i, \forall i \tag{6.2}$$

$$likelihood(O_i) > likelihood(O_j), \forall i < j \tag{6.3}$$

De esta forma, los obstáculos se mitigan progresivamente, lo cual afecta en última instancia al resto de los NFRs, como se comenta a continuación.

La probabilidad de ocurrencia de un obstáculo puede ser difícil de calcular a menudo. Lo que se propone aquí es ordenar cualitativamente los obstáculos según la frecuencia de aparición, que puede ser estimada aproximadamente.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Obstacle*.

Requisitos No Funcionales relacionados: Algunos de los NFRs más relevantes que pueden resultar afectados por la aplicación de este patrón son:

- **Robustez:** prever y prevenir los posibles obstáculos que pueden aparecer lleva a un sistema más robusto.
- **Coste:** ordenar los obstáculos por su probabilidad de ocurrencia permite que el diseñador tome decisiones sobre en qué partes del sistema deberían realizarse más esfuerzos. Un objetivo puede requerir un coste alto para ser desarrollado, pero mitiga un obstáculo que ocurre muy raramente. De esta forma, el analista afronta una situación de compromiso (*tradeoff*), debiendo decidir si el obstáculo es suficientemente importante para ser mitigado o no.
- **Aceptación de usuario:** un sistema que tiene menos fallos o que puede recuperarse de aquellos que raramente ocurren es más probable que sea aceptado por los usuarios.
- **Intrusividad:** la mitigación de obstáculos implica una menor interferencia con las actividades del usuario; por tanto, el sistema es menos intrusivo.

- **Invisibilidad:** de manera similar, si el usuario no percibe los fallos que ocurren en el sistema software, es más transparente para él.

Aplicabilidad: Este patrón puede aplicarse cuando existe un conjunto de obstáculos que puede ser ordenado en orden decreciente por la probabilidad de ocurrencia, y que puede encadenarse con el conjunto de objetivos correspondiente que los mitiga.

Estructura: La Figura 6.4 muestra la estructura del patrón, usando los conceptos propuestos en el metamodelo de REUBI.

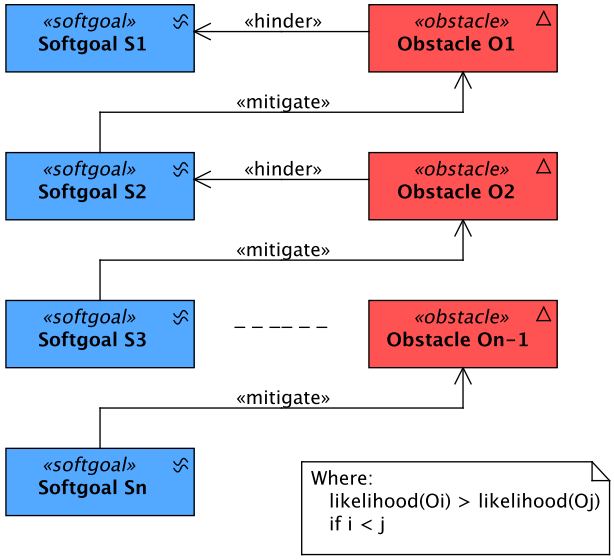

Figura 6.4: Obstacle-Objective chain Pattern.

6.3.2 *Severity-driven resolution*

Nombre del patrón: Severity-driven resolution (Resolución dirigida por la gravedad).

Resolución dirigida por la gravedad

Propósito: Modificar la estrategia de resolución de obstáculos si la gravedad de un obstáculo cambia bajo circunstancias contextuales.

Motivación: El patrón anterior presenta una estrategia para la resolución de obstáculos que tiene en cuenta la probabilidad de ocurrencia de los obstáculos, así como otros factores, tales como el coste o la facilidad de desarrollo, que afectan a la satisfacción de

otros objetivos. Además, la satisfacción de un objetivo en tiempo de ejecución puede requerir una alta consumición de recursos, que también puede tenerse en cuenta para ordenar los objetivos y colocarlos en la última parte de la cadena.

Sin embargo, hay otro factor que no se ha considerado. Los obstáculos causan daño al sistema con un cierto nivel de gravedad. Los obstáculos más probables a menudo causan daño menos severo dado que son más fáciles de predecir que los improbables.

Más aún, la severidad de un obstáculo no es fija. La ocurrencia de un obstáculo bajo circunstancias generales puede tener poco impacto en la satisfacción de los requisitos, pero, si ciertas situaciones contextuales están presentes, la severidad resultante de los obstáculos puede incrementarse, forzando la adopción de algunas otras soluciones para mitigar los efectos del obstáculo.

El patrón de resolución dirigida por la gravedad trata de dar soporte a este tipo de situaciones. Introduce la noción de situación de contexto que afecta a la mitigación de un obstáculo. El procedimiento habitual de resolución se define por la situación general como una cadena de obstáculos y objetivos, de manera similar al patrón previo. Entonces, la nueva estrategia de resolución se define para las circunstancias en las que se dan las nuevas situaciones de contexto.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Obstacle*, *Context situation*.

Requisitos No Funcionales relacionados: Algunos de los NFRs que pueden ser afectados por la aplicación de este patrón son:

- **Robustez:** la estrategia de resolución de obstáculos se mejora teniendo en cuenta la severidad de la ocurrencia de una situación inconveniente. De esta forma se obtiene un sistema más robusto.
- **Aceptación de usuario:** un sistema que tiene menor probabilidad de fallos y una mayor facilidad de recuperación es preferible para los usuarios finales.
- **Coste:** la definición de nuevas estrategias para evitar o mitigar el efecto de un obstáculo podría incrementar los costes asociados al desarrollo. Además, la resolución de daños más severos puede requerir el consumo de más recursos. Por consiguiente, el analista puede tener que solucionar el *tradeoff* existente entre el coste y otros requisitos.

Aplicabilidad: Este patrón es aplicable cuando hay una cadena de obstáculos y objetivos y la severidad de algunos de los obstáculos

varía bajo ciertas circunstancias contextuales. En ese caso, una nueva estrategia de resolución es aplicada.

Estructura: La Figura 6.5 muestra la estructura del patrón, haciendo uso de los conceptos propuestos en el metamodelo de REUBI.

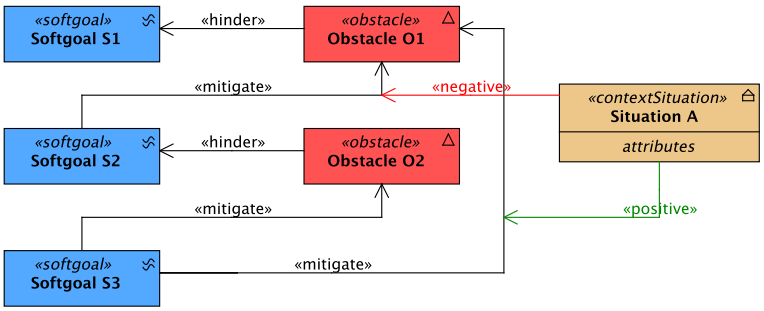

Figura 6.5: Severity-driven resolution Pattern.

Patrones relacionados: Obstacle-Objective chain

6.4 Patrones de operacionalización

Patrones de operacionalización

Buscar en el espacio de soluciones posibles para encontrar la tecnología o el método más apropiado en un sistema ubicuo es una dura tarea dada la heterogeneidad de soluciones disponibles y las diferentes propiedades de calidad que presenta cada alternativa. Además, no existe una solución que sea válida y dé los mejores resultados en todos los casos. Por tanto, optar por mecanismos de adaptación puede ser una buena opción para solucionar este problema.

Los patrones definidos en esta categoría presentan soluciones a problemas recurrentes en el análisis de sistemas ubicuos, tales como la entrada/salida de información al usuario, su autenticación mediante métodos no intrusivos, la prevención de que no exista un único punto de fallo cuando se realiza una tarea, o el intercambio de datos personales del usuario en entornos poco fiables.

6.4.1 *Context-sensitive I/O*

Nombre del patrón: Context-sensitive I/O (Entrada/Salida sensible al contexto).

Propósito: Determinar los mecanismos de entrada/salida que deberían usarse teniendo en cuenta el contexto, con el objeto de mantener la satisfacción de otros NFRS.

Motivación: Dos características clave de los sistemas ubicuos son la consciencia del contexto y la no intrusividad. El primero se refiere a la habilidad de estos sistemas de adaptar su contenido, comportamiento o estructura ante cambios en el entorno; el último, a no interferir en la forma habitual del usuario de realizar sus actividades.

Es importante que ambas características sean consideradas de manera conjunta. De hecho, dado que la consciencia del contexto implica cambios en el sistema software, el usuario puede llegar a ser consciente de tales adaptaciones y, en consecuencia, sea molestado por comportamientos disruptivos eventuales del sistema durante las adaptaciones.

Por otra parte, los sistemas ubicuos son esencialmente interactivos. Así, el usuario y el sistema deben intercambiar información usando interfaces naturales que, a la misma vez, no sean intrusivas para el usuario.

La forma en la que el sistema presenta información al usuario y el usuario proporciona entradas al sistema no debería ser fija. De hecho, depende de circunstancias relativas a su contexto. Dependiendo del entorno en el que está el usuario, el sistema puede decidir proporcionar la información de manera diferente, o incluso no hacer nada, si ello causa una disrupción en las actividades del usuario.

El patrón de entrada/salida sensible al contexto proporciona una posible solución para modelar esta situación, la cual es recurrente en diferentes sistemas ubicuos. Consideremos que hay un objetivo, G, que trata de proporcionar o recibir información del usuario. Diferentes operacionalizaciones, $O_1 \dots O_n$, pueden contribuir, positiva o negativamente, a la satisfacción de este objetivo con diferentes niveles de contribución. Finalmente, algunas situaciones de contexto, $C_1 \dots C_m$, son relevantes para el sistema.

Este patrón propone relacionar la operacionalización al objetivo, condicionado por la ocurrencia de algunas situaciones de contexto. Más formalmente:

$$O_i \; contributesTo \; G, \; if \; C_j \tag{6.4}$$

El tipo de contribución puede ser positivo o negativo, dependiendo de la situación. De esta forma, es más fácil para el analista discriminar qué opciones son más apropiadas en cada momento para la satisfacción de los objetivos. También, los mecanismos

de adaptación pueden obtenerse de la aplicación de este patrón, dado que en última instancia relaciona operacionalizaciones que deben ser ejecutadas ante eventos contextuales.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Operationalization*, *Context situation*.

Requisitos No Funcionales relacionados: Algunos de los NFRs que pueden resultar afectados por la aplicación de este patrón son:

- **Intrusividad:** este patrón intenta forzar al analista a pensar sobre aquellos mecanismos de entrada/salida que interfieren lo mínimo con las actividades del usuario, siendo capaz de diferenciar entre diferentes situaciones de contexto.
- **Usabilidad:** el empleo de interfaces naturales, adaptadas según el contexto del usuario, contribuye a crear un sistema que es más fácil de usar.
- **Facilidad de desarrollo:** la relación que se establece entre las operacionalizaciones y las situaciones de contexto para satisfacer los objetivos permite que los diseñadores y desarrolladores extraigan las reglas de adaptación de manera más fácil. Pueden ser implementadas y cambiadas con menos esfuerzo.
- **Coste:** la introducción de más mecanismos de interacción puede tener un impacto negativo en el coste. La implementación de interfaces naturales, tales como entendimiento del habla o de la escritura manual, está lejos del funcionamiento perfecto y son caras. Los analistas deberían ser conscientes de este conflicto (*tradeoff*) y considerar la importancia de usabilidad versus coste.

Aplicabilidad: este patrón es aplicable cuando hay un objetivo que requiere entrada o salida de información con el usuario, y existen varias alternativas, cada una de ellas con diferentes propiedades de calidad.

Estructura: La Figura 6.6 muestra la estructura del patrón, haciendo uso de los conceptos propuestos en el metamodelo de REUBI.

6.4.2 *Unobtrusive identification*

Nombre del patrón: Unobtrusive identification (Identificación no

Identifiación no intrusiva

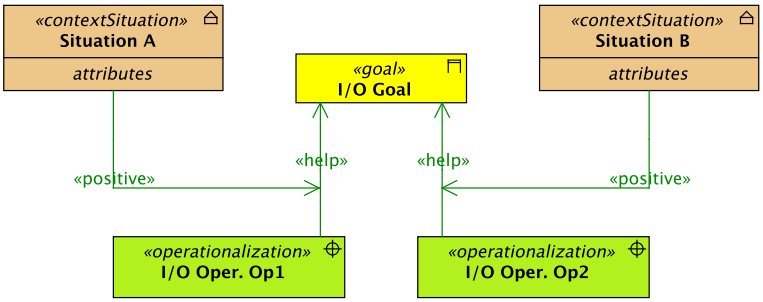

Figura 6.6: Context-sensitive I/O Pattern.

intrusiva).

Propósito: Realizar acciones relacionadas con la seguridad de manera segura sin interferir en la ejecución normal de dichas acciones y sin molestar al usuario.

También conocido como: Unobtrusive authentication (Autenticación no intrusiva).

Motivación: El paradigma de Computación Ubicua establece que la tecnología debería asistir a las personas en la realización de sus tareas mientras que son (parcialmente) inconscientes de la presencia de la propia tecnología. En otras palabras, la atención de las personas debería centrarse en la actividad en lugar de en la tecnología.

Sin embargo, en muchos casos, los usuarios necesitan realizar actividades que requieren seguridad, bien porque tratan con información privada, bien porque acceden a ciertos dispositivos cuyo uso está restringido. Así, el sistema ubicuo debe determinar la identidad del usuario para garantizar el acceso a los datos u operaciones deseadas.

En estos casos, aparece un dilema: cómo realizar la autenticación del usuario mientras que el sistema aún garantiza el requisito de no intrusividad.

Este patrón pretende cubrir esta situación mediante la aplicación de la siguiente estrategia. El analista se encuentra ante un compromiso entre seguridad y no intrusividad. Sin embargo, ambos requisitos pueden ser descompuestos en otros objetivos más concretos y de menor granularidad.

Referente a la seguridad, hay varias partes del sistema que deberían ser seguras, y cada una de ellas pueden requerir un nivel diferente de seguridad. Por ejemplo, la ejecución de operaciones requiere un nivel de seguridad más alto que las operaciones de

consulta. A este respecto, pueden aplicarse diferentes mecanismos de seguridad para cada situación que los necesite.

Paralelamente, el requisito de no intrusividad puede descomponerse también de la misma forma que el requisito de seguridad. Típicamente, la aplicación de mecanismos de seguridad más restrictivos causará un mayor daño a la satisfacción de la no intrusividad. Sin embargo, la descomposición de este requisito en diferentes subobjetivos permite el análisis para reducir el impacto de la solución de compromiso (*tradeoff*) para aquellos casos donde un alto nivel de seguridad se necesita.

En resumen, el patrón propone proceder de la siguiente manera:

1. Descomponer el *Seguridad* en diferentes sub *softgoals*, $S_1 \ldots S_n$, donde el nivel de S_i es mayor que el nivel de seguridad de S_j si $i < j$.

2. Descomponer el *softgoal* No Intrusividad en diferentes sub-*softgoals*, $U_1 \ldots U_n$, donde el nivel aceptado de no intrusividad de U_i es menor que el nivel aceptado de no intrusividad de U_j si $i < j$.

3. Asignar operacionalizaciones $O_1 \ldots O_n$, que ayuden a satisfacer los requisitos de seguridad, de tal forma que O_i contribuye positivamente tanto a S_i como a U_i, pero contribuye negativamente a U_j, para todo $j > i$.

Conceptos de REUBI involucrados: Softgoal, *Operationalization*, *Context situation*.

Requisitos No Funcionales relacionados: Algunos de los NFRs que pueden resultar afectados por la aplicación de este patrón son:

- **Seguridad:** uno de los principales requisitos que este patrón trata de garantizar es la seguridad mediante la incorporación de mecanismos apropiados que proporcionen acceso seguro a datos u operaciones sensibles.

- **Intrusividad:** el otro requisito principal es la no intrusividad, el cual está en conflicto con el requisito anterior. La aplicación de diferentes mecanismos de seguridad dependiendo de la importancia del recurso accedido permite un mejor balanceo (*tradeoff*) entre ambos requisitos, conllevando una mejor calidad del sistema.

- **Precisión:** la posibilidad de introducir mecanismos de seguridad no intrusivos que no molestan al usuario mientras

se le identifica depende en gran medida de su precisión. Así, el requisito de precisión debe tenerse en cuenta cuando se realiza la selección de las operacionalizaciones correspondientes.

- **Aceptación de usuario:** si se consigue una selección de mecanismos de seguridad no intrusivos adecuada, la aceptación del sistema por parte de los usuarios se mejorará; el sistema no interrumpirá continuamente a los usuarios y sus datos personales estarán asegurados.

- **Privacidad:** la imposición de restricciones sobre el acceso a datos personales implica una mejor satisfacción de los requisitos de privacidad.

- **Coste:** la introducción de diferentes mecanismos de seguridad puede incrementar los costes de desarrollo, dado que deben introducirse más mecanismos de autenticación. Este es otro conflicto (*tradeoff*) que el analista debería tener en cuenta cuando se aplique este patrón.

Aplicabilidad: Este patrón es aplicable cuando algunas partes del sistema deben garantizar acceso mediante autenticación segura, pero hay varios niveles de seguridad aceptables para cada parte del sistema.

Estructura: La Figura 6.7 muestra la estructura del patrón, usando conceptos propuestos en el metamodelo de REUBI.

6.4.3 *Redundant operationalization*

Operacionalización redundante

Nombre del patrón: Redundant operationalization (Operacionalización redundante).

Propósito: Evitar un único punto de fallo que pueda causar la insatisfacción de un Requisito No Funcional.

También conocido como: Operationalization composition (Composición de operacionalización).

Motivación: La dinamicidad de los entornos de computación ubicua implica la ocurrencia probable de situaciones inconvenientes que pueden causar fallos en la satisfacción de los objetivos. Como se apuntaba antes, el método REUBI propone la introducción de

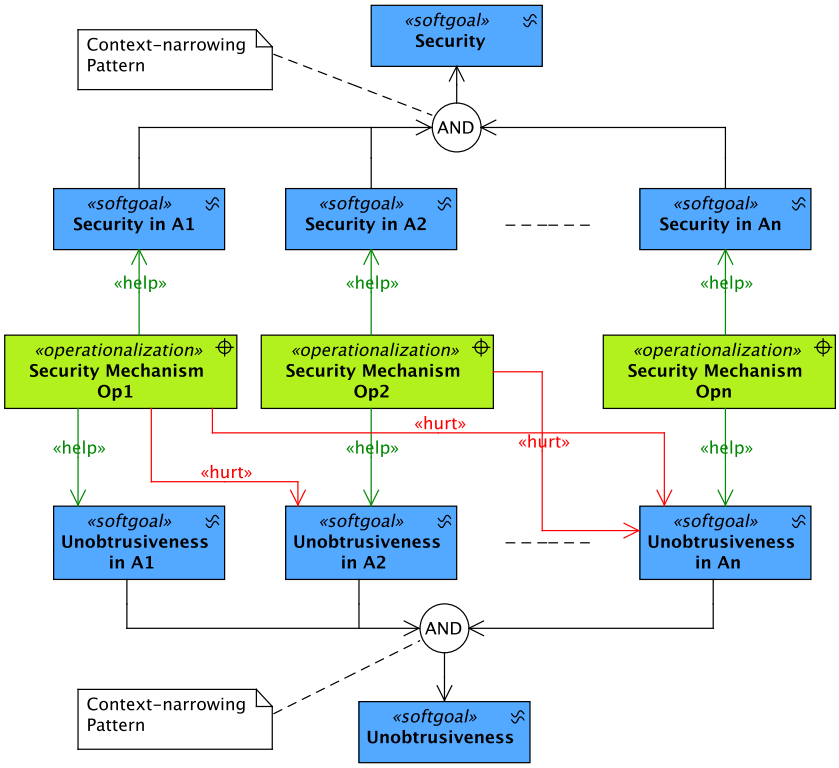

Figura 6.7: Unobtrusive identification Pattern.

objetivos (*goals* o *softgoals*) que pueden mitigar los efectos de la ocurrencia de un obstáculo.

Un problema habitual que puede ocurrir es el fallo de una de las operacionalizaciones que son responsables de la satisfacción de un objetivo. Éste es el caso de, por ejemplo, sensores o actuadores. Hay muchos escenarios posibles donde su funcionamiento no es el esperado, por ejemplo, causado por un corte en la alimentación eléctrica o un fallo en el hardware.

El patrón de operacionalización redundante propone la introducción de una combinación de operacionalizaciones. La introducción de redundancias es una técnica bien conocida en sistemas distribuidos para incrementar la robustez del sistema en caso de fallo de una de sus partes. Gracias a ello, se evita la aparición de un único punto de fallo.

El agrupamiento y la combinación de diferentes alternativas para satisfacer un objetivo también provoca la aparición de nuevas propiedades que cada una de las partes no tiene por sí misma. El analista debería ser consciente de este fenómeno, dado que podría conllevar la aparición de contribuciones positivas a la realización de otros objetivos, pero también a la ocurrencia de contribuciones negativas, teniendo así un conflicto entre diferentes requisitos.

También, la relación entre operacionalizaciones debe ser estudiada. Existen diferentes relaciones entre componentes:

- **Ortogonal:** la aplicación de ambas operacionalizaciones no se solapa; no se mejoran ni empeoran mutuamente.
- **Complementaria:** la aplicación de ambas operacionalizaciones mejora el resultado final u otras propiedades relativas al resultado.
- **Opuesta:** la aplicación de ambas operacionalizaciones degrada el resultado final u otras propiedades relativas al resultado.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Operationalization*, *Group*, *Obstacle*.

Requisitos No Funcionales relacionados: Algunos de los NFRs más relevantes que pueden resultar afectados por la aplicación de este patrón son:

- **Robustez:** como se apuntaba en la introducción del patrón, la aplicación de alternativas redundantes para satisfacer un objetivo es una técnica bien conocida para incrementar la robustez en un sistema distribuido.
- **Precisión:** en otros casos, la introducción de más operacionalizaciones para satisfacer un objetivo puede implicar una mejor precisión de los resultados, dado que las salidas de cada alternativa pueden compararse para incrementar la fiabilidad del resultado.
- **Rendimiento:** la introducción del operacionalizaciones redundantes puede implicar una degradación del rendimiento del sistema. Incluso aunque en algunos casos las operacionalizaciones combinadas pueden aplicarse en paralelo, el tiempo global para combinar sus resultados y el uso de los recursos puede incrementarse.
- **Coste:** la necesidad de implementar más alternativas para el mismo propósito incrementa el coste de desarrollo en última instancia. El analista debe ser consciente de este conflicto cuando se aplica este patrón.

Aplicabilidad: este patrón es aplicable cuando hay una combinación de operacionalizaciones posible que mejora la satisfacción de diferentes objetivos, o la mitigación de obstáculos, que no

pueden realizarse mediante la aplicación aislada de una operacionalización.

Estructura: La Figura 6.8 muestra la estructura del patrón, haciendo uso de los conceptos propuestos por el metamodelo de REUBI.

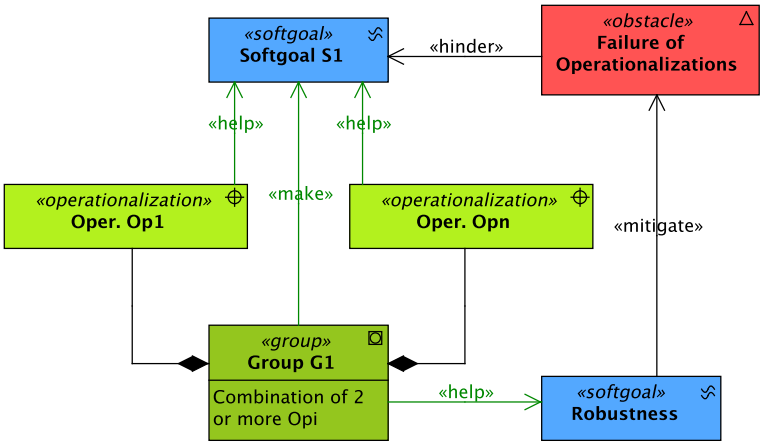

Figura 6.8: Redundant operationalization Pattern.

6.4.4 *Pseudonymity*

Nombre del patrón: Pseudonimity. *Pseudonimidad*

Propósito: Establecer la cantidad de datos personales del usuario que se intercambian dependiendo de la confianza en la situación de contexto.

También conocido como: Context-sensitive data sharing (Intercambio de datos sensible al contexto).

Motivación: La preservación de los datos personales es un reto general para los diseñadores de software, lo cual es igualmente cierto para los sistemas ubicuos. La personalización y la adaptación al usuario son características clave de estos sistemas, y para poder lograrlas, necesitan conocer grandes cantidades de datos personales del usuario.

Para conseguir esta tarea, el usuario proporciona estos datos como entrada a los mecanismos de adaptación, o bien, el sistema recopila eventos de la vida diaria del usuario y aplica técnicas de minería de datos para inferir el comportamiento del usuario, e intentar anticipar la personalización.

Pueden ocurrir escenarios peligrosos cuando todos estos datos personales tienen que ser intercambiados con otros servicios. Las adaptaciones en nuevos entornos o el intercambio de datos con otros usuarios pueden provocar la exposición de datos sensibles que el usuario no quiere compartir.

Para resolver esta situación, el patrón *Pseudonimity* propone tener en cuenta la confianza en la situación de contexto para determinar cuánta información se comparte. Para lograr esto, la información personal del usuario se organiza en un retículo. Se define una relación de orden, R, donde $R(i, j)$ significa que j proporciona al menos la misma información personal que i. Esta definición crea una relación de orden total, dado que se verifican las siguientes propiedades:

- **Reflexiva:** $R(i, i)$ es siempre cierto, dado que i siempre incluye al menos la misma información que él mismo.
- **Antisimétrica:** si $R(i, j)$ es verdadero, entonces $R(j, i)$ puede no ser cierta, ya que j puede incorporar información adicional a i.
- **Transitiva:** si $R(i, j)$ es cierto, y $R(j, k)$ es cierto, entonces $R(i, k)$ también es cierto. Dado que k tiene al menos la misma información que j, y j tiene al menos la misma información que i, entonces k tiene al menos la misma información que i.

Los elementos maximales y minimales de este retículo son siempre los siguientes:

- **Top** ($\top$): todos los datos del usuario están disponibles; i.e., $\forall x : R(x, \top)$.
- **Bottom** ($\bot$): ningún dato personal del usuario está disponible; i.e., $\forall x : R(\bot, x)$.

Cuanto el retículo ha sido creado, cada nodo representa la cantidad de información que puede ser compartida. Estos nodos se relacionan con las situaciones de contexto donde deben ser compartidos, de tal manera que cuanto menos confianza se tiene en la situación de contexto correspondiente, menos información del usuario se comparte.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Operationalization*, *Context situation*.

Requisitos No Funcionales relacionados: Algunos de los NFRs más relevantes que pueden resultar afectados por la aplicación de este patrón son:

- **Privacidad:** la organización propuesta de los datos personales de usuario ayudan a mantener la privacidad del usuario mediante el establecimiento de políticas de acceso a los datos basadas en la confianza.
- **Seguridad:** la aplicación de tales políticas de privacidad también contribuye a la obtención de un sistema más seguro, dado que previene de la exposición de datos.
- **Precisión:** determinar la confianza de la situación de contexto de manera precisa es importante para garantizar el mantenimiento de la privacidad. Un enfoque pesimista puede ser deseable en estas situaciones, dado que el objetivo es prevenir el filtrado no deseado de información.
- **Adaptabilidad:** la adaptabilidad del sistema puede, en ocasiones, verse perjudicada por la aplicación de este patrón. De hecho, si el sistema no cuenta con datos suficientes, la personalización del comportamiento del sistema puede no funcionar de la manera esperada.
- **Intrusividad:** la prevención del filtrado de información también puede contribuir a la no intrusividad del sistema, especialmente si se considera el software *malware* o *adware*. Este tipo de software emplean información del usuario para aprender comportamientos o molestarlo con publicidad.

Aplicabilidad: Este patrón es aplicable cuando los datos personales del usuario deben intercambiarse en diferentes situaciones de contexto, cada una de ellas con diferentes niveles de confianza.

Estructura: La Figura 6.9 muestra la estructura de este patrón, haciendo uso de los conceptos propuestos en el metamodelo de REUBI.

6.5 Patrones de selección

Patrones de selección

La selección entre diferentes alternativas en el método REUBI se realiza siguiendo un enfoque dirigido por la calidad; es decir, la satisfacción de los NFRs habitualmente determina la elección

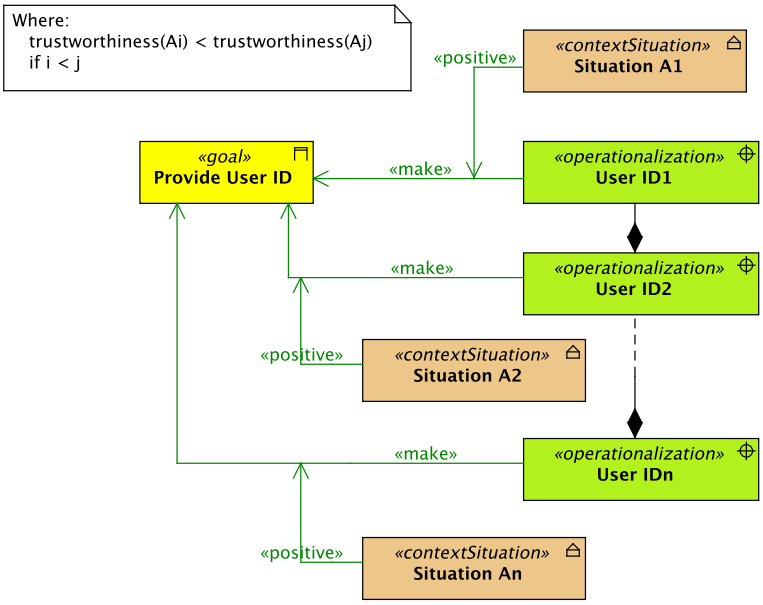

Figura 6.9: Pseudonimity Pattern.

o no de cierta operacionalización. Cuando ocurre un conflicto (*tradeoff*), la prioridad de los NFRs se emplea para deshacerlo.

Sin embargo, dado que la prioridad de los requisitos puede variar bajo diferentes situaciones, ello implica que diferentes selecciones pueden realizarse. El patrón que se define en esta categoría trata de capturar esta situación.

6.5.1 *Variable priority*

Prioridad variable

Nombre del patrón: Variable priority (Prioridad variable)

Propósito: Especificar los cambios en la prioridad de un objetivo dependiendo de cambios en el contexto.

Motivación: Los requisitos de las aplicaciones de escritorio actuales normalmente tienen una prioridad fija que no cambia a lo largo de la ejecución de la aplicación. Sin embargo, éste no es el caso de los sistemas y aplicaciones ubicuos.

Los requisitos para este tipo de sistemas tienden a cambiar durante la ejecución debido a cambios en el entorno. Esta situación a menudo causa la aplicación de adaptaciones en dichos sistemas para acomodarse a la nueva situación en términos de priorización de requisitos.

Los cambios en la priorización pueden implicar la aplicación de una solución diferente para satisfacer el objetivo, especialmente

teniendo en cuenta que el incremento del nivel de prioridad de un requisito puede llevar implícito la necesidad de reducir la prioridad de otros. Así, diferentes operacionalizaciones pueden incluirse para satisfacer los objetivos con diferentes niveles de prioridad.

El patrón de prioridad variable trata de modelar esta situación. El metamodelo de REUBI considera tres niveles de prioridad: *Normal*, *Importante* y *Crítico*. Asumiendo que un objetivo podría tener estos tres niveles de prioridad en diferentes situaciones de contexto, S_1, S_2, S_3, el analista debe encontrar operacionalizaciones, O_1, O_2, O_3, no necesariamente distintas, que contribuyan positivamente a la satisfacción del objetivo. Estas operacionalizaciones están directamente relacionadas con los objetivos que tratan de cumplir con el nivel de prioridad apropiado, mientras que los objetivos están condicionados por la ocurrencia de la correspondiente situación de contexto.

Conceptos de REUBI involucrados: Softgoal, *Goal*, *Operationalization*, *Context situation*.

Requisitos No Funcionales relacionados: Algunos de los NFRs más relevantes que pueden resultar afectados por la aplicación de este patrón son:

- **Adaptabilidad:** dada la necesidad de cambiar las operacionalizaciones en tiempo de ejecución debido a cambios en la prioridad, se fuerza necesariamente la incorporación de mecanismos de adaptación en el sistema.
- **Aceptación del usuario:** si las correspondientes adaptaciones se realizan exitosamente, las propiedades de calidad que se esperan del sistema en cada momento deberían estar garantizadas y, por consiguiente, la aceptación del usuario del sistema podría incrementarse.
- **Coste:** la incorporación de mecanismos de adaptación y diferentes alternativas para realizar el mismo objetivo puede implicar un incremento de los costes, tanto en el desarrollo como en tiempo de ejecución.

Aplicabilidad: Este patrón es aplicable cuando hay un objetivo cuya prioridad puede cambiar durante la ejecución del sistema debido a cambios en el contexto.

Estructura: La Figura 6.10 muestra la estructura del patrón haciendo uso de los conceptos propuestos por el metamodelo de REUBI.

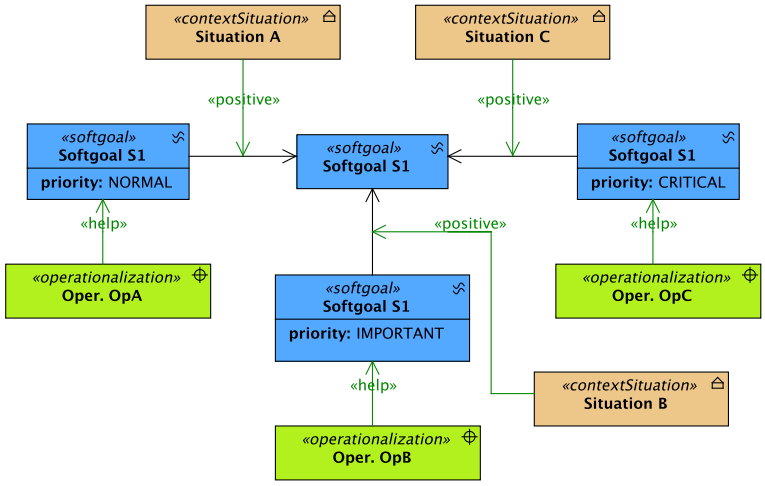

Figura 6.10: Variable priority Pattern.

6.6 Operaciones con Patrones

Operaciones con Patrones

Los patrones definidos anteriormente son sólo un subconjunto de un mayor número de patrones que podrían obtenerse para este tipo de sistemas. Además, existen diferentes operaciones que permiten la identificación de nuevos patrones, tales como generalización, especialización, composición e instanciación.

6.6.1 Generalización

Generalización de patrones

El análisis de los patrones definidos puede revelar algunos aspectos comunes entre ellos. De hecho, la idea principal tras algunos de ellos es aplicar diferentes alternativas bajo cada situación de contexto. Esta estructura de patrón puede factorizarse y extraerse como un patrón de más alto nivel; es decir, puede generalizarse. El patrón de variación sensible al contexto (*context-sensitive variation*) trata de capturar aquellas situaciones donde un cambio en el contexto del usuario provoca un cambio en el sistema (Figura 6.11).

6.6.2 Especialización

Especialización de patrones

Similarmente, los patrones definidos pueden especializarse con la incorporación de conocimiento adicional que puede abordar algunas características especiales de un campo determinado. Por ejemplo, el patrón de entrada/salida sensible al contexto (*Context-sensitive I/O*) puede especializarse en dos patrones diferentes con

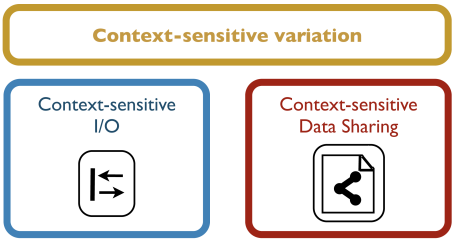

Figura 6.11: Generalización de los patrones *Context-sensitive I/O* y *Context-sensitive Data Sharing* en el patrón *Context-sensitive variation*.

la incorporación de los mecanismos específicos de entrada y salida de aplicaciones Web y móviles (Figure 6.12).

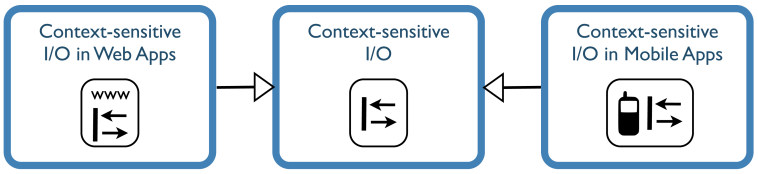

Figura 6.12: Especialización del patrón *Context-sensitive I/O* incorporando conocimiento específico para aplicaciones Web y móviles.

6.6.3 Composición

Composición de patrones

Los patrones definidos no se aplican de manera aislada. Pueden ser compuestos para dar lugar otros más complejos que puedan abordar situaciones donde existe una interacción entre diferentes NFRs. Por ejemplo, en algunos casos el sistema puede necesitar entregar un mensaje con información sensible al usuario. En este caso, los patrones *Context-sensitive I/O* y *Context-sensitive Data Sharing* pueden combinarse para determinar el nivel apropiado de información que debe entregarse al usuario y el mecanismo de salida que se empleará para tal efecto (Figura 6.13). De manera similar, algunos otros patrones pueden componerse para abordar situaciones más complejas.

6.6.4 Instanciación

Instanciación de patrones

Por último, pero no menos importante, los patrones pueden instanciarse para abordar los NFRs en un proyecto particular. En este caso, la Figura 6.14 muestra la instanciación del patrón

Figura 6.13: Composición de los patrones *Context-sensitive I/O* y *Context-sensitive Data Sharing*.

Context-sensitive I/O en un caso de estudio de un sistema de e-Learning.

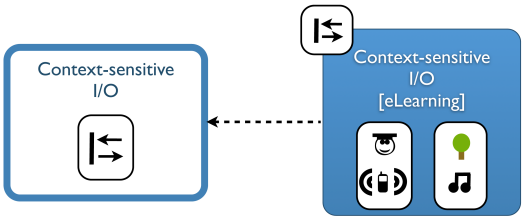

Figura 6.14: Instanciacion del patrón *Context-sensitive I/O* en un ejemplo de un sistema de e-Learning.

6.7 Resumen

En este capítulo se ha definido un conjunto de patrones que capturan información sobre situaciones recurrentes en el modelado de requisitos en Sistemas Ubicuos. El conjunto de patrones se ha organizado temáticamente en cuatro categorías: descomposición, resolución, operacionalización y selección. Cada patrón puede aplicarse en ciertas etapas del método de Ingeniería de Requisitos REUBI, propuesto en el capítulo anterior.

Los patrones han sido recogidos en una plantilla similar a la empleada para definir patrones de diseño y empleando una notación basada en el estándar UML, de manera que puedan ser comprendidos por la mayoría de ingenieros. Además, esto permite que el catálogo de patrones pueda ser extendido con nuevas situaciones recurrentes y sus correspondientes soluciones.

Adicionalmente, existen operaciones (generalización, especialización, composición o instanciación) que permiten obtener nuevos patrones a partir de los que se han propuesto en este capítulo.

La aplicación de patrones tiene numerosas ventajas: acelera el desarrollo, incrementa la posibilidad de reutilización de conocimiento y/o artefactos software, proporciona soluciones generales que pueden aplicarse en diversos ámbitos y establecen

una terminología común para referirse a problemas recurrentes, mejorando la comunicación entre analistas y diseñadores. Sin embargo, deben aplicarse cuidadosamente, dado que un mal uso de la aplicación de patrones puede llevar a un incremento innecesario de la complejidad del software.

7 SCUBI: MÉTODO DE DISEÑO DE SERVICIOS BASADO EN COMPONENTES PARA SISTEMAS UBICUOS

Índice

7.1 Introducción

Tras analizar diferentes propuestas, métodos y técnicas para el diseño de Sistemas Ubicuos en el capítulo 4, se llegó a la conclusión de que las propuestas existentes no son suficientes por tratarse de trabajos orientados a sistemas de propósito general y por tanto no abordan suficientemente y de forma específica las principales características que que exhiben los Sistemas Ubicuos.

Así, de acuerdo con los objetivos planteados para este trabajo de investigación, en este capítulo se presenta un **Método de Diseño para Servicios basado en Componentes para Sistemas Ubicuos**, denominado SCUBI, cuyo fin es tratar de cubrir las carencias detectadas en el análisis de las propuestas existentes haciendo uso de técnicas procedentes de la Ingeniería del Software. Sus características principales son:

Características de SCUBI

- Sigue un enfoque *bottom-up* para identificar los componentes básicos, agruparlos en unidades más complejas y ensamblarlos para formar servicios.
- Es conforme al estándar UML.

- Promueve la reutilización, no sólo a nivel de servicio, sino a nivel de componente, dado que las partes individuales se crean como bloques básicos que siguen los principios de la Ingeniería de Software basada en Componentes.
- Habilita el control de partes del sistema, tanto por un operador humano como por un agente software.
- Tiene en cuenta los cambios en el contexto para modificar el comportamiento y estructura de los servicios y sus partes.
- Permite la adaptación de los servicios con diferentes niveles de granularidad.

Descripción general del proceso

La Figura 7.1 muestra las distintas etapas en la aplicación del método SCUBI: identificación de la funcionalidad principal del sistema, elaboración de un diseño basado en componentes, incorporación de características de los Sistemas Ubicuos a dicho diseño y creación de servicios mediante el ensamblado de componentes. En las siguientes secciones se describen con detalle cada una de las actividades que forman parte de estas etapas.

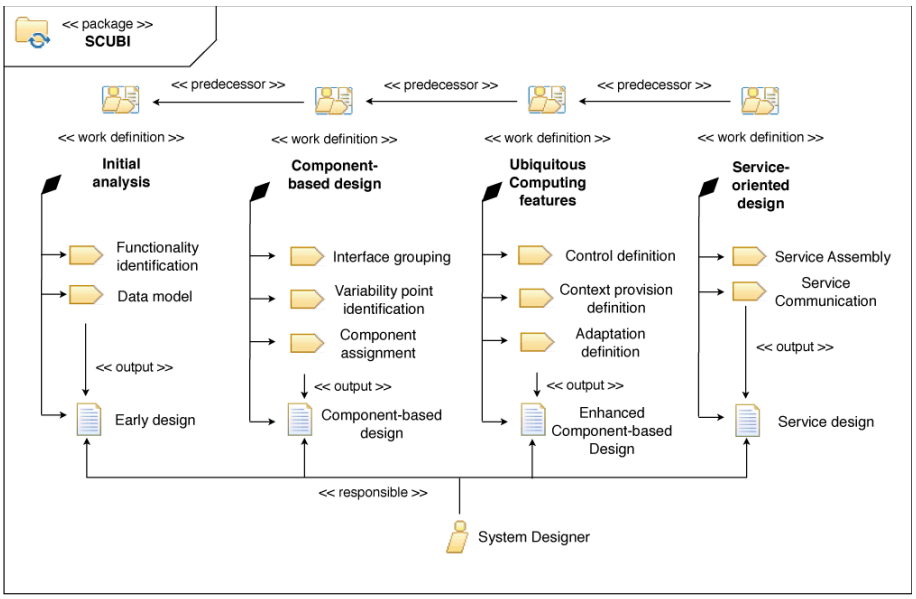

Figura 7.1: Definición de la tarea de Identificación de la funcionalidad en notación SPEM (Véase leyenda en Tabla 4.3)

En el Capítulo 8 se establecerá una correspondencia entre REUBI y SCUBI tratando de mostrar cómo realizar una transformación de modelos entre el modelo de requisitos y el modelo de diseño.

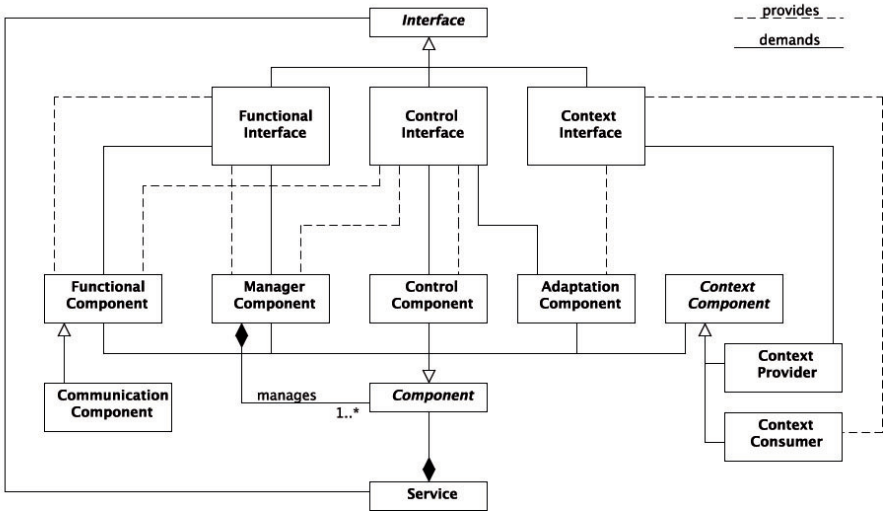

Figura 7.2: Metamodelo de SCUBI.

7.2 Identificación de la funcionalidad

Identificación de la funcionalidad

El primer paso en el método SCUBI consiste en la **identificación de la funcionalidad** (Figura 7.3) que debe proporcionar el sistema. En este sentido, esta actividad esta altamente relacionada con la fase anterior en el proceso de desarrollo; específicamente, esta actividad sirve como nexo de unión con los resultados de la fase de Ingeniería de Requisitos.

uso de REUBI

Para este fin, SCUBI recomienda comenzar el proceso con la aplicación del método REUBI, propuesto anteriormente. REUBI aborda los aspectos principales del análisis de los requisitos de los sistemas ubicuos y los resultados obtenidos son susceptibles de emplearse en el método SCUBI. En particular, REUBI ofrece una lista de los requisitos funcionales que deben cumplirse, de donde puede extraerse la información necesaria para llevar a cabo el diseño de una forma más sistemática.

Técnicas alternativas

Sin embargo, pueden aplicarse opciones alternativas para realizar esta tarea, aunque no se cuente con los beneficios que proporciona la aplicación de REUBI (tales como tratamiento sistemático de los NFR o incorporación del impacto del contexto en los requisitos). Por ejemplo, los diseñadores pueden hacer uso de casos de uso UML dado que los sistemas ubicuos son centrados en el usuario por naturaleza y los casos de uso son un buen medio para comunicarse con los usuarios. Otra posibilidad pasa por el uso de diagramas de colaboración y de secuencia UML, para

obtener una descripción más técnica de la funcionalidad. Una combinación de ambos enfoques (casos de uso para una identificación temprana, diagramas de colaboración/secuencia en una elaboración posterior), o junto con la aplicación de REUBI puede ser la opción más apropiada.

La salida de esta etapa debe ser una especificación técnica de las funcionalidades en términos de cabeceras de métodos. Puede resultar difícil obtener un conjunto completo de funcionalidades; por tanto, los diseñadores deberían seguir un proceso de refinamiento iterativo y aumentativo para completar esta actividad.

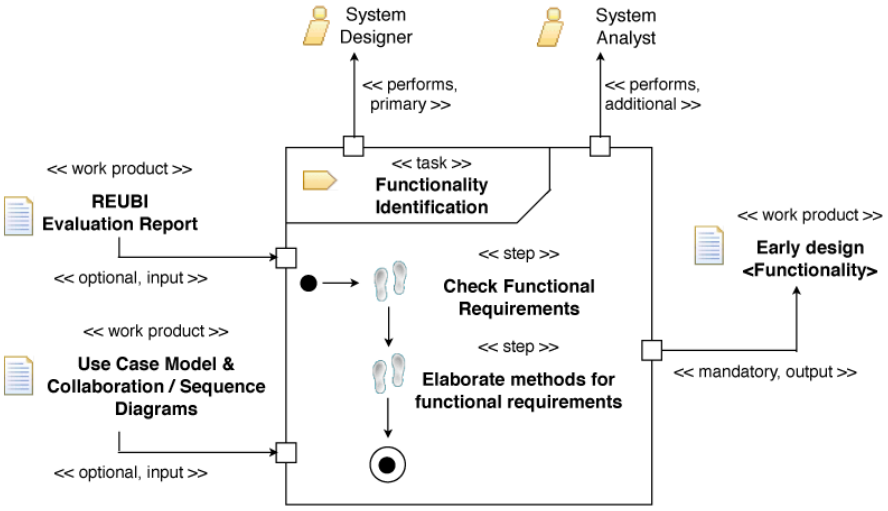

Figura 7.3: Definición de la tarea de Identificación de la funcionalidad en notación SPEM (Véase leyenda en Tabla 4.3)

7.3 Modelado de dominio

Modelado de dominio

El otro punto de entrada a SCUBI consiste en la elaboración de un modelo de datos (Figura 7.4). Esta tarea es susceptible de ser realizada en paralelo a la actividad anterior, dado que el descubrimiento de nuevas clases y tipos de datos puede implicar la necesidad de nueva funcionalidad, y viceversa.

Al mismo tiempo, esta tarea puede dividirse en dos subtareas. En los estadíos iniciales de esta fase, el objetivo es la construcción de un modelo inicial de los tipos de datos que se necesitan en el sistema, lo cual se conoce a menudo como **modelado de dominio**. Este modelo debería expresarse en términos de un diagrama de clases UML.

Más tarde en el método que se propone, el modelo de dominio puede ser iterativa e incrementalmente refinado con la incorporación de información de granularidad más fina, como los atributos de cada clase, para describir en profundidad el modelo.

El modelo de datos elaborado en esta etapa ayuda eventualmente a agrupar la funcionalidad en diferentes interfaces, y en última instancia, a descubrir los componentes y servicios necesarios, como se indica en las siguientes secciones.

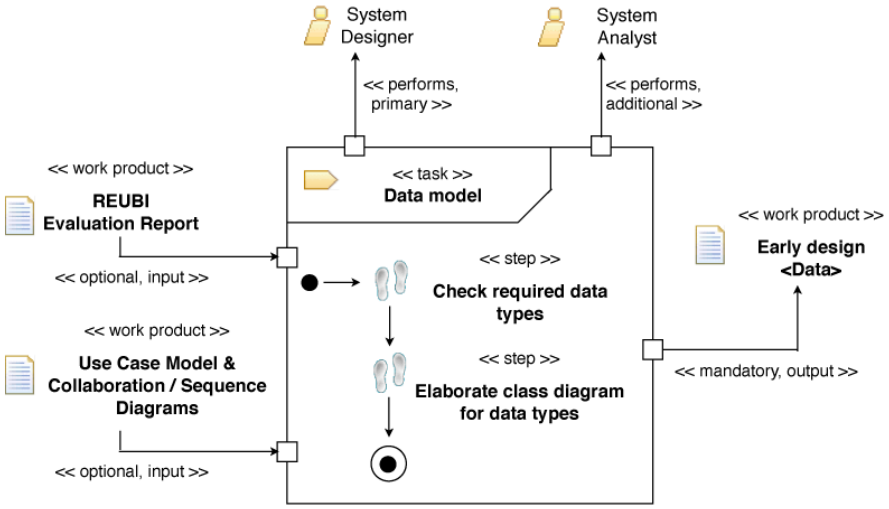

Figura 7.4: Definición de la tarea de Modelado de dominio en notación SPEM (Véase leyenda en Tabla 4.3)

7.4 Agrupamiento de interfaces

Agrupamiento de interfaces

Una vez que se han identificado las funcionalidades, deben agruparse en diferentes interfaces. El metamodelo de SCUBI (Figura **??**) distingue tres tipos de interfaces: funcionales, de control y de contexto.

Definición de interfaz funcional

Definición 7.1. Interfaz funcional
Una interfaz funcional es un conjunto de operaciones que pueden ser expuestas o invocadas por un componente o servicio.

Definición de interfaz de control

Definición 7.2. Interfaz de control
Una interfaz de control contiene un conjunto de métodos que permiten controlar un componente o servicio y alterar su comportamiento (más detalles en la Sección 7.7).

Definición 7.3. Interfaz de contexto

Definición de interfaz de contexto

Una interfaz de contexto contiene un conjunto de métodos para intercambiar información contextual entre componentes o servicios (más detalles en la Sección 7.8).

En la etapa de agrupamiento de interfaces, la atención se centra en el primer tipo de interfaces; las demás serán abordadas en las siguientes secciones. Las funcionalidades, expresadas en términos de métodos, deben agruparse en interfaces funcionales de acuerdo a algún criterio. Típicamente, esta tarea se realiza siguiendo los principios de la separación de responsabilidades establecida en el paradigma del diseño basado en componentes. Algunos criterios para realizar este agrupamiento son:

Criterios para agrupar funcionalidades en interfaces

- **Agrupamiento dirigido por los datos:** las operaciones que tratan con el mismo conjunto de datos están sujetas a ser agrupadas en la misma interfaz. A menudo involucra la presencia de operaciones CRUD (*Create*, *Retrieve*, *Update*, *Delete*) sobre un cierto tipo de datos.

- **Agrupamiento dirigido por los casos de uso:** las operaciones implicadas en el mismo caso de uso o realizadas por el mismo actor son susceptibles de ser incluidas conjuntamente en la misma interfaz.

- **Agrupamiento dirigido por categorías:** las operaciones con el mismo propósito o intención (por ejemplo, realizar la autenticación de un usuario) pueden resultar incluidas en la misma interfaz.

- **Agrupamiento dirigido por NFR:** las operaciones cuya ejecución está orientada a la mejora de ciertos NFRs (por ejemplo, mejorar la seguridad) son candidatas para ser incluidas en la misma interfaz.

En este punto, los diseñadores tienen un conjunto de interfaces que modelan comportamientos abstractos entre elementos del sistema y expuestos por el mismo.

7.5 Identificación de los puntos de variabilidad

Identificación de los puntos de variabilidad

La heterogeneidad de alternativas, especialmente desde un punto de vista tecnológico, existentes en los entornos ubicuos, cada uno de ellos con diferentes propiedades de calidad, dificulta la adopción de una única opción para realizar ciertas tareas, tales

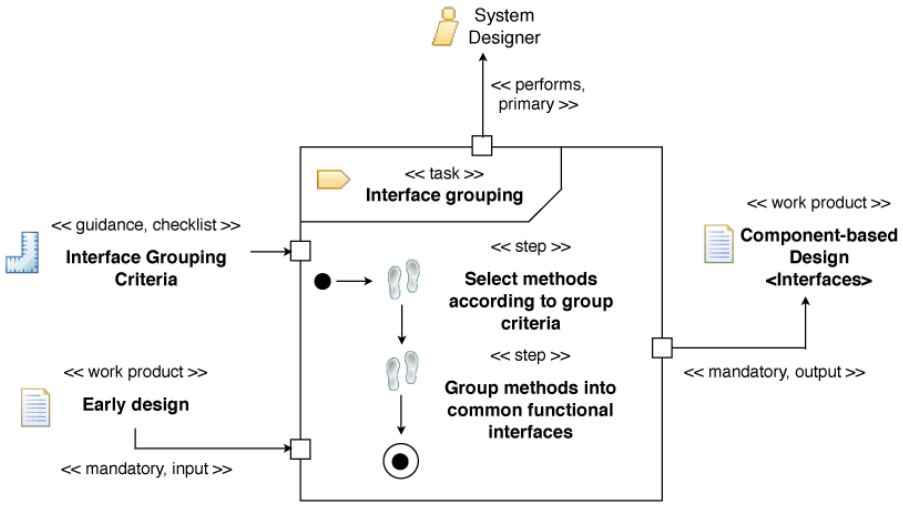

Figura 7.5: Definición de la tarea de Agrupamiento de interfaces en notación SPEM (Véase leyenda en Tabla 4.3)

como medir una señal de un sensor o realizar una acción sobre un actuador.

Además, una característica clave de las aplicaciones ubicuas es la habilidad de reaccionar y adaptarse ante cambios en el entorno. A este respecto, los posibles puntos de variabilidad deben identificarse, los cuales se definen como:

Definición 7.4. Punto de variabilidad
Un punto de variabilidad es un lugar en el diseño donde múltiples alternativas para dar soporte a un conjunto de requisitos, tanto funcionales como no funcionales, son posibles.

Definición de punto de variabilidad

Éste es el propósito de esta tarea; dado un conjunto de interfaces identificadas que modelan comportamientos abstractos, los diseñadores deben seleccionar aquellas que pueden aceptar diferentes implementaciones con ciertas variaciones.

Por ejemplo, asumiendo que hay una interfaz cuyo propósito es proporcionar métodos relacionados con la autenticación de un usuario, hay varios mecanismos de autenticación disponibles, entre los que podemos encontrar desde identificación por NFC (*Near Field Communication*) hasta el clásico mecanismo de autenticación por nombre de usuario y contraseña. Por lo tanto, esta interfaz es un punto de variabilidad potencial.

Hay algunos criterios que pueden ayudar al descubrimiento de los puntos de variabilidad. Un conjunto de posibles guías para su identificación incluye:

Criterios para el descubrimiento de puntos de variabilidad

- **Sensores y actuadores:** los sensores y actuadores son dispo-

sitivos clave en el ámbito de la Computación Ubicua. Dado el amplio número de fabricantes tecnológicos y alternativas existentes que pueden encontrarse, es difícil seleccionar uno. Por tanto, una interfaz que modele el comportamiento de estos dispositivos es un punto de variabilidad potencial.

- **Interacción del usuario:** la Computación Ubicua establece el uso de interfaces naturales para la interacción de usuario. Dependiendo de las circunstancias, numerosas alternativas están disponibles, tales como comprensión y síntesis del lenguaje natural o interacciones multimodales. Las interfaces que modelan capacidades de interacción con el usuario están sujetas a ser seleccionadas como puntos de variabilidad.

- **Presentación de información:** relacionado con el punto anterior, la presentación de información desde el sistema al usuario puede diferir en términos de cantidad, temática o apariencia, entre otros, haciendo de éste otro punto de variabilidad.

- **Almacenamiento de información:** para garantizar la persistencia de la información, deben incorporarse mecanismos de almacenamiento. Dependiendo de las capacidades de almacenamiento de los diferentes dispositivos, hay varias alternativas disponibles, desde almacenamiento en texto plano hasta sistemas de gestión de bases de datos.

- **Obtención de información:** de manera similar al punto anterior, la información puede recuperarse desde diferentes fuentes y de distintas maneras (cantidad, ordenación, relevancia, etc).

- **Comunicación entre servicios:** hay un amplio número de tecnologías de comunicación, protocolos o soluciones middleware, lo cual sugiere que las interfaces relativas a aspectos de la comunicación son puntos de variabilidad potenciales.

7.6 Asignación de componentes

Asignación de componentes

El siguiente paso propuesto por este método consiste en la **asignación** de las interfaces identificadas a los componentes que pueden implementarlas. Así, el metamodelo de SCUBI distingue diferentes tipos de componentes, a saber, funcionales, de control,

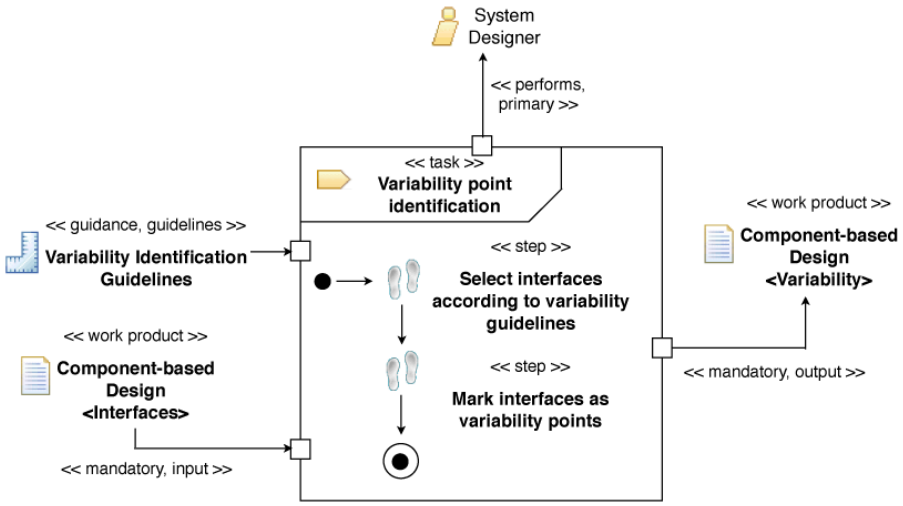

Figura 7.6: Definición de la tarea de Identificación de los puntos de variabilidad en notación SPEM (Véase leyenda en Tabla 4.3)

de contexto, de adaptación y de gestión, además de algunas especializaciones de los mismos.

Definición 7.5. Componente funcional
Un componente funcional es el tipo más básico de componente. Implementa la funcionalidad expresada en los métodos contenidos en las interfaces que proporciona. Pueden ofrecer y demandar varias interfaces funcionales, y proporciona, como máximo, una interfaz de control.

Definición de componente funcional

Definición 7.6. Componente de comunicación
Un componente de comunicación es una especialización de un componente funcional que implementa operaciones mediante la invocación remota de métodos (más detalles en la Sección 7.11).

Definición de componente de comunicación

Definición 7.7. Componente de control
Un componente de control es aquél cuyo propósito es alterar el comportamiento o estructura de otro componente mediante la aplicación de ciertas acciones (más detalles en la Sección 7.7). Ofrece y proporciona interfaces de control.

Definición de componente de control

Definición 7.8. Componente de contexto
Un componente de contexto es aquél cuyo propósito es lidiar con los cambios que ocurren en el contexto. Se distinguen dos tipos de componentes de contexto: proveedores y consumidores (más detalles en la Sección 7.8).

Definición de componente de contexto

Definición de componente proveedor de contexto

Definición 7.9. Componente proveedor de contexto
Un componente proveedor de contexto es aquél que sirve como fuente de información contextual que puede resultar útil para otras entidades del sistema. Demanda interfaces de contexto.

Definición de componente consumidor de contexto

Definición 7.10. Componente consumidor de contexto
Un componente consumidor de contexto es aquél que recibe el consumo de información contextual y emplea estos datos para realizar acciones basadas en ellos. Proporciona interfaces de contexto.

Definición de componente de adaptación

Definición 7.11. Componente de adaptación
Un componente de adaptación es aquél que incorpora el conocimiento necesario sobre la adaptación de un servicio (más detalles en la Sección 7.9). Proporcionan interfaces de contexto y demandan interfaces de control.

Definición de componente de gestion

Definición 7.12. Componente de gestión
Un componente de gestión es una agrupación de componentes de manera que pueda realizarse una selección entre las alternativas disponibles o combinar sus resultados. Proporcionan, a lo sumo, una interfaz de control, y ofrecen o demandan diferentes tipos de interfaces dependiendo de los componentes gestionados.

El foco de atención en esta etapa se centra en los componentes funcionales y de gestión; los demás serán presentados en las siguientes secciones. Los diseñadores deben tomar las interfaces funcionales identificadas y asignarlas a los componentes funcionales que las implementan; es decir, cada uno de los componentes creados proporcionan la interfaz funcional correspondiente. Es importante apuntar que cada interfaz funcional puede asignarse a múltiples componentes funcionales. Los componentes que deben incorporarse al sistema deben ser elegidos por los diseñadores. Si el método REUBI ha sido aplicado previamente en el proceso de desarrollo, el procedimiento de evaluación puede resultar útil para tomar esta decisión.

A continuación, los diseñadores deben considerar los puntos de variabilidad identificados. Para cada interfaz que ha sido marcada como punto de variabilidad, un componente de gestión debe crearse. Este componente de gestión ofrece y demanda la misma interfaz, y actúa como un *proxy* para delegar estas funciones en los componentes funcionales correspondientes. La responsabilidad de los componentes de gestión consiste en seleccionar entre las alternativas o combinar sus resultados, si es posible. Por esta razón, los diseñadores deben decidir cuántos componentes funcionales pueden seleccionarse por el gestor a la misma vez,

así como el procedimiento para combinar sus resultados (por ejemplo, unión de resultados, media, suma, etc.).

Otro aspecto importante que debe considerarse en casos donde pueden seleccionarse múltiples componentes funcionales son las interacciones y efectos laterales que pueden aparecer de su combinación. Sean C_i y C_j dos componentes que pueden combinarse, las interacciones posibles entre ellos son:

Definición 7.13. Interacción complementaria
Los componentes C_i y C_j son complementarios si su aplicación simultánea muestra propiedades positivas adicionales a su aplicación separada.

Definición de Interacción complementaria

$$properties(C_i \cup C_j) > properties(C_i) \cup properties(C_j) \quad (7.1)$$

Definición 7.14. Interacción ortogonal
Los componentes C_i y C_j son ortogonales si su aplicación simultánea no muestra propiedades positivas ni negativas adicionales a su aplicación separada.

Definición de Interacción ortogonal

$$properties(C_i \cup C_j) = properties(C_i) \cup properties(C_j) \quad (7.2)$$

Definición 7.15. Interacción opuesta
Los componentes C_i y C_j son opuestos si su aplicación simultánea muestra propiedades negativas.

Definición de Interacción opuesta

$$properties(C_i \cup C_j) < properties(C_i) \cup properties(C_j) \quad (7.3)$$

Finalmente, dada la estructura recursiva de la gestión de componentes, el metamodelo definido contempla la posibilidad de crear componentes de gestión cuyo propósito sea el de seleccionar otros componentes de gestión. De esta manera, se puede cambiar el protocolo de selección para elegir componentes o la política para combinar sus resultados.

7.7 Definición de mecanismos de control

Definición de control

En este punto, el diseño incluye un conjunto de interfaces que

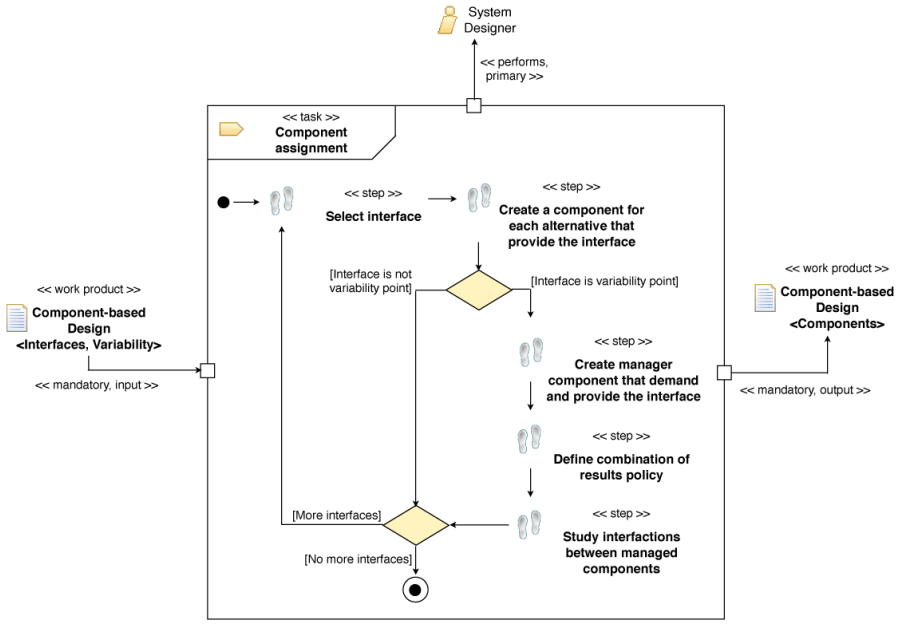

Figura 7.7: Definición de la tarea de Asignación de componentes en notación SPEM (Véase leyenda en Tabla 4.3)

son implementadas por un conjunto de diferentes componentes y gestionados por otros de más alto nivel. El objetivo de esta etapa es incorporar mecanismos de control que puedan alterar el comportamiento de cada uno de estos componentes básicos.

Definición de los mecanismos de control

Para lograr esto, se han de **definir los mecanismos de control**. En un primer paso, los componentes que son susceptibles de ser controlados, es decir, aquéllos cuyo comportamiento o estructura puede modificarse, son seleccionados. Para cada uno de ellos, se define una interfaz de control. Esta interfaz contiene los métodos necesarios para indicar al componente las modificaciones que acepta. Los componentes con la misma interfaz funcional deben compartir la misma interfaz de control para hacer su uso tan transparente como sea posible. Los componentes de gestión deben proporcionar una interfaz de control, mientras que los componentes funcionales podrían o no ofrecerla. En un segundo paso, las interfaces de control deben asignarse a los componentes de control que las implementen.

Controlo por configuración o adaptación

El control puede tener su origen en diferentes fuentes. Por una parte, las personas encargadas de mantener y gestionar el sistema pueden decidir qué componentes funcionales se seleccionan y configurarlos para abordar las necesidades del entorno de despliegue. Por otra parte, el control puede provenir de otras unidades software como respuesta a cambios en el entorno y

la necesidad de adaptarse a ellos para garantizar propiedades de calidad. En las siguientes secciones se profundizará en estos aspectos.

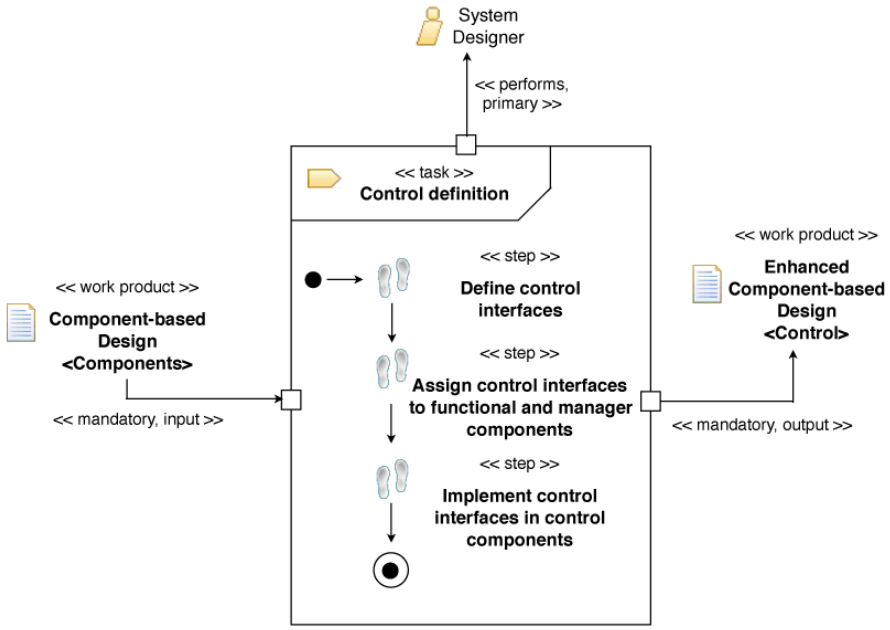

Figura 7.8: Definición de la tarea de Definición de control en notación SPEM (Véase leyenda en Tabla 4.3)

7.8 Definición de aprovisionamiento de contexto

Definición de aprovisionamiento de contexto

Esta etapa trata de incorporar en el diseño la influencia del contexto sobre los servicios, dada su importancia en el paradigma de la Computación Ubicua.

Descubrimiento de información contextual de interés

En un paso inicial, los diseñadores deben identificar qué eventos contextuales son relevantes para los servicios y elaborar un modelo de datos en términos de un diagrama de clases UML. Si se ha aplicado REUBI anteriormente, los diseñadores cuentan con un modelo del contexto inicial que puede resultar útil para realizar esta tarea.

Creación de interfaces

El siguiente paso consiste en la definición de las correspondientes interfaces contextuales donde los eventos del contexto serán notificados. Como regla general, debería crearse una interfaz de contexto por cada evento relevante. Si ciertas entidades siempre responden a los mismos eventos, o hay eventos altamente relacionados, sus interfaces correspondientes pueden ser combinadas en una sola para reducir su número.

Asignación a componentes de control

Finalmente, estas interfaces deben implementarse por un conjunto de componentes. Como se apuntaba en la Sección 7.6, los componentes de contexto pueden ser proveedores o consumidores. Para cada consumidor de contexto, al menos un proveedor debe existir. Los proveedores de contexto demandan una interfaz de contexto, mientras que los consumidores la proporcionan. Aunque pueda parecer contraintuitivo por su nomenclatura, el motivo por el que se toma esta decisión es para que los proveedores puedan invocar a los consumidores haciendo uso de mecanismos de notificación *push*. Si las interfaces se asignan al contrario, generalmente involucraría la adopción de mecanismos *pull* para comprobar si ha habido algún cambio en el contexto, lo cual no es muy eficiente y puede conllevar la pérdida de algunos cambios en el contexto.

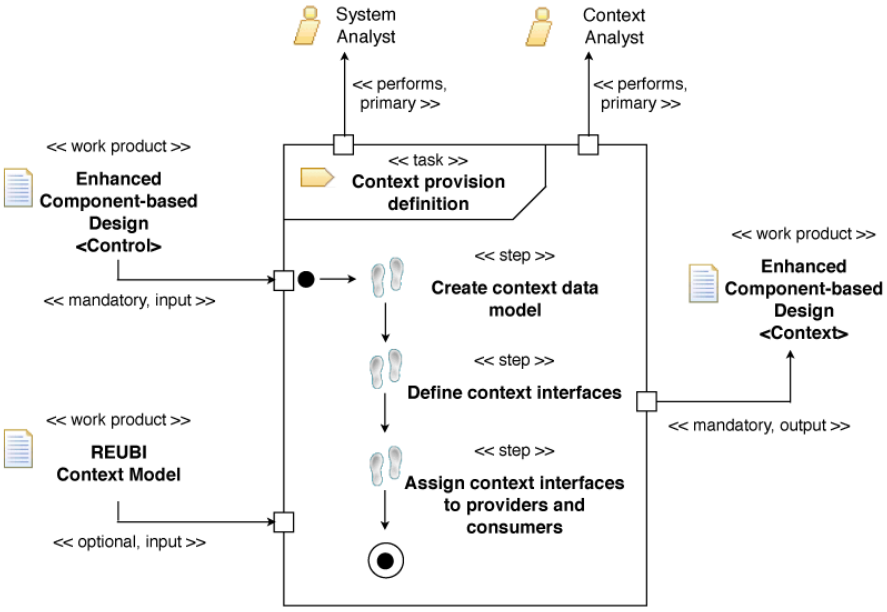

Figura 7.9: Definición de la tarea de Definición de aprovisionamiento de contexto en notación SPEM (Véase leyenda en Tabla 4.3)

7.9 Definición de la adaptación

Definición de la adaptación

La adaptación de software en Sistemas Ubicuos consta típicamente de tres pasos: obtener información de contexto, razonar sobre la adaptación que es necesario llevar a cabo y realizar las acciones apropiadas para modificar la estructura o el comportamiento del sistema, si hay alguna que deba aplicarse. En las secciones anteriores se ha considerado cómo se adquiere el

contexto mediante componentes de contexto y cómo se aplican las modificaciones en el comportamiento o estructura mediante componentes de control.

En esta etapa deben incorporarse los componentes de adaptación. Un componente de adaptación proporciona, al menos, una interfaz de contexto donde se le notifica cuando un evento de contexto relevante ha ocurrido. También, demanda, al menos, una interfaz de control donde puede enviar un mensaje para indicar las acciones que deben realizarse cuando se inicia una adaptación.

El diseño de este componente depende de la complejidad del sistema. Los diseñadores deben decidir qué técnicas deben usarse entre las que están disponibles, tal como se mostró en la Sección 4.5.1. Cada una de ellas presenta diferentes ventajas e inconvenientes; por lo tanto, debe realizarse un estudio cuidadoso antes de tomar esta decisión.

También, el diseñador ha de considerar la granularidad de la adaptación. El método SCUBI distingue dos tipos de adaptaciones, de granularidad fina y de granularidad gruesa, las cuales se aplican a través de las interfaces de control.

Definición 7.16. **Adaptación de granularidad fina**
Una adaptación de grano fino es aquélla que se realiza sobre un componente funcional, dado que implica una modificación concreta en un aspecto determinado de un componente.

Definición de adaptación de granularidad fina

Definición 7.17. **Adaptación de granularidad gruesa**
Una adaptación de granularidad fina es aquélla que se realiza sobre un componente de gestión, dado que implica la adición, eliminación o sustitución de otros componentes.

Definición de adaptación de granularidad gruesa

Tiempo de adaptación

La adaptación puede ocurrir en momentos diferentes del ciclo de vida del software. SCUBI permite la posibilidad de adaptar en tiempo de diseño y despliegue, gracias al uso de interfaces compatibles, pero también en tiempo de ejecución, gracias a la incorporación de los componentes de adaptación.

Reglas de adaptación

Finalmente, las adaptaciones se especifican en términos de los eventos que las disparan, las condiciones que deben cumplirse para ello, y las acciones que deben tomarse para realizar la adaptación. Si se ha aplicado el método REUBI previamente, los diseñadores tienen información sobre las variaciones que son posibles, cuándo deben aplicarse y cómo su aplicación cambia el sistema. Tener esta información en cuenta facilita la definición de las adaptaciones.

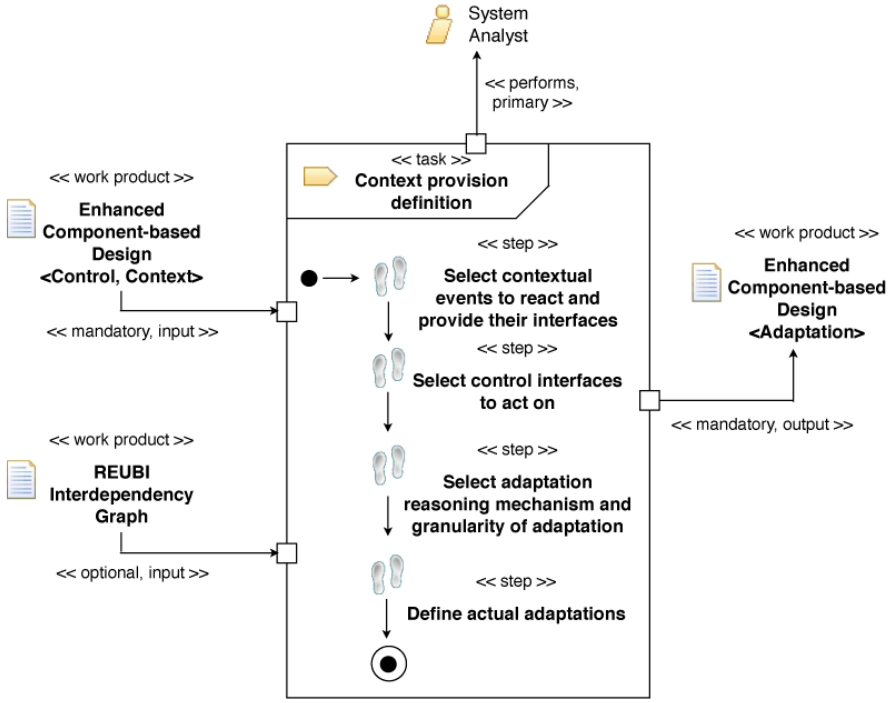

Figura 7.10: Definición de la tarea de Definición de adaptación en notación SPEM (Véase leyenda en Tabla 4.3)

7.10 Ensamblado de servicios

Ensamblado de servicions

Cuando los componentes básicos han sido identificados, deben ensamblarse para conformar los servicios correspondientes. Múltiples ensamblados diferentes pueden ser posibles; por esta razón, los diseñadores deben tener en cuenta las características del paradigma de la Arquitectura Orientada a Servicios, tales como separación de responsabilidades, bajo acoplamiento o alta reusabilidad. Algunos de los criterios expuestos en la Sección 7.4 pueden emplearse en esta etapa para agrupar los componentes en servicios.

Especificación de las interfaces del servicio

Una vez que los servicios han sido identificados y construidos en términos de diferentes componentes, sus interfaces deben especificarse. Los servicios en SCUBI pueden tener tres tipos de interfaces, tal como se definió en la Sección 7.4. Los diseñadores deben decidir cuál de estas interfaces internas proporcionadas por los componentes van a ser expuestas como interfaces del servicio, sean funcionales, de control o de contexto. En el caso más simple, la interfaz del servicio es la unión de todas las interfaces proporcionadas por cada componente.

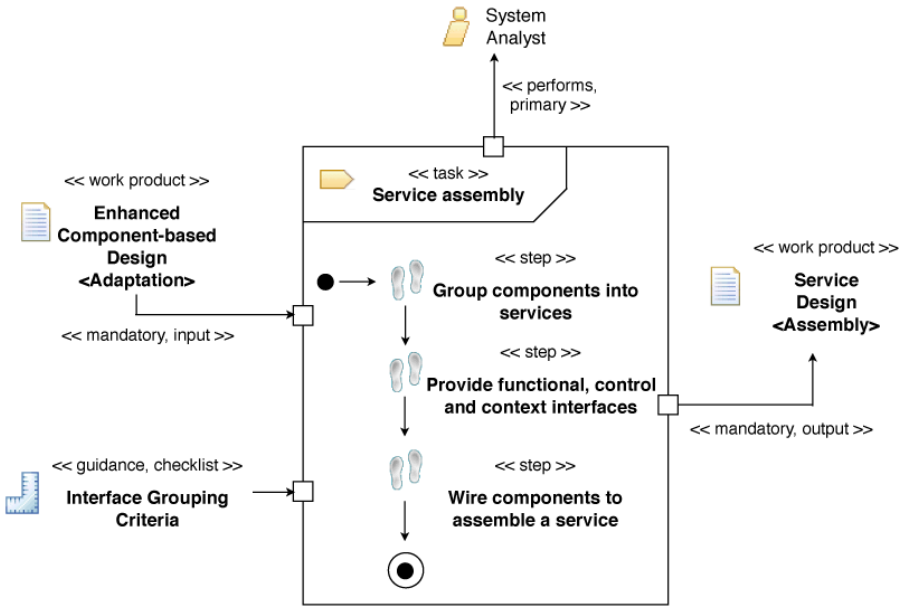

Figura 7.11: Definición de la tarea de Ensamblado de servicios en notación SPEM (Véase leyenda en Tabla 4.3)

7.11 Comunicación entre servicios

Comunicación entre servicios

Finalmente, cuando los servicios han sido identificados, las dependencias entre ellos han de gestionarse. Los servicios tienen que comunicarse y coordinarse unos con otros para realizar algunas tareas. Para este propósito, pueden introducirse componentes de comunicación, tal como se describieron en la Sección 7.6, para tratar con los aspectos de descubrimiento, coordinación y comunicación entre servicios.

Algunas decisiones importantes que deben tomarse en esta etapa son:

Decisiones a tomar para determinar la comunicación entre servicios

- **Mecanismo de coordinación:** los diseñadores deben decidir cómo se realizará la coordinación entre servicios. Entre las opciones disponibles, destacan la orquestación y la coreografía de servicios.
- **Estilo de la arquitectura:** el descubrimiento de los servicios depende del estilo de arquitectura que se decida para SOA. Varias alternativas son posibles, tales como *matchmaking*, enfoques basados en *broker* o *peer-to-peer*.

En cualquier caso, estas decisiones pueden ser encapsuladas en los componentes de comunicación, los cuales pueden ser fácilmente reemplazados por otros si los requisitos cambian.

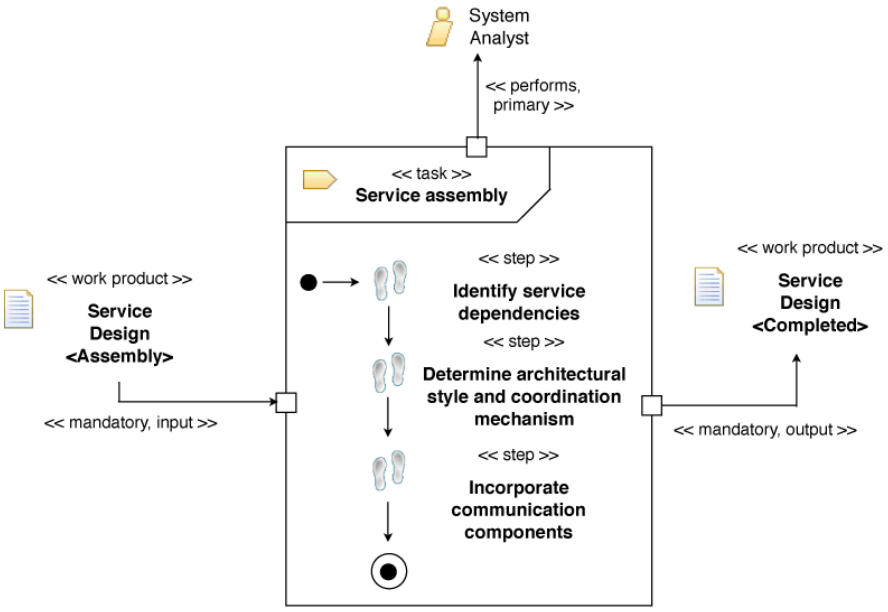

Figura 7.12: Definición de la tarea de Comunicación entre servicios en notación SPEM (Véase leyenda en Tabla 4.3)

7.12 Resumen

En este capítulo se ha presentado un método de Diseño de Servicios basados en Componentes para Sistemas Ubicuos. El objetivo de este método es dar un soporte preciso y sistemático al diseño y desarrollo de este tipo de sistemas, incorporando las características principales de los Sistemas Ubicuos y potenciando principios importantes de la Ingeniería del Software, tales como la reutilización o la separación de aspectos. A modo de resumen, las principales contribuciones SCUBI son:

- Derivar un diseño que garantiza las propiedades de calidad a partir de la información obtenida durante la etapa de Ingeniería de Requisitos.
- Proponer un metamodelo que indica cómo deben modularizarse los servicios en diferentes componentes, cada uno con responsabilidades concretas y bien definidas.
- Encapsular decisiones de diseño en componentes que pueden ser añadidos, eliminados o intercambiados de manera sencilla y con un impacto mínimo en el resto del sistema.
- Establecer mecanismos de control de la configuración de los componentes y servicios, permitiendo realizar cambios a

nivel funcional (operaciones realizadas) como no funcional (propiedades de calidad vinculadas al software).

- Incorporar estructuras que permiten la reorganización del software, incluso en tiempo de ejecución, adaptándose a cambios ocurridos en el contexto.
- Incorporar la presencia de tres interfaces diferentes para un servicio, encargadas de la funcionalidad, el control y los cambios según el contexto, respectivamente.
- Promover la reutilización de artefactos tanto a nivel de componentes como a nivel de servicios.

Aunque SCUBI ha sido diseñado para ser aplicado tras obtener los resultados del método REUBI, puede emplearse de manera complementaria a otras técnicas de Ingeniería de Requisitos existentes, aunque no se obtengan los mismos beneficios que se obtienen empleando REUBI. Asimismo, SCUBI permite la incorporación de variaciones para ajustarse mejor a las necesidades particulares de cada proyecto, gracias a que ha sido definido conforme al estándar SPEM 2.0.

8 MDUBI: ENFOQUE DIRIGIDO POR MODELOS PARA EL DESARROLLO DE SISTEMAS UBICUOS

Índice

8.1 Introducción

Una vez presentados los metamodelos y métodos para Ingeniería de Requisitos y Diseño para Sistemas Ubicuos, REUBI y SCUBI, en este capítulo se presenta una metodología general que integra ambos enfoques. Esta metodología, a la que hemos denominado MDUBI, consiste en un enfoque dirigido por modelos (MDA) para el desarrollo de servicios para Sistemas Ubicuos cumpliendo con propiedades de calidad. Así pues, esta parte del trabajo de investigación de la propuesta intenta contribuir hacia una metodología completa para el desarrollo de sistemas ubicuos, es decir, cubrir con todas las etapas del ciclo de vida del software. No obstante, por el momento esta metodología se centra en las etapas de Ingeniería de Requisitos y Diseño de Servicios Software de soporte al desarrollo de aplicaciones software en Sistemas Ubicuos, por entender que son las etapas que pueden dar lugar a la generación de un mayor número de artefactos comunes (modelos, servicios, etc.) y más fácilmente reutilizables entre Sistemas Ubicuos.

Inspirado por el Proceso Unificado de desarrollo de software, esta metodología consta de cuatro fases principales, tal como se detalla en el diagrama de componentes de proceso de la Figura 8.1. Cada fase se representa por un componente con sus puertos de entrada y salida. Estos puertos representan los artefactos que se requieren o se proporcionan en cada fase. Las fases principales son las siguientes:

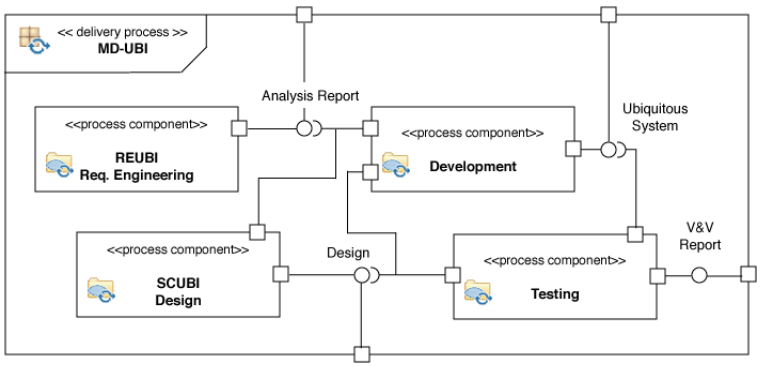

Figura 8.1: Componentes de la metodología MDUBI en notación SPEM.

Fases de la metodología

- **Ingeniería de Requisitos:** es el punto de inicio de la metodología. Consiste en un método para especificar los requisitos del sistema en desarrollo (REUBI), prestando especial atención al tratamiento de los Requisitos No Funcionales, dado que están altamente relacionados con la calidad del sistema final, y tiene en cuenta las características especiales de los Sistemas Ubicuos. Cuando la fase de Ingeniería de Requisitos ha sido completada, se crea un documento de análisis de los requisitos. Este artefacto contiene un conjunto de modelos, como se describió en el Capítulo 5, que son útiles para los diseñadores.

- **Diseño:** siguiendo un enfoque dirigido por modelos, se realizan un conjunto de transformaciones sobre los modelos de la etapa anterior para obtener un diseño preliminar. Esta etapa aplica el método SCUBI, descrito en el Capítulo 7 para guiar en la transformación de modelos y en la incorporación de nueva información.

- **Desarrollo:** el documento de diseño creado en la fase anterior se toma en esta etapa y, tras la selección de una plataforma concreta, los modelos que contiene se transforman a un modelo conforme a la plataforma de destino. Hay que hacer notar que estas transformaciones pueden

requerir que el desarrollador tenga que incorporar nueva información para enriquecer los modelos. Posteriormente, pueden aplicarse estrategias de generación de código a los modelos resultantes para obtener código libre de errores para el sistema. El desarrollador podría necesitar incorporar o modificar parte del código tras su generación.

- **Verificación y Validación:** finalmente, cuando el sistema software ha sido implementado completamente, debe ser verificado y validado para garantizar que los requisitos de calidad que fueron especificados inicialmente se han cumplido a través del desarrollo. Como resultado de esta fase se genera un informe de verificación y validación.

Niveles MDA en MDUBI

Como se ha comentado, la propuesta se basa en MDA y las etapas corresponden a los tres niveles propuestos por el estándar. De hecho, los modelos que se crean durante la fase de Ingeniería de Requisitos corresponden a un modelo independiente de la computación (CIM) del sistema bajo análisis. A continuación, estos modelos pueden transformarse en modelos independientes de plataforma (PIM) en la fase de diseño. Debemos hacer hincapié en que puede que se obtengan varios PIM en esta etapa mientras se refinan hasta que se alcanza un diseño detallado. A continuación, cuando la plataforma de ejecución se haya elegido, estos PIM pueden transformarse en modelos específicos de plataforma (PSM) a los que eventualmente se le aplicarán técnicas de generación de código para dicha plataforma. Esto permite que, desde un mismo diseño, se obtengan sistemas para diferentes plataformas, promoviendo la reutilización de los modelos de diseño creados. A diferencia de otras propuestas existentes en la bibliografía, mostradas en el Capítulo 4, MDUBI comienza en el nivel CIM.

Todos los modelos creados a diferentes niveles (CIM, PIM, PSM) son conformes a sus correspondientes metamodelos, lo que posibilita la especificación de transformaciones siguiendo un enfoque basado en metamodelos, como se detallará en las siguientes secciones.

8.2 De Modelo de Valores a Grafo de Interdependencia

En el capítulo 5, el método REUBI sugiere el comienzo de esta fase con la elaboración de un Modelo de Valores en el que se realiza una captura inicial de los requisitos. También se descri-

bió la conveniencia de llevar a cabo una correspondencia entre los elementos de este Modelo de Valores y los elementos del Grafo de Interdependencia. En esta sección se especifica dicha correspondencia.

Transformaciones de Modelo de Valores a Grafo de Interdependencia

Los elementos del Modelo de Valores que se transforman según esta correspondencia son los Valores y los Potenciadores de Valores (Figura 8.2). De acuerdo con lo expuesto en el capítulo 5, un valor se transformará en un *goal*, dado que el sistema que se está analizando tiene como objetivo proporcionar dicho valor. Igualmente, los potenciadores de valores son transformados en *softgoals*, dado que el sistema tiene como objetivo potenciar (i.e. mejorar la calidad del valor intercambiado), pero puede no existir un criterio unívoco para determinar si esto está logrado o no. En ambos casos la correspondencia es uno a uno y de manera directa, sin tener en cuenta otras entidades.

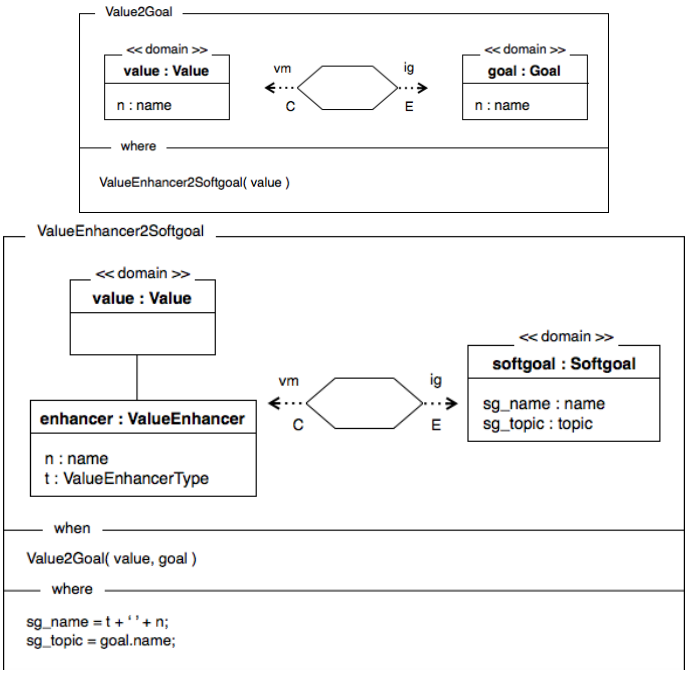

Figura 8.2: Definición de las transformaciones correspondientes a valores y potenciadores.

8.3 De REUBI a SCUBI

Tras la realización de la primera transformación, MDUBI propone continuar con la aplicación del método REUBI para elaborar el Grafo de Interdependencia mediante el refinamiento de los ob-

jetivos, la incorporación de operacionalizaciones, el estudio de la influencia del contexto en los objetivos, la priorización y la selección de las opciones más adecuadas de acuerdo al procedimiento de evaluación.

Transformaciones de REUBI a SCUBI

Una vez obtenido y validado el Grafo de Interdependencia, MDUBI propone la transformación de este modelo en un modelo de diseño de acuerdo al metamodelo de SCUBI. En las siguientes secciones se detallan las transformaciones correspondientes a cada entidad del metamodelo de REUBI y sus correspondientes resultados en el metamodelo de SCUBI. Es importante destacar que no todos los elementos del modelo REUBI se transforman en entidades de SCUBI puesto que el modelo de requisitos se encuentra en el nivel CIM y algunas de las entidades no son computables. También hay que hacer notar que, tras la ejecución del procedimiento de evaluación, pueden existir elementos (e.g. operacionalizaciones) que hayan sido descartados por el procedimiento de evaluación de REUBI, por lo que estos tampoco son transformados.

8.3.1 De Operacionalización a Componente Funcional

Transformación de Operacionalización a Componente Funcional

Cada operacionalización corresponde a una decisión que se toma para contribuir a la satisfacción de un objetivo. De esta forma, y tratando de potenciar la separación de responsabilidades, una operacionalización se transformará en un componente funcional que implementa dicha decisión.

Este tipo de componentes son los más básicos y deben implementar, al menos, una interfaz funcional. Para ello, deben examinarse las contribuciones positivas que realiza la operacionalización a los *goals*. Cada uno de estos *goals* a los que contribuye la operacionalización dará lugar a una interfaz funcional, tal como se indicará en la Sección 8.3.3. El componente funcional resultante deberá implementar estas interfaces para contribuir a satisfacer el *goal* al que contribuye (Figura 8.3).

Transformación de Grupo a Componente de gestión

Un caso particular de operacionalización es el grupo de operacionalizaciones. En este caso, cuando contamos con un grupo de operacionalizaciones, cada una de las que componen el grupo se transforma en un componente funcional como se ha especificado anteriormente. Además, el grupo se transforma en un componente de gestión que las agrega y gestiona. Puesto que la contribución a los objetivos se realiza desde el grupo, dicho componente de gestión deberá implementar las interfaces funcionales correspondientes, así como cada uno de los componentes que gestiona. Dentro de este componente de gestión se inclui-

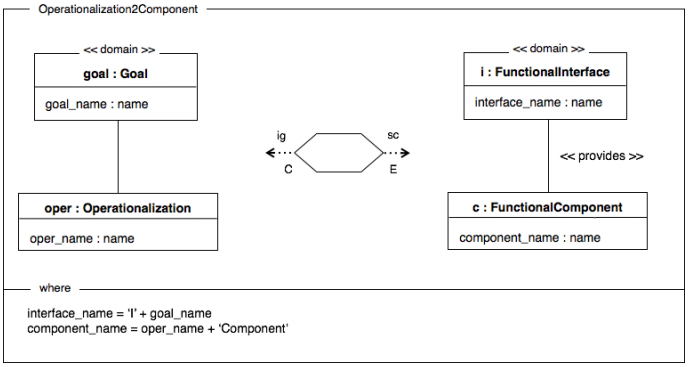

Figura 8.3: Definición de la transformación de una operacionalización.

rá la política que se decida para la combinación de resultados proporcionados por los componentes individuales (Figura 8.4).

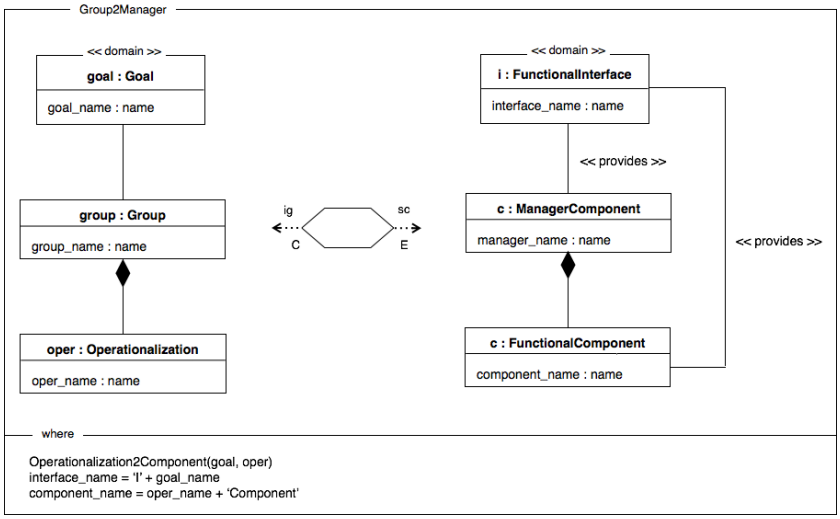

Figura 8.4: Definición de la transformación de un grupo de operacionalizaciones.

8.3.2 De Recurso a Clase

Transformación de Recurso a Clase

Un recurso es una pieza de información relevante para la consecución de un objetivo, o que resulta de su realización. Para poder manejar esta información, un recurso se transformará en un tipo de datos representado por una clase. Dado que un recurso puede tener atributos, cada atributo de un recurso se transformará en una variable de instancia de la clase para mantener dicha información.

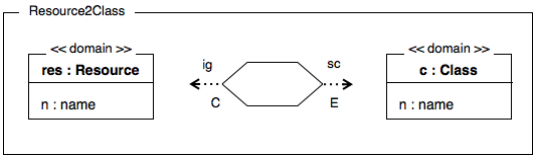

Figura 8.5: Definición de la transformación correspondiente a un recurso.

8.3.3 De *Goal* a Componente de gestión

Transformación de Goal a Componente de gestión

Tipos de transformaciones según el objetivo

Cuando se tiene una instancia de un *goal*, pueden darse 3 casos diferentes:

- El *goal* es de granularidad muy fina; es decir, no se descompone en otros objetivos más pequeños, sino que recibe contribuciones de operacionalizaciones.
- El *goal* es de granularidad media, es el resultado de descomponer un objetivo y a su vez puede descomponerse en subobjetivos.
- El *goal* es el de más alto nivel; se descompone en subobjetivos y no es resultado de la descomposición de otro objetivo.

En todos estos casos se encuentra una circunstancia común. Cuando este objetivo se vaya a realizar en un diseño, se puede encapsular en un componente que gestione a otros que realizan otras tareas y los orqueste. Así, en el primer caso, dado que las operacionalizaciones se transforman en componentes funcionales y pueden variar a lo largo de la ejecución del sistema, este componente deberá gestionar cada una de ellas. En los dos últimos casos, el componente gestionará a otros que están encargados de cumplir los subobjetivos en los que se ha dividido el *goal*. En todos estos casos, un *goal* da lugar a los siguientes elementos:

- **Interfaz funcional:** es una interfaz que contiene una única operación (realizar el objetivo). Es la vía de comunicación entre el *manager* y sus componentes internos.
- **Componente de gestión** es un componente encargado de gestionar los componentes resultantes de los subobjetivos del *goal* o de los componentes resultantes de las operacionalizaciones que contribuyen a su satisfacción.

Por otra parte, un *goal* puede demandar o proporcionar recursos. En este caso, se deben recuperar los recursos vinculados al *goal* y relacionarlos con la interfaz funcional resultante de su

transformación. En particular, la interfaz funcional contendrá un método (realizar el objetivo) que tiene como parámetros las clases resultantes de transformar los recursos que el *goal* demanda. Asimismo, dicho método devolverá un valor correspondiente a la clase resultante de transformar el recurso que proporciona el *goal*. Si un *goal* no demanda ni proporciona recursos, el método no tendrá parámetros ni valor de retorno.

Adicionalmente, dado que el resultado es un componente *manager*, está sujeto a que tanto un operador humano como un agente software ejecute acciones de reconfiguración de los componentes (e.g. por un proceso de adaptación). Por este motivo, junto al componente *manager*, se crean los siguientes artefactos:

- **Interfaz de control:** cuenta con un método para indicar la acción de control que se va a llevar a cabo.
- **Componente de control:** implementa la interfaz de control.
- **Componente de adaptación:** determina, en función de los cambios contextuales, cuáles son las acciones que se deben llevar a cabo.

Por último, cuando el *goal* que se va a transformar es de alto nivel, adicionalmente se crea un servicio. Dicho servicio aglutina todos los componentes creados en su jerarquía y expone sus interfaces, agrupándolas en tres: funcional, de control y de contexto (Figura 8.6).

8.3.4 De Datos de contexto a Componente de Contexto

Transformación de datos de contexto a componente de contexto

Para poder incorporar mecanismos de consciencia del contexto, es necesario crear los componentes y tipos de datos necesarios que los puedan manejar. Así, para cada tipo de datos de contexto que se hayan definido en el Grafo de Interdependencia, se generan los siguientes elementos de SCUBI (Figura 8.7):

- **Clase:** se crea una clase para manejar el evento de un cambio en el valor de dicho tipo de datos de contexto.
- **Interfaz de contexto:** se crea una interfaz para notificar la ocurrencia de un evento relacionado con el tipo de datos.
- **Componente consumidor de contexto:** se crea un componente que implementa la interfaz anterior y es el encargado de recibir dicho evento y transmitirlo a otros componentes para, por ejemplo, iniciar un proceso de adaptación.

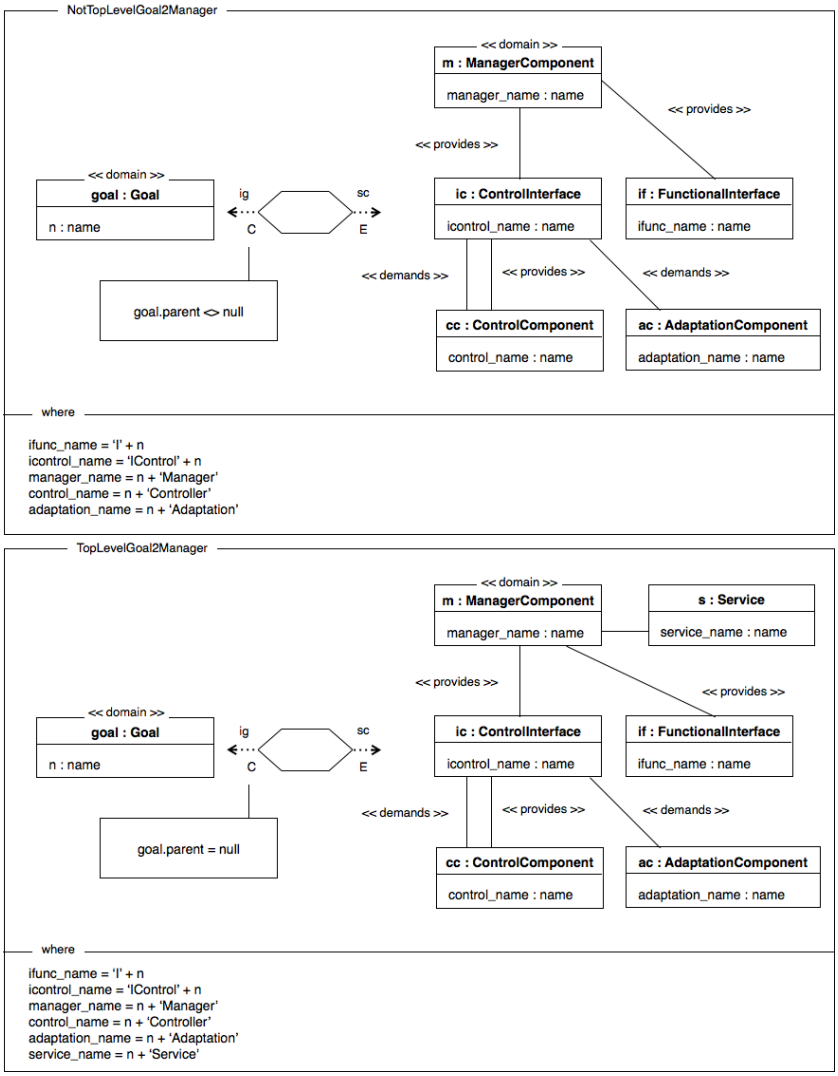

Figura 8.6: Definición de la transformación correspondiente a un *goal*.

- **Componente proveedor de contexto:** se crea un componente que proporciona el evento generado cuando ocurre un cambio en el contexto.

8.3.5 De Contribución a Regla de Adaptación

Transformación de contribución a regla de adaptación

Por último, cada contribución positiva de una operacionalización a un objetivo puede estar condicionada por una situación de contexto. Esto implica que la operacionalización correspondiente solamente es aplicable cuando ocurre dicha situación de contexto. Para incorporar esto al modelo de diseño, una contribución positi-

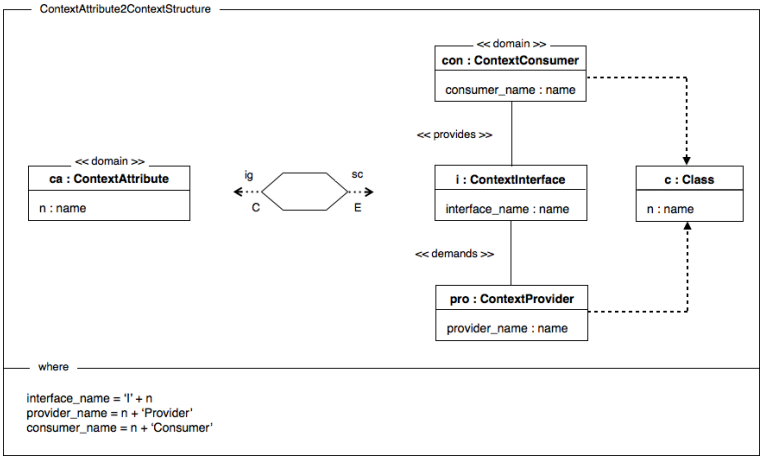

Figura 8.7: Definición de la transformación correspondiente a datos de contexto.

va de una operacionalización a un objetivo condicionada por una situación de contexto se transforma en una regla de adaptación. Esta regla de adaptación tiene tres partes:

- **Eventos:** los eventos de esta regla vienen dados por los tipos de datos de contexto presentes en la situación de contexto, transformados de acuerdo a lo expuesto en la sección 8.3.4.
- **Condiciones:** las condiciones vienen dadas por los atributos de la situación de contexto y sus correspondientes valores.
- **Acciones:** las acciones corresponden a la aplicación de los componentes resultantes de transformar las operacionalizaciones de acuerdo a lo expuesto en la sección 8.3.1.

Dichas reglas de transformación son incorporadas a los componentes de adaptación creados en la transformación indicada en la sección 8.3.3 (Figura 8.8).

8.4 De SCUBI a PSM

Transformación de SCUBI a PSM

Tras realizar la transformación de REUBI a SCUBI, le obtendrá un diseño preliminar en el que existen diversos servicios formados por diferentes componentes. El diseñador deberá refinarlo e incorporar aquellos aspectos que considere relevantes siguiendo el método SCUBI, propuesto en el capítulo 7. De esta manera, se obtendrá un diseño más detallado del sistema a nivel PIM.

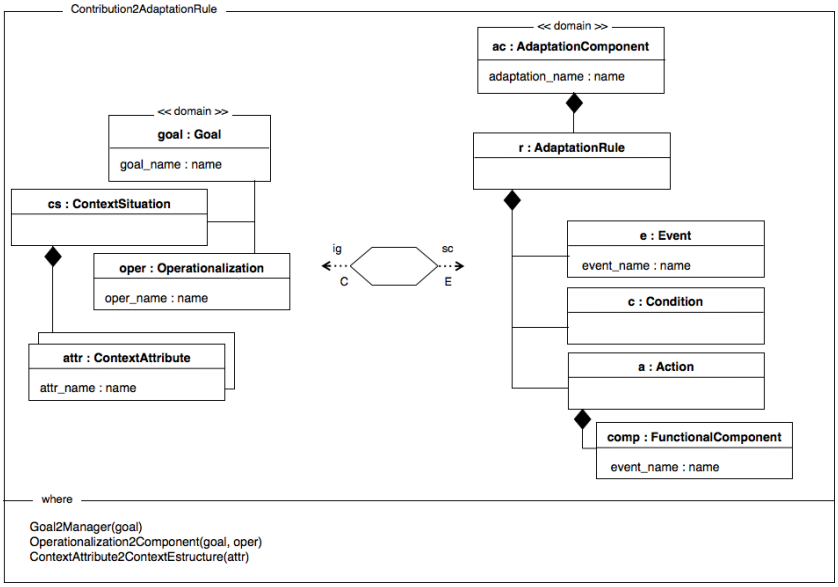

Figura 8.8: Definición de la transformación correspondiente a una contribución positiva influenciada por una situación de contexto.

Tal como se expuso en el capítulo 4, uno de los beneficios que tiene MDA es que los modelos pueden transformarse de diferentes modos para pasar al siguiente nivel. En este caso, debemos transformar un modelo conforme al metamodelo de SCUBI en un modelo específico de plataforma que contenga los detalles de la implementación en una plataforma concreta. En esta sección se detalla cómo se realiza esta correspondencia con dos PSM distintos: el metamodelo de Java y el de WSDL.

8.4.1 De SCUBI al Metamodelo de Java

Transformación de SCUBI a Java

Las transformaciones necesarias para convertir los elementos de SCUBI en entidades del metamodelo de Java son sencillas y directas dada la simplicidad del metamodelo de Java. En este caso, hemos de tener en cuenta que los principales elementos de éste que son relevantes son las interfaces, las clases, los métodos y las variables de instancia. De esta forma, la correspondencia entre elementos es:

- **Interfaces:** las interfaces funcionales, de control y de contexto se transforman en interfaces Java. Los métodos que contienen se transforman en métodos de instancia de visibilidad pública contenidos en dichas interfaces.

- **Clases:** las clases y componentes presentes en SCUBI se transforman en clases Java. Las interfaces que proporcionan dichos componentes son implementadas por las clases, mientras que las interfaces que demandan son transformadas en variables de instancia de las clases. Si dichas las clases o componentes de SCUBI presentan variables o métodos de instancia, éstos son transformados en variables o métodos de instancia en Java.

8.4.2 De SCUBI al Metamodelo de WSDL

Transformación de SCUBI a WSDL

En el caso de esta transformación, no todos los elementos de SCUBI se transforman a WSDL. Dado que WSDL se emplea para definir la interfaz de los servicios, los elementos que se transformarán son las interfaces expuestas por los servicios. Teniendo en cuenta que cada servicio proporciona tres interfaces (funcional, de control y de contexto), se crearán tres servicios WSDL, uno por cada tipo de interfaz. La correspondencia entre elementos de SCUBI y WSDL es:

- ***Message*:** para cada método existente en una interfaz en SCUBI, se crearán dos *message*, uno para realizar la petición (*request*) y otro para devolver el resultado (*response*). Dentro de estos mensajes se definirán los tipos de datos de los parámetros que tenga cada mensaje.

- ***PortType*:** para cada interfaz, se definirá un *port type* que tendrá tantas *operation* como métodos tenga la interfaz. Dichas *operation* tendrán elementos *input* y *output* correspondientes a los mensajes *request* y *response* definidos en el punto anterior, respectivamente.

- ***Binding*:** para cada interfaz, se definirá un *binding* que tendrá tantas *operation* como métodos tenga la interfaz. Dichas *operation* tendrán elementos *SOAP operation* con la url donde se puede realizar la llamada a la operación correspondiente.

- ***Service*:** para cada interfaz, se definirá un *service* que tendrá tantos *port* como métodos tenga la interfaz. Dichos *port* tendrán una referencia al *binding* correspondiente y contendrán la url del servicio.

8.5 Implementación de MDUBI

Los metamodelos y transformaciones propuestas han sido validados mediante su implementación haciendo uso de las herramientas que proporciona el entorno de desarrollo integrado Eclipse en su *framework* para transformación entre modelos y de modelos a texto.

Uso de EMF para implementar metamodelos

Para la implementación de los metamodelos del modelo de valores, REUBI y SCUBI, se ha empleado el *Eclipse Modeling Framework* (EMF). EMF se proporciona como un *plug-in* que puede ser instalado en Eclipse y proporciona un editor gráfico para la creación de los metamodelos en términos de metaclases y relaciones entre ellas. El metamodelo se almacena en formato `.ecore` basado en XML (Figura 8.9).

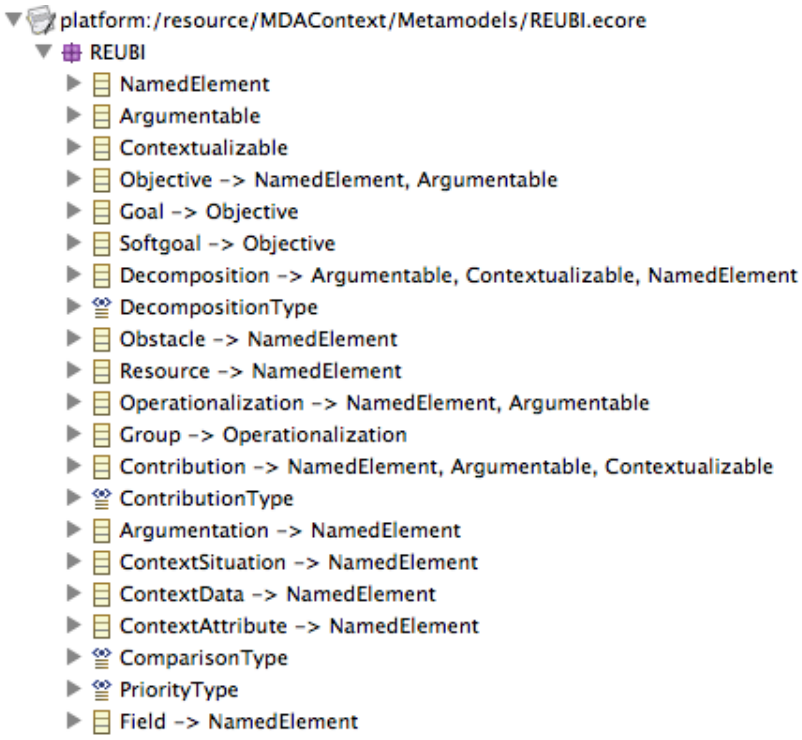

Figura 8.9: Definición del metamodelo de REUBI empleando las herramientas de EMF.

Uso de XML para implementar modelos

La definición de los modelos que conforman los metamodelos creados se realiza mediante la escritura de ficheros XML, empleando como etiquetas los nombres de las metaclases definidas en el metamodelo, y explicitando las relaciones entre clases a través de referencias a claves (Figura 8.10).

Uso de ATL para implementar reglas de transformación

Para la realización de las transformaciones entre modelos se

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.0"
        xmlns:xmi="http://www.omg.org/XMI"
        xmlns:reubi="http://www.ugr.es/~tomruiz/REUBI">

  <reubi:Goal name="Positioning"/>
  <reubi:Goal name="IndoorPositioning"/>
  <reubi:Goal name="OutdoorPositioning" provides="Location"/>
  <reubi:Goal name="MeasureSignal" provides="Fingerprint"/>
  <reubi:Goal name="EstimateLocation" demands="Fingerprint" provides="Location"/>

  <reubi:Decomposition parent="Positioning"
                       children="IndoorPositioning OutdoorPositioning" type="OR"/>
  <reubi:Decomposition parent="IndoorPositioning"
                       children="MeasureSignal EstimateLocation" type="AND"/>

  <reubi:Operationalization name="WiFi"/>
  <reubi:Operationalization name="ZigBee"/>
  <reubi:Operationalization name="Bluetooth"/>
  <reubi:Group name="AllTech" contains="WiFi ZigBee Bluetooth"/>

  <reubi:Operationalization name="GPS"/>
  <reubi:Operationalization name="kNN"/>
  <reubi:Operationalization name="NeuralNetwork"/>
  <reubi:Operationalization name="Probabilistic"/>

  <reubi:Resource name="Fingerprint" attributes="measurements"/>
  <reubi:Field name="measurements" type="Measurement"/>

  <reubi:Resource name="Measurement" attributes="value feature"/>
  <reubi:Field name="value" plaintype="String"/>
  <reubi:Field name="feature" plaintype="String"/>
```

Figura 8.10: Definición de un modelo de requisitos según REUBI en formato XML.

ha empleado el lenguaje de transformación ATL. ATL se proporciona también como *plug-in* de Eclipse y cuenta con un editor con resaltado de sintaxis, un compilador, un intérprete y un depurador. En este lenguaje se han definido diferentes reglas de transformación que relacionan elementos en los metamodelos de origen y destino (Figura 8.11). Para ejecutar las transformaciones, el *script* de ATL hace uso de los ficheros `.ecore` que definen los metamodelos de origen y destino, así como el fichero XML que contiene el modelo que se ha de transformar. Como resultado, se genera un modelo que se almacena en XML. La implementación de las reglas de transformación puede encontrarse en el Anexo B.

Uso de Acceleo para la generación de código

Por último, una vez obtenidos los modelos conforme a los metamodelos de Java y WSDL, se han aplicado técnicas de generación de código para obtener el resultado final de las transformaciones. Para este propósito, se ha empleado el *plug-in* Acceleo, también disponible para Eclipse. En este caso, Acceleo permite la generación de código mediante la definición de plantillas que son rellenadas tomando información del fichero XML que contiene el modelo (Figura 8.12). Para ejecutar el *script* de generación de código, se necesita conocer el fichero `.ecore` con la definición del metamodelo de origen y el fichero XML con el contenido del modelo. Como resultado, se obtiene un conjunto de ficheros

```
rule Operationalization2FunctionalComponent {
    from
        oper : REUBI!Operationalization(not oper.oclIsKindOf(REUBI!Group))
    using{
        contributions : Sequence(REUBI!Contribution) = oper.contributesTo->
                select(o | (o.type = #MAKE or o.type = #HELP));
        goals : Sequence(REUBI!Goal) = contributions->collect(c | c.target)->asSet();
        groups : Sequence(REUBI!Group) = oper.containedBy;
    }
    to
        component : SCUBI!FunctionalComponent(
            name <- oper.name + 'Component',
            provides <- Set{goals->collect(g | thisModule.resolveTemp(g, 'interface'))->asSet(),
                            groups->collect(g | thisModule.resolveTemp(g, 'component').provides)
                            ->flatten()->asSet()}->flatten()->asSet()
        )
}
```

Figura 8.11: Definición de una regla de transformación empleando el lenguaje ATL.

(en Java, WSDL o el formato elegido) que contienen el código generado.

```
[comment encoding = UTF-8 /]
[module Class2Code('http://www.ugr.es/~tomruiz/SCUBI')]


[template public generateClass(aClass : Class)]

[file('datatypes/' + aClass.name + '.java', false, 'UTF-8')]

package edu.ugr.mdubi.datatypes;

public class [aClass.name/]{
    [for(field : Field | aClass.attributes)]
    protected [field.class.name/] [field.name/] = null;
    [/for]

    public [aClass.name/](){

    }

    [for(field : Field | aClass.attributes)]
    public [field.class.name/] get[field.name.toUpperFirst()/](){
        return this.[field.name/];
    }

    public void set[field.name.toUpperFirst()/]([field.class.name/] [field.name/]){
        this.[field.name/] = [field.name/];
    }

    [/for]
}

[/file]

[/template]
```

Figura 8.12: Definición de una plantilla para la generación de código en Acceleo.

8.6 Resumen

En este capítulo se ha presentado un enfoque dirigido por modelos para el desarrollo de Sistemas Ubicuos que trata de cubrir

el ciclo de vida del software completo, desde su especificación hasta su validación. Este proceso integra los métodos presentados en anteriores capítulos para la Ingeniería de Requisitos (REUBI) y el Diseño de Servicios (SCUBI).

Asimismo, se especifica un conjunto de transformaciones y estrategias de generación de código que permiten obtener el código de un sistema en una determinada plataforma a través de la realización de diversas transformaciones que comienzan aplicándose en el modelado de sus requisitos.

Las principales contribuciones de este enfoque son:

- Incorpora las características principales de los Sistemas Ubicuos, por lo que lo hace adecuado para su aplicación al desarrollo de este tipo de sistemas.
- Permite realizar las transformaciones de modelos desde el nivel CIM hasta la generación de código, cubriendo los 3 niveles que propone el estándar MDA.
- Integra un método de Ingeniería de Requisitos, REUBI, orientado al tratamiento sistemático de los NFR, con objeto de obtener sistemas de alta calidad.
- Integra, a su vez, un método de diseño, SCUBI, que toma los resultados de la aplicación del método de Ingeniería de Requisitos y trata de establecer cuál debería ser el diseño apropiado para cumplir las propiedades de calidad que se esperan del mismo.
- Establece las transfomaciones necesarias para convertir un modelo de requisitos en uno de diseño mediante la definición de correspondencias entre los elementos de los metamodelos origen y destino.

Parte IV

Validación de la propuesta

9 APLICACIÓN DE REUBI PARA EL ANÁLISIS DE LOS REQUISITOS DE UN SISTEMA DE POSICIONAMIENTO

Índice

9.1 Introducción

Sistema de Posicionamiento

Tras presentar el método de Ingeniería de Requisitos para Sistemas Ubicuos (REUBI), en este capítulo se procede a su aplicación a un caso de estudio concreto. El sistema que se trata de diseñar es un **Sistema de Posicionamiento** que sea capaz de dar soporte a la localización de usuarios tanto en **interiores** como en **exteriores**.

Poco tratamiento de NFR

Actualmente existen numerosos trabajos de investigación que tratan de resolver este problema. Sin embargo, dichos trabajos se centran principalmente en tres aspectos: ofrecer la funcionalidad requerida por los sistemas que hacen uso de ellos [GFDT05], mejorar la precisión del sistema de posicionamiento [PKC01] e incluir los últimos avances tecnológicos en su desarrollo [LDBL07]. De esta forma, requisitos no funcionales que son críticos para obtener un sistema de posicionamiento de calidad, tales como

la escalabilidad o la robustez, quedan relegados a un segundo plano y en ocasiones no se tienen en cuenta [LL05] [RLGBC10].

Heterogeneidad de soluciones

Adicionalmente, existe una amplia variedad de técnicas y tecnologías de posicionamiento, siendo cada una de ellas apropiada para un determinado entorno con unas condiciones específicas. Ello ha dado lugar a numerosas combinaciones posibles en sistemas de posicionamiento *ad hoc*, lo cual dificulta en gran medida la reusabilidad del sistema diseñado: dado su alto grado de acoplamiento a las condiciones del entorno, si estas cambian, es necesario reprogramar o rediseñar el sistema para adecuarlo a las nuevas características.

9.2 Sistemas de Posicionamiento

Sistemas Conscientes del Contexto

Durante los últimos años ha habido un creciente interés en los **Sistemas Conscientes del Contexto** (*Context–aware Systems*) [Fer06], sistemas capaces de adquirir y procesar el contexto del usuario, y adaptar sus contenidos, funcionalidad y forma de presentación dependiendo de dichas características.

Sistemas Basados en la Localización

Ello ha dado lugar a la aparición de los **Sistemas Basados en la Localización** (*Location-aware Systems* o *Location-based Systems*, LBS [SV04]), ya que la localización de un usuario es una característica muy importante del contexto en el que se halla inmerso. Los Sistemas basados en la Localización son servicios de información accesibles desde dispositivos móviles mediante redes de comunicación, y que son capaces de hacer uso de su ubicación para adaptarse a las necesidades del usuario.

Problemática en Sistemas de Posicionamiento

El rápido desarrollo de las tecnologías móviles en los últimos años, junto con otras tecnologías como el **Sistema de Posicionamiento Global** (*Global Positioning System*, GPS) han contribuido a la aparición de este tipo de sistemas. Sin embargo, el GPS no es capaz de proporcionar la precisión necesaria en interiores al existir gran cantidad de elementos que distorsionan la señal recibida. Más aún, para ciertos tipos de aplicaciones puede ser necesaria una precisión a nivel de habitación de la localización del usuario, algo imposible de conseguir empleando solamente el GPS. Por lo tanto, es necesario adoptar nuevas técnicas y métodos de posicionamiento para solventar este problema. Asimismo, existe una gran cantidad de tecnologías diferentes que han sido usadas exitosamente y documentadas en la bibliografía sobre sistemas de posicionamiento. Dependiendo de las necesidades concretas de la aplicación que se va a desarrollar, unas se ajustan mejor que otras.

Propiedades de Calidad

Desde el punto de vista de la Ingeniería del Requisitos, existen

varios requisitos no funcionales que se esperan en un sistema de posicionamiento. Tales requisitos, como la ya mencionada precisión del sistema, robustez, escalabilidad o interoperabilidad, son propiedades de calidad habitualmente olvidadas en el diseño de sistemas de posicionamiento, los cuales se centran principalmente en proporcionar la funcionalidad necesaria, incluir los últimos desarrollos tecnológicos y, a lo sumo, garantizar la precisión del sistema en detrimento de otros atributos de calidad. No obstante, algunos de estos requisitos son conflictivos; es decir, cuando se trata de satisfacer uno o varios de ellos, existen otros que se pueden ver perjudicados. Este hecho implica que el diseñador debe tomar decisiones de compromiso para salvaguardar las prestaciones generales del sistema. Adicionalmente, existen otras relaciones entre los requisitos (e.g. de refinamiento), y entre estos y sus operacionalizaciones, las cuales deben ser estudiadas para obtener la combinación de éstas que dé las mejores prestaciones al sistema.

Todo lo anterior motiva el caso de estudio que nos ocupa: diseñar un Sistema de Posicionamiento que sea capaz de adaptarse para cumplir las propiedades de calidad que se esperan de él, haciendo uso de las técnicas y tecnologías más apropiadas en cada momento. En las siguientes subsecciones mostraremos las principales características de un sistema de localización: arquitecturas, métodos y tecnologías. Después, aplicaremos el método REUBI para analizar de forma sistemática los requisitos de este sistema. En el capítulo siguiente, continuaremos hasta obtener su diseño e implementación aplicando SCUBI y MDUBI.

9.2.1 Arquitecturas de un Sistema de Posicionamiento

Entidades de un Sistema de Posicionamiento

Dentro de un Sistema de Posicionamiento existen dos tipos de entidades fundamentales:

- **Estaciones base:** dispositivos emisores o receptores de señal cuya posición es fija y conocida, y en base a los cuales se realiza el posicionamiento.
- **Dispositivos móviles:** dispositivos cuya posición es cambiante y debe ser determinada.

Arquitecturas de un Sistema de Posicionamiento

La taxonomía general para las diferentes Arquitecturas de un Sistema de Posicionamiento comprende tres categorías [DMS98] [Kja07]:

- **Basados en red:** del inglés *Network–based*, la posición del terminal es calculada por la propia red, para lo cual es

necesario que la red lo detecte directamente, o el terminal envíe una señal para ser detectado.

- **Basados en terminal:** del inglés *Terminal–based*, la posición del terminal es calculada por el propio terminal teniendo en cuenta las mediciones realizadas de las señales recibidas desde las estaciones base.
- **Asistidos por terminal:** del inglés *Terminal–assisted*, la posición del terminal es calculada por un servidor de la red. El terminal realiza las mediciones pertinentes de las señales recibidas desde las estaciones base, envía dicha información al servidor, que realiza los cálculos de la posición y la envía de vuelta.

9.2.2 Métodos y Técnicas de Posicionamiento

Módulos de un Sistema de Posicionamiento

Existe una gran diversidad de métodos y técnicas de posicionamiento, los cuales presentan cierta similaridad. Pahlavan [PLM02] propone un conjunto de bloques funcionales de un sistema de geolocalización habitual. Dicho sistema consta de tres partes bien diferenciadas. En primer lugar, nos encontramos con una serie de sensores que obtienen ciertas características de la señal recibida, como puede ser el tiempo desde que se transmitió la señal hasta que se recibe, la intensidad con la que llega la señal o el ángulo de llegada de la señal.

A continuación, dichos rasgos obtenidos por los sensores se envían a un algoritmo de posicionamiento, que realiza el cálculo de la localización del dispositivo móvil, y por último, se muestra al usuario de manera apropiada.

Categorías de Métodos de Posicionamiento

Podemos distinguir 4 categorías de métodos de posicionamiento: Triangulación, Métodos basados en Proximidad, Extrapolación y *Scene Analysis*. La Tabla 9.1 resume las características principales de cada categoría.

Triangulación

Métodos basados en Triangulación

Los métodos basados en **triangulación** usan las propiedades geométricas de los triángulos para determinar la posición del usuario. Existen dos variantes: *Lateration* y *Angulation*. La Tabla 9.2 resume las principales técnicas que se emplean dentro de esta categoría [ZGL02].

Lateration

Las técnicas basadas en *Lateration* realizan una estimación de la distancia desde los dispositivos móviles hasta las estaciones base. Para realizar esta tarea es necesario medir algún rasgo de la señal que varíe con la distancia, y entonces, haciendo uso de

Técnica	Basada en	Ventajas	Problemas
Triangulación	Propiedades geométricas de triángulos	Bien estudiadas y probadas (GPS)	Puede requerir hardware adicional. No útil para interiores
Proximidad	Proximidad a estaciones base	Implementación sencilla	Gran densidad de estaciones base para alcanzar buena precisión
Extrapolación	Cálculo sobre mediciones inerciales	Complemento a otras técnicas	Necesita otros métodos. Varias estimaciones sucesivas pueden llevar a resultados erróneos
Scene Analysis	Extracción de características de las señales y algoritmos de similaridad	Fácil de implementar. Variedad de algoritmos	Fase de entrenamiento muy larga. Reentrenamiento ante cambios de topología

Tabla 9.1: Resumen de métodos y técnicas de posicionamiento.

un modelo de propagación de la señal, transformar la medición a una distancia. Algunos de los rasgos que pueden medirse son el Tiempo de llegada (*Time of Arrival*, TOA), Diferencia entre tiempos de llegada (*Time Difference of Arrival*, TDOA), Tiempo de vuelta (*Roundtrip Time of Flight*, RTOF), Intensidad de Señal Recibida (*Received Signal Strength*, RSS) o Fase de la Señal (*Received Signal Phase*, RSP) [LDBL07].

Angulation

Por otra parte, las técnicas basadas en *Angulation* miden el ángulo de llegada (*Angle of Arrival*, AOA) o la dirección de llegada (*Direction of Arrival*, DOA) de la señal para determinar la posición del dispositivo móvil. Con estas mediciones, se calcula la intersección de las direcciones y se obtiene la localización, ya que la posición de las estaciones base es conocida.

Problemas de la Triangulación en interiores

Aunque estas técnicas son conocidas y han sido satisfactoriamente aplicadas en exteriores, suelen presentar problemas de degradación de la precisión en interiores, fundamentalmente debido a la ausencia de línea de vista entre el emisor y el receptor de la señal, lo cual causa reflexiones y refracciones de la misma, fenómeno que se conoce como efecto multicamino [Gei].

Técnicas basadas en Proximidad

Técnicas basadas en Proximidad

Las **Técnicas basadas en Proximidad** se basan en la cercanía del usuario a una estación base para proveer su posición [CJJA04]. Cuando se aplican estas técnicas, se cuenta con una malla muy densa de estaciones base cuya posición es conocida, y cuyo rango

Técnica	Basada en	Ventajas	Problemas
TOA	Tiempo de propagación entre el emisor y el receptor	Algorítmicamte simple	Sincronización de relojes. Marcas de tiempo
TDOA	Diferencia de tiempos entre pares de señales	Resuelve parte de los problemas de sincronización	Algorítmicamte complejo, aún algunos problemas de sincronización
RTOF	Tiempo de propagación desde que se emite la señal hasta que se recibe eco	Evita problemas de sincronización	Tiempo de procesamiento en dist. cortas degrada precisión
RSS	Intensidad de Señal Recibida	Disponible y fácil de medir	Degradación en interiores por falta de línea de vista
RSP	Fase de la Señal Recibida	Complementa para mejorar la precisión	Se degrada cuando no hay línea de vista
AOA, DOA	Ángulo o Dirección de llegada	Menos mediciones necesarias, algorítmicamente simple	Hardware complejo. Degradación al alejarse de los emisores de señal

Tabla 9.2: Resumen de técnicas basadas en Triangulación.

de alcance es relativamente limitado (del orden de unos pocos metros).

Cuando se detecta el dispositivo desde una única estación base, se supone la localización de dicha estación. Cuando se detecta el dispositivo desde más de una estación, se toma la posición de la que reciba mayor intensidad.

Estas técnicas presentan la ventaja de ser sumamente fáciles de implementar. Las tecnologías usadas habitualmente en este tipo de técnicas son los infrarrojos y RFID.

La técnica de *Cell–Identification*, *Cell–ID* o *Cell of Origin* es otra técnica que se engloba dentro de esta categoría. La posición del usuario es determinada por el punto de acceso al que está accediendo el terminal para realizar sus comunicaciones. Es una técnica que se encuentra ya en uso y es soportada por la mayoría de los terminales móviles.

Extrapolación

Dead-Reckoning

La técnica de ***Dead Reckoning*** (abreviación de *Deduced Reckoning*, que podríamos traducir como extrapolación), obtiene la posición del usuario basándose en posiciones anteriores y haciendo uso de la dirección de movimiento del dispositivo, la

velocidad y la aceleración [RDM03]. Dicha información puede obtenerse de diversas maneras: deduciéndolas a partir de puntos anteriores, o haciendo uso de otros sensores, tales como acelerómetros, cuentarrevoluciones y giroscópios. Como desventaja, hay que apuntar que si el tiempo de muestreo es demasiado amplio, el cálculo efectuado puede ser bastante erróneo.

Para poder usar esta técnica, es necesario conocer la posición original del dispositivo, la cual puede ser calculada mediante otro método. Esto implica que la técnica de Dead Reckoning no está pensada para ser empleada aisladamente, sino como complemento de ayuda a la mejora de la precisión de otras técnicas.

Scene Analysis

Scene Analysis

Los métodos basados en **Scene Analysis** se basan en la extracción de rasgos de la señal recibida en diferentes puntos de referencia, para obtener lo que se conoce como *fingerprint*, que son etiquetados y caracterizan una localización. Una vez recopilada una colección de *fingerprints*, el posicionamiento se realiza tomando una medición de los mismos rasgos que se tomaron durante la recopilación, y se compara con la colección empleando un algortimo determinado. Dicho algoritmo retorna la localización cuyo *fingerprint* se asemeje más a la medición realizada.

Fases de Scene Analysis

Vemos, pues, que esta técnica consta de 2 fases: una fase *offline* en la que se realiza el entrenamiento (toma de muestras y etiquetado), y otra fase *online* en la que el sistema se encuentra en funcionamiento y se realiza el posicionamiento. La Tabla 9.3 resume las principales técnicas que se emplean en esta categoría.

Técnica	**Basada en**	**Ventajas**	**Problemas**
Vecinos más cercanos	Calcular métrica de distancia y elegir los k más similares	Simple y fácil de implementar	No emplea mucha información de la fase de entrenamiento
Redes Neuronales	Perceptrón multicapa para clasificar patrones	Buena generalización de información	Difícil de entrenar. Problemas de escalabilidad
Métodos Probabilísticos	Calcular probabilidades dadas las mediciones observadas	Mejor precisión, usa más datos del entrenamiento	Requiere mayor computación

Tabla 9.3: Resumen de técnicas basadas en *Scene Analysis*.

9.2.3 Tecnologías de Posicionamiento

Elección de Infraestructura de Posicionamiento

A la hora de realizar la selección de tecnologías que se van a usar en el posicionamiento, y por lo tanto, del hardware necesario, se pueden seguir dos enfoques. Por una parte, podemos diseñar una infraestructura dedicada únicamente al posicionamiento. Dicho enfoque proporcionará mejor cobertura y mejores resultados, a costa de un incremento en el coste del sistema. Por otro lado, puede que existan restricciones de coste en el desarrollo del sistema, por lo que el otro enfoque consiste en reutilizar la infraestructura existente para otro propósito (e.g. infraestructura de comunicaciones). El coste de este enfoque es menor, pero requiere mejores algoritmos de posicionamiento para compensar las posibles imprecisiones. Nótese también que la colocación de las estaciones base debe realizarse de manera que garantice la comunicación y el posicionamiento. En [PKC01] se discute este hecho, ya que cuando la señal atraviesa muros, se incrementa el error, y se recomienda colocar las estaciones base de manera que la señal no atraviese más de 2 ó 3 muros para cada localización.

Categorías de Tecnologías de Posicionamiento

De acuerdo con Kaemarungsi [Kae], podemos clasificar las posibles tecnologías de sensores en tres categorías:

- **Infrarrojos:** la tecnología de infrarrojos se comporta de manera similar a la luz. No puede atravesar las paredes, por lo que su rango de aplicación es limitado. Su alta velocidad, $3 \cdot 10^8$m/s, requiere de circuitería sofisticada para realizar las mediciones. La iluminación del edificio interfiere, causando problemas en las mediciones.

- **Radio frecuencia:** las señales de radio pueden penetrar casi cualquier material existente en un edificio, lo que las hace especialmente apropiadas para el posicionamiento en interiores. Posee el rango más amplio de alcance, comparada con las otras dos categorías.

- **Ultrasonidos:** los ultrasonidos son ondas a frecuencias muy bajas (40 kHz). Los dispositivos son simples y baratos, y poseen buena precisión para realizar las mediciones. No penetra a través de las paredes, pero sí refleja las obstrucciones encontradas. Su rango de alcance es muy corto, pero posee gran resolución (hasta 1 cm). Presenta el problema de que la temperatura puede afectar a su rendimiento.

Adicionalmente, proponemos una categoría de dispositivos más, que permite realizar el posicionamiento con otras técnicas:

- **Inerciales:** los dispositivos inerciales miden características del movimiento de un dispositivo móvil, como puede ser la velocidad o la aceleración. La posición del usuario puede obtenerse mediante la integración de esas medidas.

Dado que las tecnologías de radio frecuencia son las más extendidas, fácilmente disponibles y han sido ampliamente estudiadas en la bibliografía, analizaremos algunas de tecnologías concretas dentro de esta categoría como son GPS, WiFi, Bluetooth, ZigBee y RFID.

GPS

El Sistema de Posicionamiento Global (GPS) es un sistema de navegación global basado en satélites, probablemente el más extendido y popular. Se basa en un conjunto de satélites alrededor de la Tierra que están transmitiendo señales. De cada señal se mide su TOA y se aplica una técnica basada en triangulación para obtener la posición del dispositivo en exteriores [AAH+97] [Mar99]. Sin embargo, no puede dar la precisión necesaria en entornos interiores. Por tanto, otras tecnologías deben ser consideradas para este propósito.

WiFi

IEEE 802.11, también conocido como WiFi o WLAN es un estándar para comunicaciones de red de área local de manera inalámbrica. Tiene un rango de aplicación medio y habitualmente está disponible en muchos lugares, lo que lo hace especialmente apropiado para posicionamiento en interiores [PKC01] [APT+09]. Además, algunos trabajos emplean esta tecnología para obtener posicionamiento en exteriores [Kaw09].

Bluetooth y ZigBee

IEEE 802.15 es un conjunto de estándares que tratan sobre comunicaciones en redes de área personal. Los protocolos más populares dentro de esta categoría son 802.15.1, Bluetooth, que fue diseñado para el intercambio de datos y la sincronización de dispositivos móviles, y 802.15.4, ZigBee, que se aplica en redes de sensores y actuadores en ambientes inteligentes. Ambos han sido empleados para sistemas de posicionamiento con diferentes propósitos; por ejemplo, un sistema de publicidad basado en la localicación [AGKO04] [APT+09], via Bluetooth, o un entorno de *Ambient Assisted Living* con ZigBee [BMC+09].

RFID

Por último, la tecnología de Identificación por Radio Frecuencia (RFID) es una forma de almacenamiento y recuperación de información mediante una transmisión electromagnética a un circuito compatible. Su rango de alcance, muy corto, la hace muy apropiada para aplicarse junto con técnicas basadas en la proximidad [CJJA04]. Sin embargo, si la RSS está disponible, o cualquier otra característica, también se puede usar con otras técnicas.

9.3 Aplicación del Método REUBI

Aplicación del Método

En esta sección se aplica el método REUBI al análisis de un Sistema de Posicionamiento, basándose en la breve revisión del estado de la técnica en sistemas de posicionamiento expuesta en la sección anterior. Cada una de las siguientes subsecciones corresponde a una fase del método, en las que se presentan los diagramas obtenidos junto con una breve explicación de los mismos. La construcción del grafo de interdependencia se irá mostrando parcial e incrementalmente en las siguientes figuras. Para facilitar la legibilidad de los diagramas, se omite la argumentación se omite en los diagramas y se justifican las decisiones en el texto.

9.3.1 Modelo de Valores

Actores y Valores

De acuerdo con lo explicado en el capítulo 5, en primer lugar debe construirse el Modelo de Valores. El principal actor identificad es un actor que desempeña el rol de un **Sistema basado en la Localización** (LBS) [SV04]. El sistema de posicionamiento proporciona un valor que es la **localización** del LBS. Dicho valor es un servicio, el cual es demandado por el LBS.

Potenciadores

Adicionalmente, existen varias características que potencian la adquisición del valor, tal y como puede verse en la Figura 9.1. Los potenciadores son:

- Obtener localizaciones con una alta precisión.
- Obtener localizaciones de manera rápida.
- Minimizar el coste de obtener posicionamiento.
- Preservar la privacidad del usuario.

9.3.2 Definición de *Goals* y *Softgoals*

Definición de Goals y Softgoals

A partir del Modelo de Valores descrito en la subsección anterior, podemos derivar los *goals* y *softgoals* principales del sistema. De esta manera, el *goal* **Positioning** se deriva del valor intercambiado por los actores, tal y como puede verse en la Figura 9.2.

Refinamiento de objetivos

Este objetivo es aún demasiado genérico, por lo que debe ser refinado (Figura 9.2). Puede ser descompuesto de manera inclusiva en dos *subgoals*: **Posicionamiento en Interiores** y **Posicionamiento en Exteriores**. Para la consecución del objetivo de alto nivel

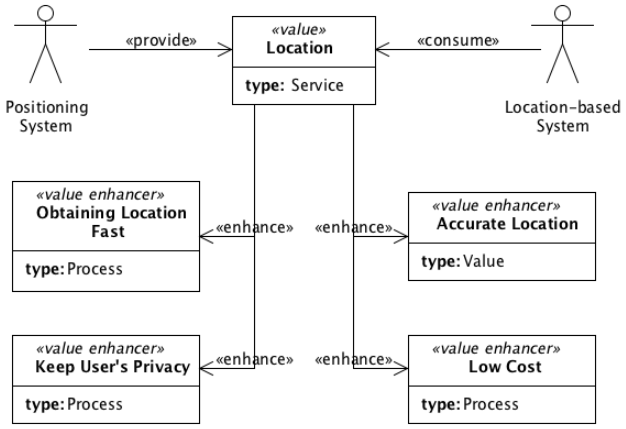

Figura 9.1: Modelo de Valores para un Sistema de Posicionamiento.

es necesaria la realización de ambos subobjetivos. Asimismo, el objetivo **Posicionamiento en Interiores** puede descomponerse en otros dos subobjetivos: **Realizar una medición de una señal** y **Realizar una estimación de la posición**. Al igual que en el caso anterior, ambos deben realizarse para satisfacer el objetivo de más alto nivel.

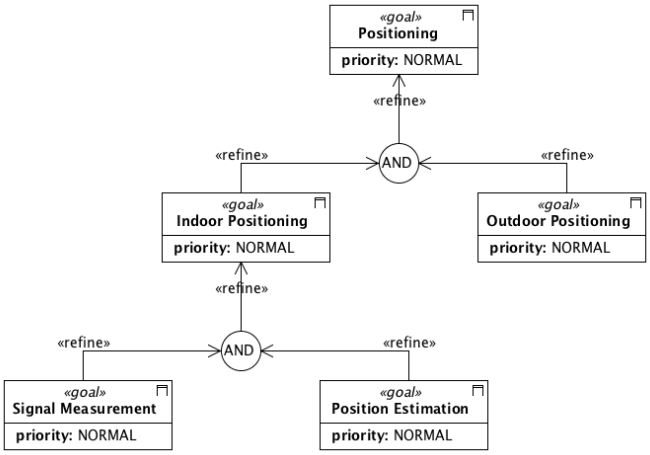

Figura 9.2: Descomposición de *Goals*.

Derivación de Softgoals

Considerando los potenciadores descritos en el Modelo de Valores, podemos derivar un conjunto de *softgoals*, tal y como puede verse en las Figuras 9.3, 9.4, 9.5 y 9.6.

El *softgoal* ***Performance*** (Figura 9.3), derivado a partir del potenciador *Obtener localizaciones de manera rápida*, se descompone en dos *softgoals* de más bajo nivel: ***Responsiveness*** (garantizar

un bajo tiempo de respuesta) y **Escalabilidad** (garantizar que el sistema sea escalable). Este último se descompone en dos *softgoals* de más bajo nivel: **Escalabilidad Geográfica** (garantizar que el sistema puede ampliar su área de cobertura) y **Escalabilidad en Densidad** (garantizar que el sistema puede aumentar el número de unidades posicionadas por unidad de espacio y tiempo) [LDBL07].

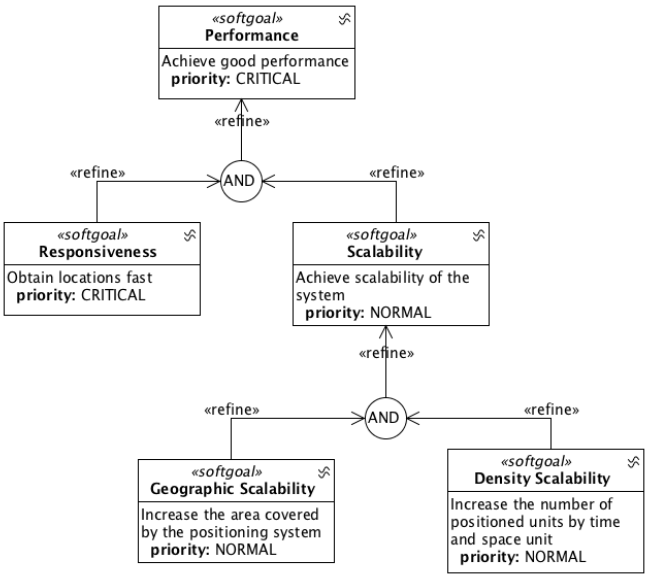

Figura 9.3: Descomposición de *Softgoals* I.

El *softgoal* **Bajo coste** (Figura 9.4), derivado a partir del potenciador *Minimizar el coste de obtener posicionamiento*, se descompone en dos *softgoals* de más bajo nivel: **Minimizar el coste del hardware**, **Minimizar el consumo de energía** y **Minimizar el tiempo de mantenimiento del sistema**. Este último se descompone a su vez en **Minimizar el tiempo de instalación del sistema** (despliegue de dispositivos, entrenamiento del sistema...) y **Minimizar el tiempo de operación del sistema** (tareas de recalibración, mantenimiento...).

El *softgoal* **Precisión** (Figura 9.5), derivado a partir del potenciador *Obtener localizaciones con una alta precisión*, se descompone en dos subobjetivos: **Precisión en interiores** (garantizar una alta precisión en interiores) y **Precisión en exteriores** (garantizar una alta precisión en exteriores).

El *softgoal* **Robustez** (Figura 9.5) aparece como consecuencia del análisis de obstáculos (véase Sección 9.3.3). Puede descomponerse en dos *softgoals*: **Robustez frente a fallos en la transmisión de señales** (el sistema debe garantizar el correcto funcionamiento

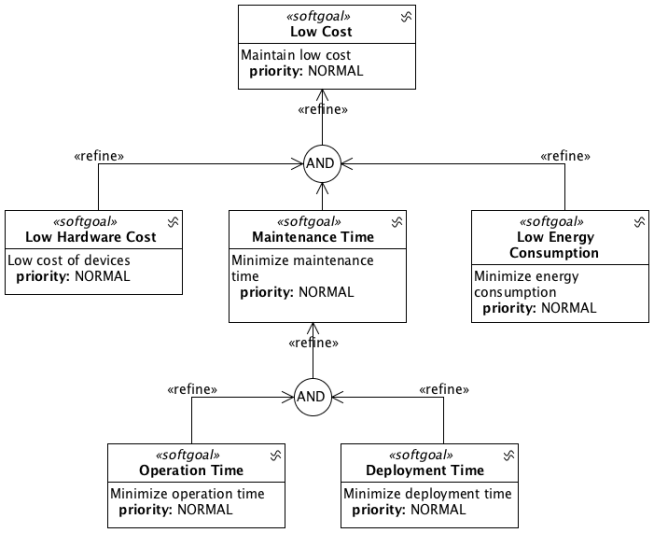

Figura 9.4: Descomposición de *Softgoals* II.

del sistema ante fallos en la transmisión de las señales, causados por caídas de los transmisores, fallos de corriente...) y **Robustez frente a la recepción de señales desconocidas** (el sistema debe garantizar el correcto funcionamiento ante la recepción de una señal cuyo identificador nunca haya sido visto previamente) [LL05].

Por último, el *softgoal* **Privacidad** (Figura 9.6), derivado del potenciador *Preservar la privacidad del usuario* [Bec03], se descompone en otros dos *softgoals* de más bajo nivel: **Mantener el anonimato del usuario** (la identidad del usuario sólo debe ser revelada a otros si éste lo desea) y **Mantener la intimidad del usuario** (no debe registrarse la ubicación del usuario) [Lan02].

9.3.3 Análisis de Obstáculos

Análisis de Obstáculos

Posteriormente, se procede a la identificación de obstáculos y su resolución (Figura 9.7). El principal obstáculo encontrado es que ocurra un fallo en la toma de mediciones de una señal (caída del transmisor de señal, fallo en el suministro de energía...) [SJ99]. Para paliar este efecto, se introduce el *softgoal* **Robustez**, implicando que el sistema debe ser robusto ante fallos en la transmisión de la señal, tal y como se apuntó en la subsección anterior.

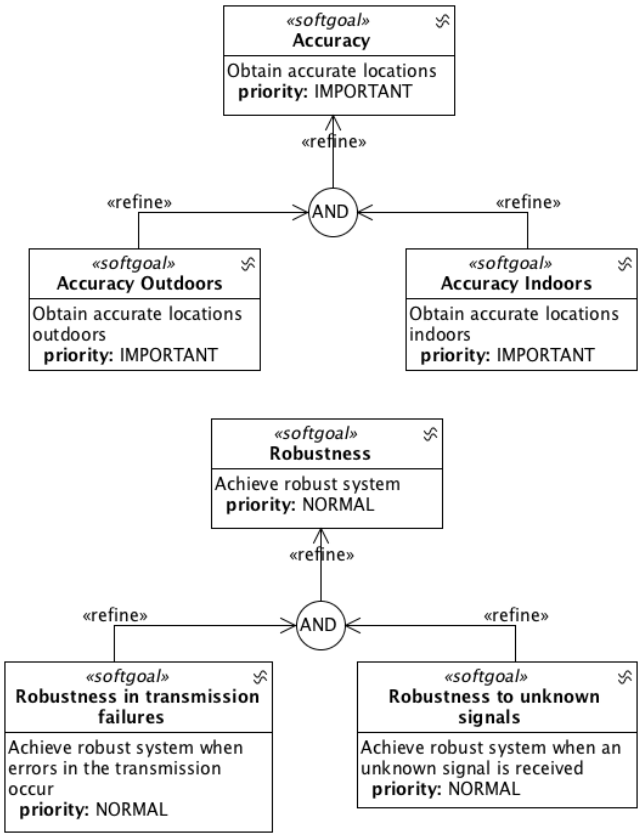

Figura 9.5: Descomposición de *Softgoals* III.

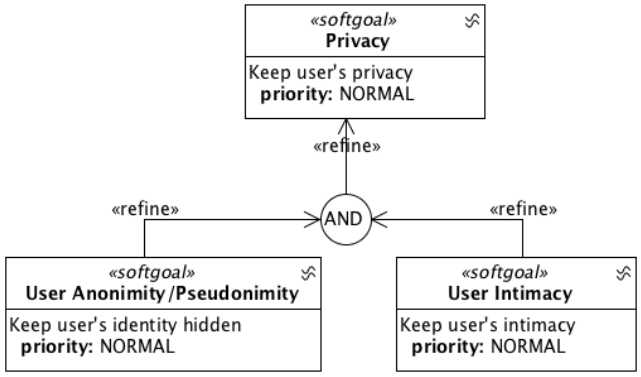

Figura 9.6: Descomposición de *Softgoals* IV.

9.3.4 Intercambio de Recursos

Intercambio de Recursos

A continuación debe procederse a analizar los recursos que son necesarios para la consecución de los objetivos, y los que se

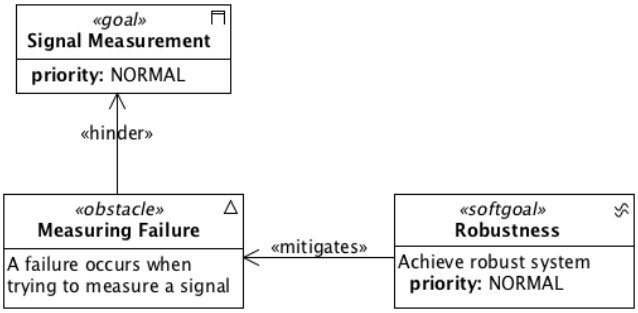

Figura 9.7: Obstáculos que afectan al sistema de posicionamiento.

generan como resultado de la consecución de un objetivo. En el caso que nos ocupa, el *goal* **Realizar una medición de una señal** produce como resultado un conjunto de mediciones sobre las señales recibidas, mientras que el *goal* **Realizar una estimación de la posición** necesita dichas mediciones para realizar el cálculo de la posición. Es decir, temporalmente, el primero de los objetivos debe realizarse antes que el segundo de ellos para satisfacer esta dependencia, tal y como puede observarse en la Figura 9.8.

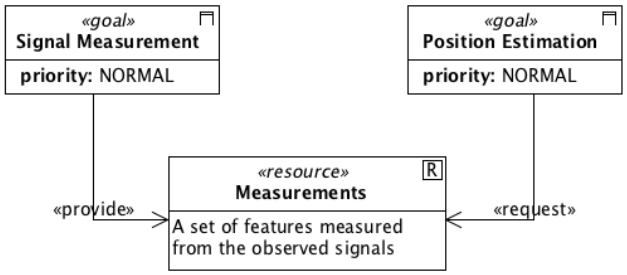

Figura 9.8: Intercambio de recursos.

9.3.5 Operacionalización

En esta etapa del método deben estudiarse las posibles decisiones arquitectónicas y de diseño que pueden tomarse, junto con las contribuciones, tanto positivas como negativas, que dichas decisiones hacen a la satisfacción de los objetivos.

Operacionalización de Goals

En primer lugar, se consideran los *goals* de más bajo nivel (aquéllos que no son refinados en sub-objetivos). Así, el *goal* **Tomar una medición de una señal** puede realizarse mediante cuatro decisiones correspondientes al uso de las tecnologías de infrarrojos, ultrasonidos, radio frecuencia o inerciales (Figura

9.9). Algunos ejemplos concretos de estas dos últimas son WiFi, Bluetooth, ZigBee o RFID (para radio frecuencia), y acelerómetros, giroscopios o brújulas (para inerciales).

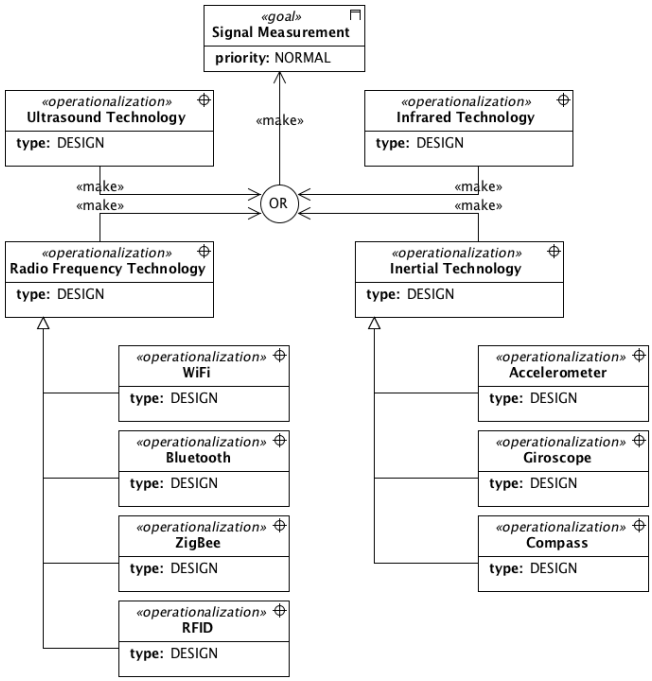

Figura 9.9: Operacionalización I.

El *goal* **Realizar una estimación de la posición** puede operacionalizarse tomando otras cuatro decisiones de manera alternativa, que implican el uso de técnicas basadas en proximidad, *dead-reckoning*, triangulación o *scene analysis* (Figura 9.10). Si se opta por el empleo de la triangulación, pueden optarse por usar técnicas basadas en tiempo de llegada, diferencia entre tiempos de llegada, tiempo de ida y vuelta de la señal, intensidad de señal recibida o fase de la señal recibida. En el caso de *scene analysis*, algunos de los algoritmos que pueden emplearse son *k*-Vecinos más cercanos, redes neuronales, métodos probabilísticos, *support vector machines* o *smallest m-vertex polygon*.

Para el último *goal*, **Obtener posicionamiento en exteriores**, la única operacionalización posible es el uso del Sistema de Posicionamiento Global (GPS) (Figura 9.11).

Operacionalización de softgoals

A continuación, deben operacionalizarse los *softgoals* que no pueden ser descompuestos en otros de menor granularidad. Para el caso del *softgoal* ***Responsiveness***, contamos con las siguientes contribuciones [LDBL07] (Figura 9.12):

Justificación de las decisiones tomadas para operacionalizar el tiempo de respuesta

- El uso de técnicas basadas en proximidad satisface el *soft-*

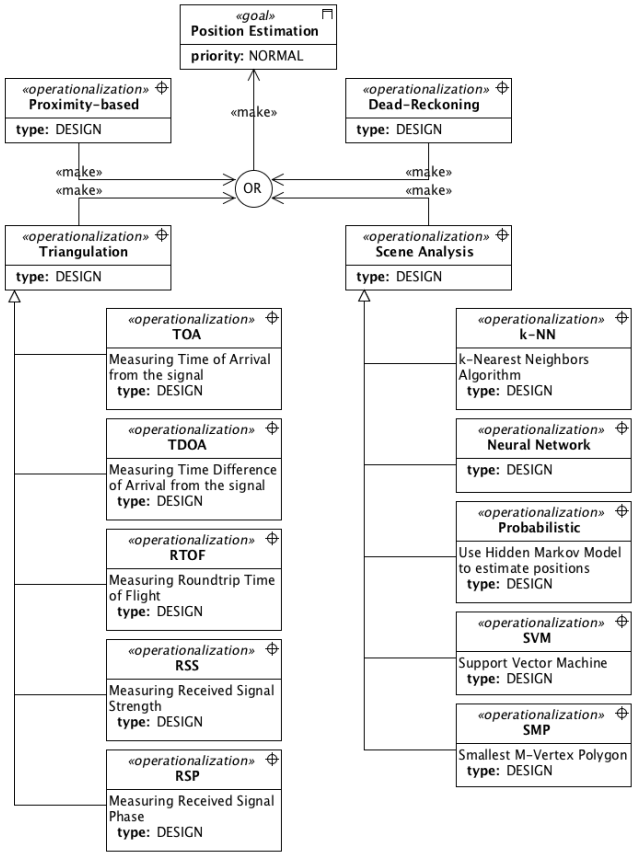

Figura 9.10: Operacionalización II.

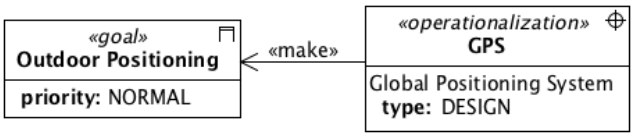

Figura 9.11: Operacionalización III.

goal, ya que estas técnicas apenas deben realizar cómputos.

- El uso de triangulación o *dead-reckoning* contribuye positivamente a la satisfacción del *softgoal*, ya que deben realizar algunos cálculos más que la anterior.

- *k*-NN contribuye positivamente a la satisfacción, pero en menor medida que las dos anteriores, puesto que debe realizar un mayor número de cálculos.

- Los métodos probabilísticos tienen un impacto negativo en la satisfacción del *softgoal*, ya que es la técnica que mayor número de cálculos involucra.

- Se desconoce el impacto que tienen las redes neuronales, las *support vector machines* y la técnica de *smallest m-vertex polygon* ya que no han sido documentadas en la bibliografía.

- La agrupación y combinación de varias de estas técnicas presenta un impacto negativo en la satisfacción del *softgoal*, ya que se requiere un tiempo mayor para realizar el cómputo de la localización varias veces.

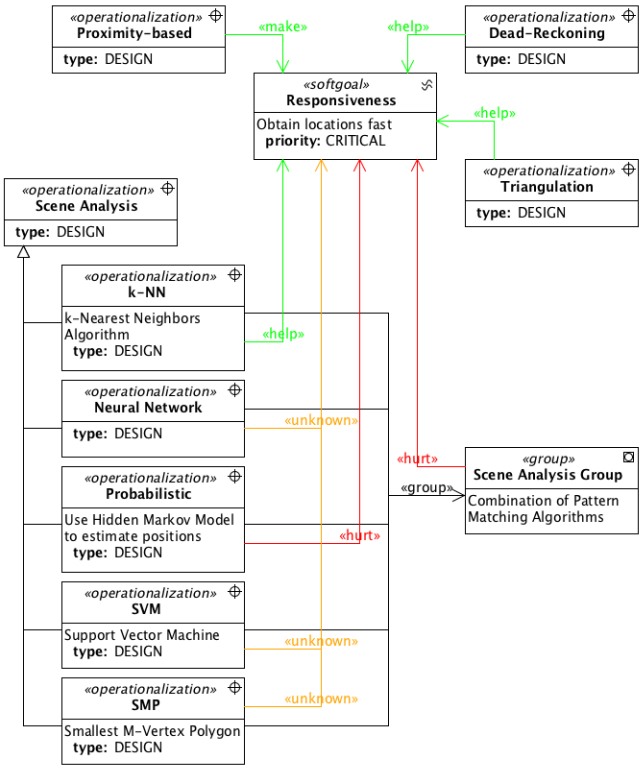

Figura 9.12: Operacionalización IV.

Para satisfacer los *softgoals* de **Escalabilidad Geográfica** y **Escalabilidad en Densidad**, pueden tomarse las siguientes decisiones arquitectónicas:

Operacionalizaciones para la escalabilidad

- En cuanto a la arquitectura de posicionamiento, podemos optar por Asistida por Terminal, Basada en Terminal y

Basada en Red. La contribución de esta última a la escalabilidad geográfica es desconocida, mientras que las otras dos contribuyen de manera positiva.

- Un balanceador de carga contribuye positivamente tanto a la escalabilidad geográfica (redirige el tráfico del sistema por zonas) como a la escalabilidad en densidad (redirige el tráfico a aquellos servidores que estén menos congestionados).

- Un servicio replicado, igualmente, contribuye tanto a la escalabilidad geográfica (replicación por zonas) como a la escalabilidad en densidad (cada uno se hace cargo de un subconjunto de los usuarios del sistema).

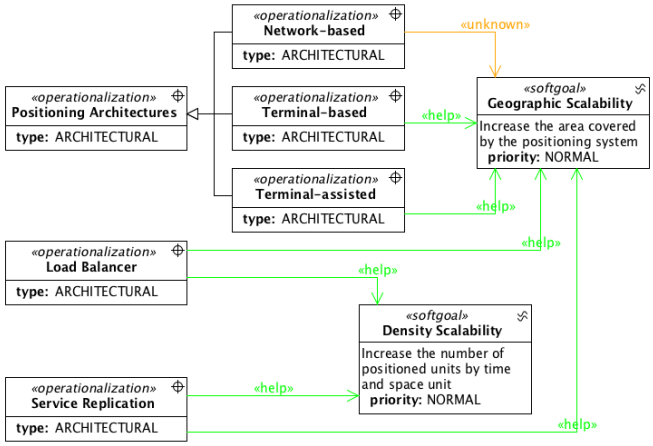

Figura 9.13: Operacionalización V.

Operacionalizaciones para el coste

Respecto a los *softgoals* relativos a la minimización del tiempo de despliegue y de operación, las técnicas de triangulación y *scene analysis* requieren un tiempo de entrenamiento prolongado, especialmente en el caso de estas últimas, por lo que contribuyen de manera negativa a la minimización del tiempo de despliegue. El resto de técnicas presentan contribuciones positivas a la satisfacción de ambos *softgoals* (Figura 9.14).

En cuanto al *softgoal* **Minimizar el coste del hardware**, las tecnologías de infrarrojos y ultrasonidos presentan habitualmente un coste más elevado, por lo que su contribución es negativa. Por otra parte, se desconoce el coste de los dispositivos inerciales, mientras que las tecnologías de radio frecuencia están más extendidas y pueden ser reutilizadas para el posicionamiento,

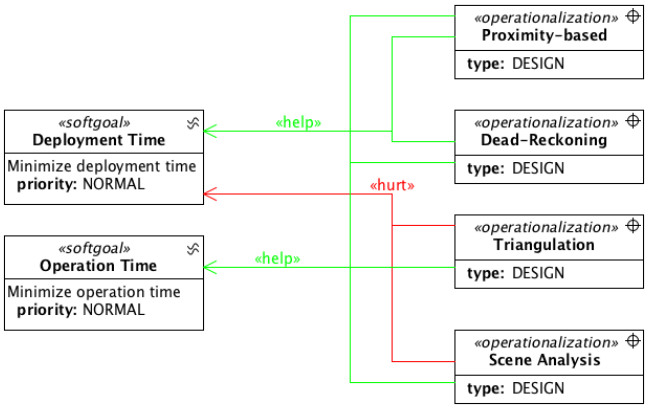

Figura 9.14: Operacionalización VI.

contribuyendo así de manera positiva a la consecución del *softgoal*. Para **Minimizar el consumo de energía**, los únicos datos existentes en la bibliografía apuntan que el estándar ZigBee promueve un bajo consumo en sus dispositivos, tal y como se refleja en la Figura 9.15. Adicionalmente, algunos autores proponen la combinación de las tecnologías inerciales junto con otro tipo de tecnologías para reducir la transmisión de señales y, por tanto, el consumo de energía.

Operacionalizaciones para la precisión

La precisión en exteriores viene dada por el empleo del Sistema de Posicionamiento Global. Sin embargo, para la obtención de una buena precisión en interiores, existe una amplia variedad de opciones con diferentes contribuciones [LDBL07] (Figuras 9.16 y 9.17):

- Las tecnologías de infrarrojos y ultrasonidos presentan una gran resolución (hasta 1 cm), por lo que su uso satisface el objetivo.
- Las tecnologías inerciales presentan el problema de que, si el intervalo de muestreo no es suficientemente pequeño, su precisión se degrada rápidamente. Además, deben complementarse con otras técnicas, ya que el error cometido después de varias estimaciones puede crecer considerablemente. Por este motivo, su contribución a la satisfacción es negativa.
- Dentro de las tecnologías de radio frecuencia, existen diferentes contribuciones. WiFi, Bluetooth y ZigBee han sido documentadas como opciones que proporcionan una buena precisión en interiores (hasta 1.5 m). RFID proporciona una

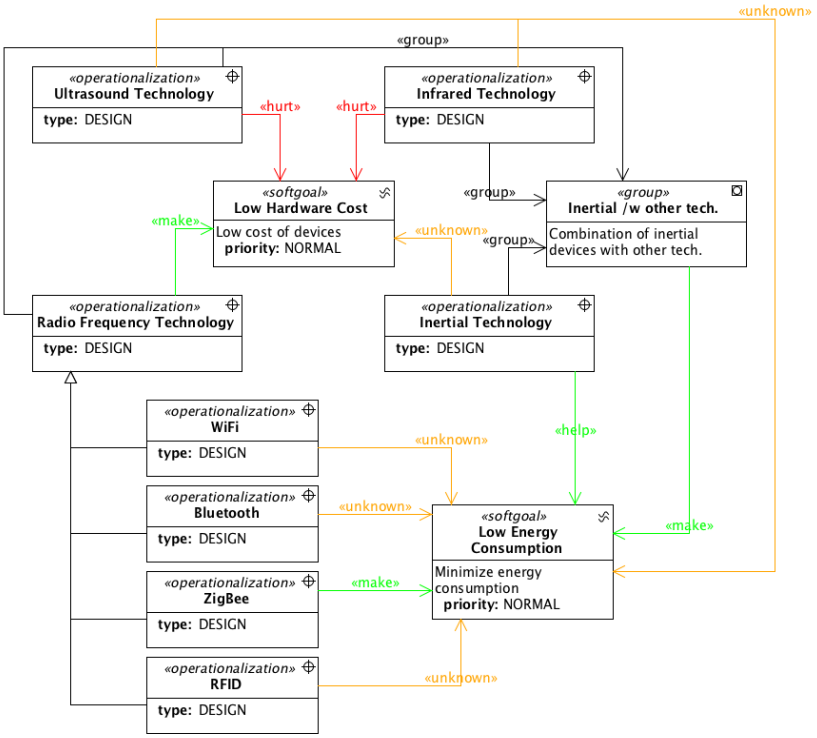

Figura 9.15: Operacionalización VII.

mayor precisión gracias a su reducido alcance, por lo que su contribución a la satisfacción del objetivo es mayor que en las anteriores.

- La combinación de diferentes tecnologías de posicionamiento, haciendo uso de las bondades de cada una de ellas, satisface el objetivo.

- Las técnicas basadas en *dead-reckoning* implican el uso de tecnologías inerciales, y tal como se ha comentado anteriormente, presentan problemas de precisión si se utilizan de manera aislada.

- Las técnicas basadas en proximidad sólo estiman la posición en base a la estación transmisora más cercana, por lo que pueden cometer grandes errores, de ahí la relación negativa a la satisfacción del objetivo.

- Las técnicas de triangulación presentan problemas de precisión cuando se aplican en interiores, ya que la señal recibida puede resultar distorsionada por los elementos presentes

en el ambiente, dificultando la estimación de distancias y, por tanto, el cálculo de posiciones.

- Dentro de las técnicas basadas en *scene analysis*, no se han encontrado datos sobre la precisión obtenida empleando redes neuronales y *support vector machines*, por lo que su contribución es desconocida. La técnica de *smallest m-vertex polygon* no presenta buenos resultados por ser demasiado simple. *k*-NN presenta resultados aceptables, pero la mejor precisión viene dada por el empleo de técnicas probabilísticas, las cuales hacen uso de conocimiento sobre el entorno de aplicación del sistema para propocionar la localización.
- La combinación de diferentes métodos y técnicas de posicionamiento, aprovechando los aspectos más positivos de cada uno de ellos, también satisface el objetivo.

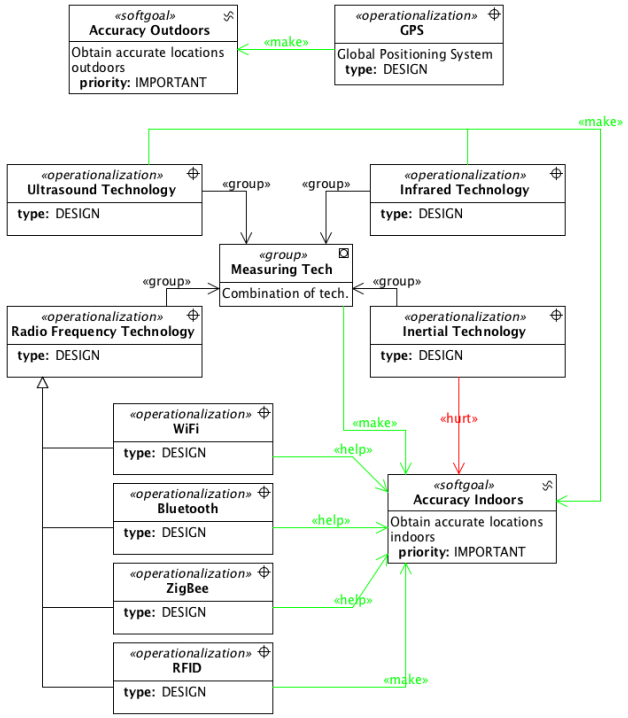

Figura 9.16: Operacionalización VIII.

Operacionalizaciones para la robustez

El *softgoal* **Mantener la robustez frente a fallos en la transmisión** recibe una contribución positiva de la combinación de varias tecnologías (Figura 9.18), ya que el sistema no depende de una única tecnología, y si alguna de ellas falla, el sistema puede continuar funcionando haciendo uso de otras [RLGBC10]. Adicionalmente, recibe otra contribución positiva de la operacionalización

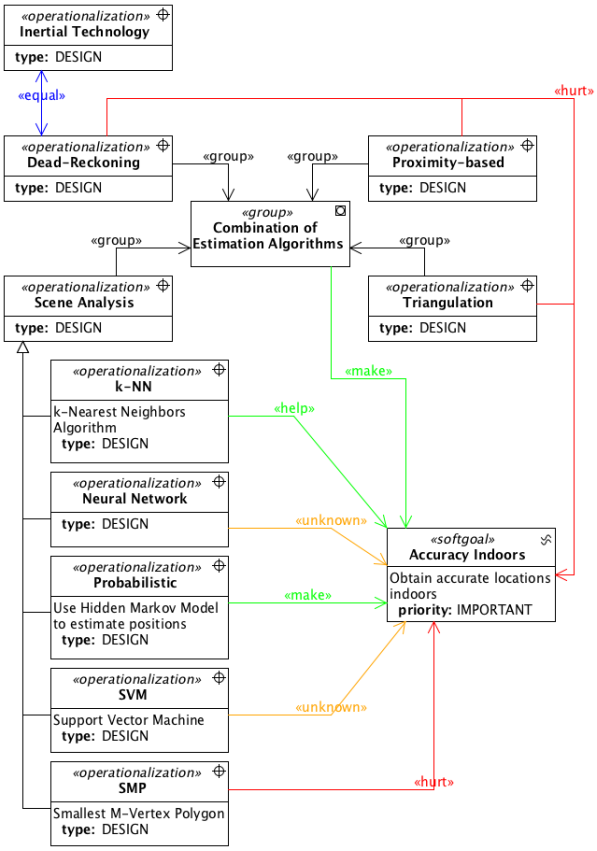

Figura 9.17: Operacionalización IX.

de replicación del servicio, ya que si una de las instancias del servicio no es capaz de dar posicionamiento, puede emplearse otra diferente. Por otra parte, para cumplir el *softgoal* **Mantener la robustez frente a la recepción de señales desconocidas**, la decisión tomada es ignorar dichas señales, ya que no provienen del sistema de posicionamiento y no proporcionan información útil.

Operacionalizaciones para la privacidad

Por último, para los *softgoals* relativos a mantener el anonimato y la intimidad del usuario[Bec03], la arquitectura más apropiada es la basada en terminal, mientras que la basada en red daña la intimidad del usuario (éste puede no ser consciente de que está siendo posicionado). En cuanto a la asistida por terminal, se desconoce el efecto que tiene en ambos requisitos. Adicionalmente, se toman otras decisiones:

- Proporcionar *feedback* y control para que el usuario conozca qué información sobre su identidad se está dando a conocer, y controlarla[BS93].

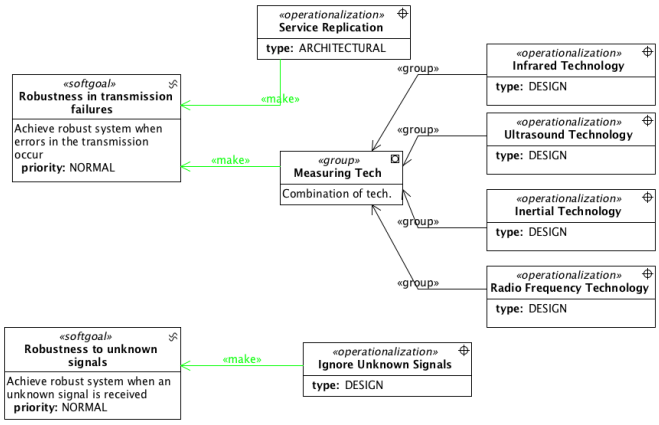

Figura 9.18: Operacionalización X.

- Organizar la identidad del usuario de manera jerárquica (por ejemplo, Usuario anónimo → Pseudónimo → Nombre real), de manera que, dependiendo de las preferencias del usuario se toma un nivel de la jerarquía para mostrar su identidad [JL02].
- Mantener los datos de manera local, sin que se tenga acceso externo a la posición de un usuario [XZZG08].

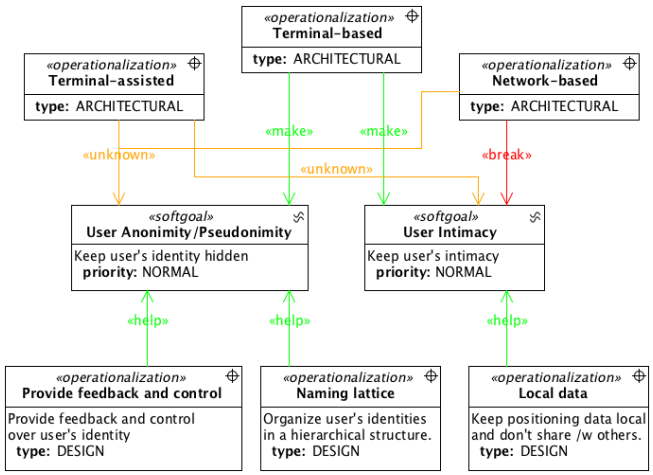

Figura 9.19: Operacionalización XI.

9.3.6 Modelado de Contexto

Modelado de Contexto

Existen numerosas situaciones en las que el contexto que rodea

al sistema de posicionamiento tiene una influencia sobre éste y, más concretamente, sobre la priorización de los objetivos. Puesto que no es factible analizar todas y cada una de las situaciones que pueden darse dada la cantidad y variedad de éstas, en esta sección se analizan, a modo de ejemplo, cinco situaciones que pueden suceder con una mayor probabilidad.

Situaciones de Contexto

La Figura 9.20 muestra tres posibles situaciones de contexto. La primera de ellas corresponde a que un usuario se encuentre en casa. Dicha situación viene caracterizada por dos atributos: recibir una señal de un dispositivo que se encuentre en casa (identificado por su dirección MAC) y con una intensidad superior a media-alta. La segunda situación corresponde a que el usuario se encuentre fuera de casa, pero cerca de ella. Los atributos que caracterizan esta situación son idénticos a la anterior, salvo que la intensidad con la que se recibe la señal es inferior a media-alta. Se determina este nivel ya que la señal puede recibirse desde fuera de casa con una intensidad inferior. Por último, la tercera situación corresponde a que el usuario se encuentra fuera de la casa, donde no se recibe la señal procedente del dispositivo situado en su interior.

La Figura 9.21 muestra otras dos posibles situaciones de contexto. La primera de ellas representa un estado del dispositivo móvil en el que sus recursos se están acabando. Viene caracterizada por un nivel de batería inferior al nivel bajo. Por otra parte, existe otra situación de interés correspondiente a la ocurrencia de un accidente en el lugar donde se encuentra el sistema. Dicho evento está caracterizado por la activación de una alarma.

9.3.7 Priorización de Objetivos

Priorización de Objetivos

La priorización de objetivos se ha realizado paralelamente a su modelado en las secciones anteriores, como puede observarse en los diagramas asociados. En esta sección se estudia cómo las situaciones de contexto modeladas en la Sección 9.3.6 afectan a la priorización de objetivos.

Impacto del Contexto en Goals

La Figura 9.22 muestra el impacto de las situaciones *Estar en casa*, *Estar fuera de casa* y *Estar cerca de casa, en el exterior*. En la primera de ellas, tan sólo el objetivo relativo a *Posicionamiento en Interiores* debe satisfacerse. En la segunda, de igual manera, sólo el objetivo *Posicionamiento en Exteriores* debe satisfacerse. Por último, en la situación *Estar cerca de casa*, ambos deben satisfacerse, ya que el usuario puede encontrarse, por ejemplo, en el jardín colindante.

Impacto del Contexto en la Priorización

La Figura 9.23 muestra el cambio de prioridades en una situa-

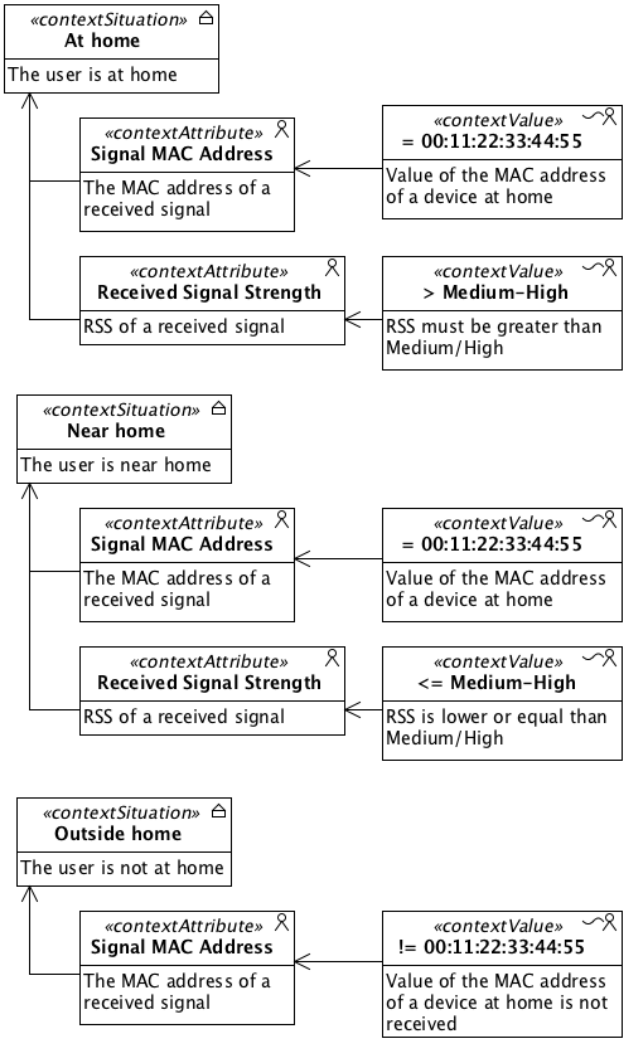

Figura 9.20: Modelado de Contexto I

ción de accidente. En el caso de operación normal del sistema, la precisión tiene una prioridad *importante* y el tiempo de respuesta es *crítico*. Cuando un accidente ocurre, la precisión del sistema obtiene una prioridad *crítica*, mientras que el tiempo de respuesta recibe una prioridad *importante*, ya que estamos interesados en conocer con el mayor grado de precisión posible dónde puede haber posibles víctimas del accidente, aunque ello requiera un mayor tiempo de cómputo de la localización.

Por último, la Figura 9.24 muestra el cambio de prioridad en una situación en la que el dispositivo se está quedando sin recursos. El objetivo *Minimizar el consumo de energía* tiene una prioridad normal en el funcionamiento habital del sistema. Cuan-

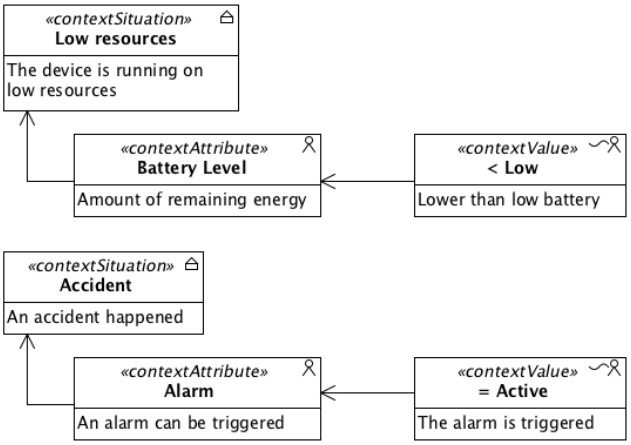

Figura 9.21: Modelado de Contexto II

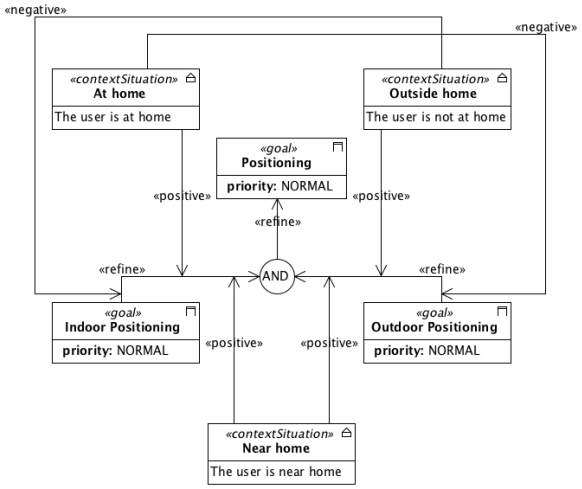

Figura 9.22: Priorización de objetivos I

do esta situación se da, la prioridad de este objetivo es *crítica*, ya que si el dispositivo se queda sin batería, el posicionamiento no puede realizarse.

9.3.8 Evaluación

Una vez construido el Grafo de Interdependencia, se debe aplicar el algoritmo de evaluación presentado en la Sección 5.10 junto con las heurísticas descritas. En el caso de estudio que nos ocupa, tomaremos como entrada:

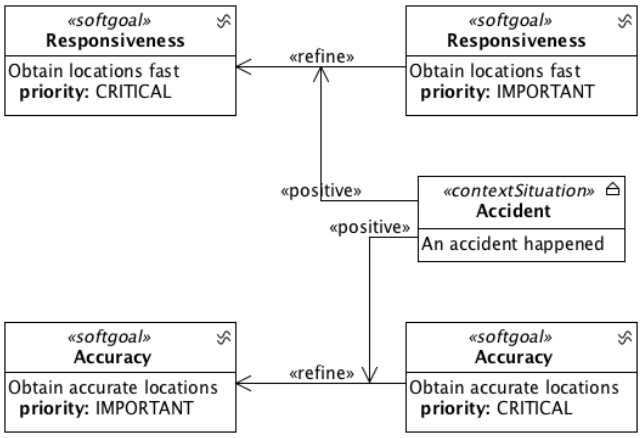

Figura 9.23: Priorización de objetivos II

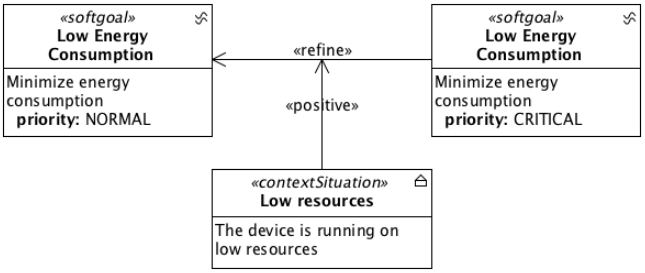

Figura 9.24: Priorización de objetivos III

Entradas al Procedimiento de Evaluación

- **Conjunto de Objetivos** α**:** *Positioning, Indoor Positioning, Outdoor Positioning, Signal Measurement, Position Estimation, Performance, Responsiveness, Scalability, Geographic Scalability, Density Scalability, Low Cost, Low Hardware Cost, Maintenance Time, Operation Time, Deployment Time, Low Energy Consumption, Accuracy, Accuracy Indoors, Accuracy Outdoors, Robustness, Robustness in Transmission Failures, Robustness to unknown signals, Privacy, User's Anonimity / Pseudonimity, User's Intimacy.*

- **Conjunto de Situaciones de Contexto** Δ**:** *Normal operation, At Home, Near Home, Ouside Home, Accident, Low Resources.*

- **Conjunto de Operacionalizaciones** Γ**:** *Infrared, Ultrasounds, Inertial, Accelerometer, Giroscope, Compass, Radio-frequency, Wi-Fi, Bluetooth, ZigBee, RFID, GPS, Proximity-based, Dead Reckoning, Scene Analysis, k-NN, Neural Networks, Probabilistic, SVM, SMP, Triangulation, TOA, TDOA, RTOF, RSS, RSP,*

Network-based, Terminal-based, Terminal-assisted, Load Balancer, Service Replication, Ignore Unknown signals, Provide Feedback and Control, Naming Lattice, Local data.

Traza del procedimiento de evaluación

Se omite el conjunto de justificaciones por simplicidad, asumiendo que las justificaciones que se han proporcionado a lo largo del capítulo son todas ciertas. De esta forma, para la situación de contexto de operación normal del sistema, tenemos:

- Puesto que cualquier operacionalización proporciona una contribución de satisfacción al objetivo *Signal Measurement*, este objetivo se encuentra satisfecho (las operacionalizaciones que lo satisfacen serán determinadas por las restricciones para satisfacer los demás objetivos).

- Puesto que cualquier operacionalización proporciona una contribución de satisfacción al objetivo *Position Estimation*, este objetivo se encuentra satisfecho (las operacionalizaciones que lo satisfacen serán determinadas por las restricciones para satisfacer los demás objetivos).

- El objetivo *Indoor Positioning* se encuentra satisfecho, ya que todos sus subobjetivos (*Signal Measurement*, *Position Estimation*) se encuentran satisfechos.

- Puesto que la operacionalización *GPS* proporciona una contribución de satisfacción al objetivo *Outdoor Positioning*, este objetivo se encuentra satisfecho.

- El objetivo *Positioning* se encuentra satisfecho por estar satisfechos todos sus subojetivos (*Indoor Positioning, Outdoor Positioning*).

- Si se seleccionan las operacionalizaciones *k-NN* y *Proximity-based* se satisface el objetivo *Responsiveness* por recibir múltiples contribuciones positivas.

- Si se seleccionan las operacionalizaciones arquitectónicas *Terminal-assisted, Load Balancer* y *Service Replication*, se satisface el objetivo *Geographic Scalability* por recibir múltiples contribuciones positivas.

- Si se seleccionan las operacionalizaciones arquitectónicas *Load Balancer* y *Service Replication*, se satisface el objetivo *Density Scalability* por recibir múltiples contribuciones positivas.

- El objetivo *Scalability* se encuentra satisfecho dado que sus subobjetivos se encuentran satisfechos (*Geographic Scalability, Density Scalability*).
- El objetivo *Performance* se encuentra satisfecho dado que sus subobjetivos se encuentran satisfechos (*Responsiveness*, *Scalability*).
- Si se seleccionan las operacionalizaciones *ZigBee* y *WiFi*, se satisface el objetivo *Low Hardware Cost* por recibir contribución de satisfacción.
- Si se seleccionan las operacionalizaciones *k-NN* y *Proximity-based* se satisface el objetivo *Operation Time* por recibir múltiples contribuciones positivas.
- Si se seleccionan las operacionalizaciones *k-NN* y *Proximity-based* se produce un conflicto en la satisfacción del objetivo *Deployment Time* por recibir una contribución positiva y una negativa. Puesto que mientras se realiza el entrenamiento necesario para que el método *k-NN* el sistema podría funcionar empleando el método basado en proximidad, y este posee un tiempo muy bajo de despliegue, consideramos que el objetivo *Deployment Time* está parcialmente satisfecho.
- El objetivo *Maintenance Time* se encuentra satisfecho por estarlo todos sus subobjetivos (*Operation Time, Deployment Time*).
- Si se seleccionan las operacionalizaciones *ZigBee* y *WiFi* se satisface el objetivo *Low Energy Consumption* por recibir contribución de satisfacción.
- El objetivo *Low Cost* se encuentra satisfecho por estarlo todos sus subobjetivos (*Low Hardware Cost, Maintenance Time, Low Energy Consumption*).
- Si se seleccionan las operacionalizaciones *k-NN, Proximity-based, ZigBee* y *WiFi*, el objetivo *Accuracy Indoors* se encuentra satisfecho por recibir múltiples contribuciones positivas.
- Si se selecciona la operacionalización *GPS*, el objetivo *Accuracy Outdoors* se encuentra satisfecho.
- El objetivo *Accuracy* se encuentra satisfecho por estarlo todos sus subobjetivos (*Accuracy Indoors, Accuracy Outdoors*).
- Si se seleccionan las operacionalizaciones *Service Replication, WiFi* y *ZigBee*, el objetivo *Robustness in Transmission Failures*

se encuentra satisfecho por recibir múltiples contribuciones positivas.

- Si se selecciona la operacionalización *Ignore Unknown Signals*, el objetivo *Robustness to unknown signals* se encuentra satisfecho por recibir una contribución de satisfacción única.
- El objetivo *Robustness* se encuentra satisfecho por estarlo todos sus subobjetivos (*Robustness in Transmission Failures, Robustness to unknown signals*).
- Si se seleccionan las operacionalizaciones *Provide feedback and control* y *Naming lattice*, el objetivo *User's Anonimity / Pseudonimity* se encuentra satisfecho por recibir múltiples contribuciones positivas.
- Si se selecciona la operacionalización *Local data*, el objetivo *User's Intimacy* se encuentra parcialmente satisfecho por recibir una contribución positiva.
- El objetivo *Privacy* se encuentra satisfecho por estarlo todos sus subobjetivos (*User's Anonimity / Pseudonimity, User's Intimacy*).

De esta forma todos los objetivos se encuentran satisfechos y el conjunto de operacionalizaciones que permite dicha satisfacción es:

- **Operacionalizaciones elegidas** Γ': *ZigBee, WiFi, GPS, k-NN, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*.

Si la salida del algoritmo no hubiera sido satisfactoria, es decir, algún objetivo hubiera quedado insatisfecho, el Ingeniero de Requisitos debería proceder a la iteración entre las fases del método, tratando de elegir diferentes operacionalizaciones que satisfagan todos los requisitos, o bien, buscando nuevas soluciones que no se hubieran considerado con anterioridad. Dicho proceso debe realizarse de manera iterativa hasta que el resultado de la evaluación sea satisfactorio y todos los objetivos hayan sido cubiertos.

Salida del procedimiento de evaluación

De manera similar, se debe proceder para evaluar los objetivos con sus distintas priorizaciones debidas a cambios en el contexto. Finalmente, se obtendría una salida del algoritmo como la que se muestra en la Tabla 9.4. Para cada situación de contexto, se determina el conjunto de operacionalizaciones que satisfacen todos los objetivos con la prioridad deseada.

Situación de Contexto	Operacionalizaciones
Operación Normal	*ZigBee, WiFi, GPS, k-NN, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*
At home	*ZigBee, WiFi, k-NN, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*
Near home	*ZigBee, WiFi, GPS, k-NN, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*
Outside home	*GPS, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*
Low resources	*ZigBee, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*
Accident	*RFID, ZigBee, WiFi, GPS, Probabilistic, k-NN, Proximity-based, Terminal-assisted, Load balancer, Service Replication, Ignore Unknown Signals, Provide Feedback and Control, Naming Lattice, Local data*

Tabla 9.4: Salida del procedimiento de evaluación aplicado al Sistema de Posicionamiento.

Los resultados obtenidos de la aplicación del método REUBI a este caso de estudio son, a modo de resumen:

- Un conjunto de operacionalizaciones que satisfacen los requisitos no funcionales y que permitirán al diseñador, eventualmente, determinar la arquitectura del sistema, los módulos que deben existir en el software y sus conexiones.

- Para cada situación de contexto, las operacionalizaciones que deben tenerse en cuenta para mantener la calidad del sistema que se espera según la prioridad que se otorga a los requisitos bajo dichas circunstancias.

- Las adaptaciones que deben realizarse para que el sistema mantenga las propiedades de calidad ante cambios en el entorno.

- Los eventos que disparan las adaptaciones que deben realizarse, dados por los atributos que caracterizan las situaciones de contexto y sus valores asociados.

9.4 Resumen

Con el objetivo final de validar la parte de este trabajo de investigación que corresponde al método REUBI, este capítulo ha mostrado su aplicabilidad para analizar los requisitos de un Sistema de Posicionamiento. Dicho sistema es un ejemplo que se puede considerar como bastante representativo por ser comúnmente utilizado en la mayoría de los sistemas ubicuos, así como por la heterogeneidad de soluciones tecnológicas existentes en la bibliografía relacionada.

El Grafo de Interdependencia permite estudiar cómo los objetivos son refinados por otros de menor granularidad, así como las implicaciones que tienen las diferentes decisiones que se toman habitualmente para su satisfacción. En numerosas ocasiones dichas decisiones se toman con el único propósito de incluir los últimos desarrollos tecnológicos, mientras que no se considera el impacto que ello tiene en otro tipo de requisitos, especialmente los no funcionales. La aplicación de REUBI permite abordar de manera sistemática el tratamiento de dichos requisitos, de manera que se tomen aquellas decisiones que proporcionarán un diseño de calidad para el software, y no guiado por los últimos avances tecnológicos.

La salida que proporciona REUBI tras la aplicación del procedimiento de evaluación es particularmente útil para los diseñadores de software que se encarguen de realizar el diseño del sistema de posicionamiento, puesto que de su análisis pueden deducir la arquitectura que debe tener el software, el conjunto de módulos que se deben diseñar, sus interconexiones, la tecnología que se debe emplear, las adaptaciones que se deben hacer y los eventos que determinan dichas adaptaciones, tanto en tiempo de diseño como en ejecución.

10 APLICACIÓN DE SCUBI Y MDUBI PARA DISEÑAR UN SISTEMA DE POSICIONAMIENTO

Índice

10.1 Introducción

Continuando con el método MDUBI propuesto en el Capítulo 8, una vez obtenida la especificación de requisitos del servicio de localización aplicando el método REUBI, este capítulo tiene como principal objetivo validar el método MDUBI mostrando cómo derivar el diseño de dicho servicio mediante la aplicación de las reglas de transformación especificadas en el Capítulo 8.

En la Sección 10.2 se mostrarán, paso a paso, las entidades resultantes de la aplicación de las reglas de transformación al modelo de requisitos obtenido en el capítulo anterior, así como las relaciones existentes entre las entidades generadas.

10.2 Aplicación de MDUBI

En esta sección se muestra el resultado de la aplicación de las reglas de transformación de MDUBI. En las siguientes secciones se detallan las interfaces resultantes, los tipos de datos necesarios, los componentes que implementan las interfaces y su ensamblado en servicios.

10.2.1 Interfaces

Como se indicaba en el Capítulo 7, distinguimos tres tipos de interfaces: funcionales, de control y de contexto. Mediante la aplicación de diferentes reglas de transformación podemos obtener cada una de ellas de la forma en la que se indica a continuación.

Interfaces funcionales

Las interfaces funcionales se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a cada uno de los *goals* que se encuentren en el modelo de REUBI. Así, como resultado de esta aplicación, obtenemos las siguientes interfaces funcionales (Figura 10.1):

Interfaces funcionales generadas

- `ISignalMeasurement`: esta interfaz contará con un método cuya responsabilidad será realizar una medición de una señal.
- `IPositionEstimation`: esta interfaz contará con un método cuya responsabilidad será realizar una estimación de una posición.
- `IIndoorPositioning`: esta interfaz contará con un método cuya responsabilidad será realizar el posicionamiento en interiores.
- `IOutdoorPositioning`: esta interfaz contará con un método cuya responsabilidad será realizar el posicionamiento en exteriores.
- `IPositioning`: esta interfaz contará con un método cuya responsabilidad será realizar posicionamiento.

Interfaces de control

De igual manera, las interfaces de control se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a cada uno de los *goals* que se encuentren en el modelo de REUBI. Así, como resultado de esta aplicación, obtenemos las siguientes interfaces de control (Figura 10.2):

Interfaces de control generadas

- `ISignalMeasurementControl`: esta interfaz contará con un método encargado de seleccionar el mecanismo de medición de la señal.

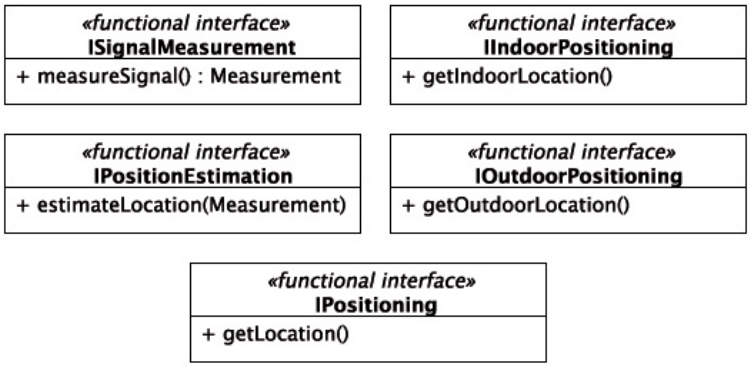

Figura 10.1: Interfaces funcionales generadas.

- `IPositionEstimationControl`: esta interfaz contará con un método encargado de seleccionar la técnica de estimación de la ubicación.
- `IPositioningControl`: esta interfaz contará con un método encargado de seleccionar el tipo de posicionamiento (interior o exterior).

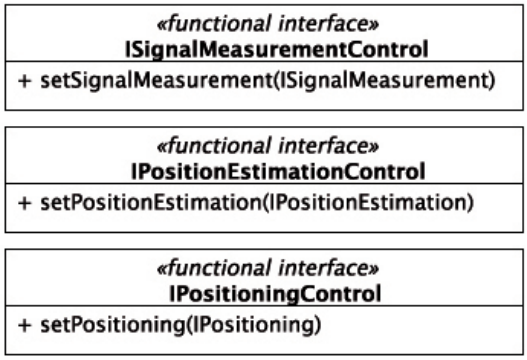

Figura 10.2: Interfaces de control generadas.

Interfaces de contexto

Las interfaces de contexto se obtienen mediante la aplicación de la regla *De Datos de Contexto a Componente de Contexto* a cada uno de los atributos de contexto presentes en el modelo de requisitos. Así, como resultado de esta transformación, se obtienen las siguientes interfaces de contexto (Figura 10.3):

Interfaces de contexto generadas

- `ISignalMACAddress`: esta interfaz contiene un método para notificar eventos de cambio de la dirección MAC del dispositivo emisor de la señal.

- `IReceivedSignalStrength`: esta interfaz contiene un método para notificar eventos de cambio en la intensidad de señal recibida.
- `IBatteryLevel`: esta interfaz contiene un método para notificar eventos de cambio en el nivel de batería del dispositivo.
- `IAlarm`: esta interfaz contiene un método para notificar eventos de alarma.

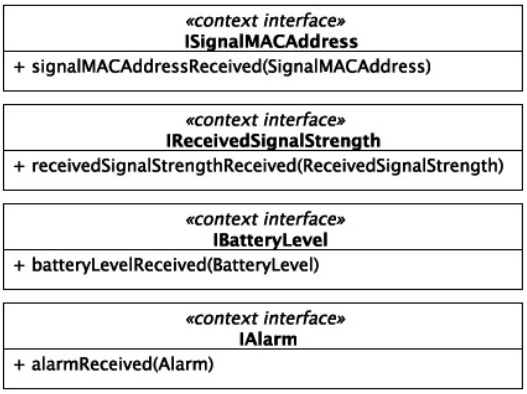

Figura 10.3: Interfaces de contexto generadas.

10.2.2 Tipos de datos

Los tipos de datos que se necesitan dentro del servicio de localización se obtienen mediante la aplicación de dos reglas. Por un lado, mediante la aplicación de la regla *De REcurso a Clase* a cada uno de los recursos dentro del modelo de requisitos obtenemos la siguiente clase (Figura 10.4):

Tipos de datos generados

- `Measurement`: representa una medición de un conjunto de señales.

Por otra parte, mediante la aplicación de la regla *De Datos de Contexto a Componente de Contexto* a cada uno de los atributos de contexto presentes en el modelo de requisitos, obtenemos las siguientes clases (Figura 10.4):

Tipos de datos generados para manejar eventos de contexto

- `SignalMACAddress`: representa una dirección MAC de una señal.
- `ReceivedSignalStrength`: representa una medición de intensidad de señal.

- `BatteryLevel`: representa una medición del nivel de batería de un dispositivo.
- `Alarm`: representa la ocurrencia de una alarma.

Podemos observar que estos últimos tipos de datos no son datos estructurados, sino que solamente se componen de un único valor. A nivel conceptual pueden mantenerse como tipos de datos, pero cuando el sistema sea implementado, se podrán hacer corresponder con los tipos primitivos existentes en la plataforma que se escoja.

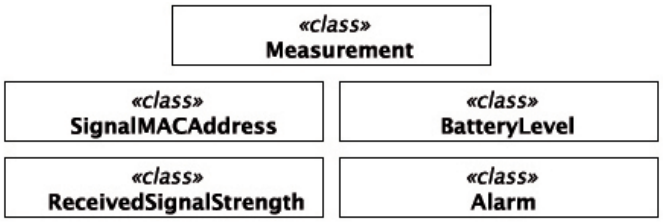

Figura 10.4: Tipos de datos generados.

10.2.3 Componentes

Como se indicó en el Capítulo 7, se distinguen diferentes tipos de componentes: funcionales, de gestión, de control, de contexto y de adaptación. A continuación se muestran los componentes resultantes de la aplicación de las reglas de transformación a algunas de las entidades de modelo de requisitos.

Componentes funcionales

Los componentes funcionales se obtienen mediante la aplicación de la regla *De Operacionalización a Componente Funcional* a aquellas operacionalizaciones que realizan una contribución positiva a un *goal* y que hayan sido seleccionadas en el proceso de selección de REUBI. Además, los componentes obtenidos por la aplicación de esta regla implementan la interfaz correspondiente a la transformación del *goal* al que contribuyen. Así, obtendremos los siguientes componentes:

Componentes funcionales generados

- Componentes que implementan `ISignalMeasurement`: `WifiComponent`, `ZigBeeComponent`, `RFIDComponent`.
- Componentes que implementan `IPositionEstimation`: `kNNComponent`, `ProximityComponent`, `ProbabilisticComponent`.

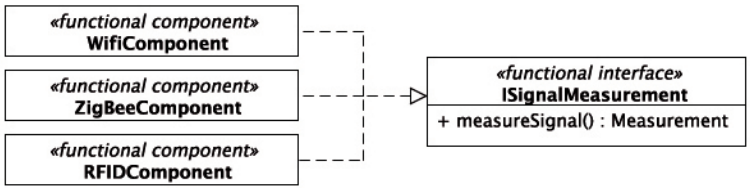

Figura 10.5: Componentes que implementan `ISignalMeasurement`.

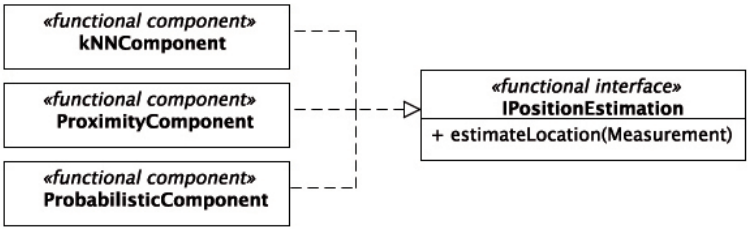

Figura 10.6: Componentes que implementan `IPositionEstimation`.

- Componentes que implementan `IOutdoorPositioning`: `GPS-Component`.

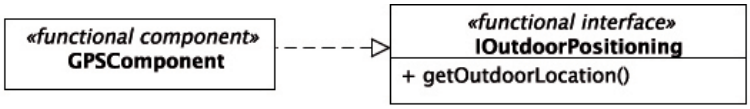

Figura 10.7: Componentes que implementan `IOutdoorPositioning`.

Componentes de gestión

Los componentes de gestión se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a cada uno de los *goals*. Así, como resultado de la aplicación de esta regla, obtendremos los siguientes componentes:

Componentes de gestión generados

- `SignalMeasurementManager`, que implementa `ISignalMeasurement` y `ISignalMeasurementControl`, encargado de gestionar los componentes responsables de realizar la medición de una señal.
- `PositionEstimationManager`, que implementa `IPosition-Estimation` y `IPositionEstimationControl`, encargado de gestionar los componentes responsables de realizar la estimación de una posición.
- `OutdoorPositioningManager`, que implementa `IOutdoor-Positioning`, encargado de gestionar los componentes responsables de realizar el posicionamiento en exteriores.

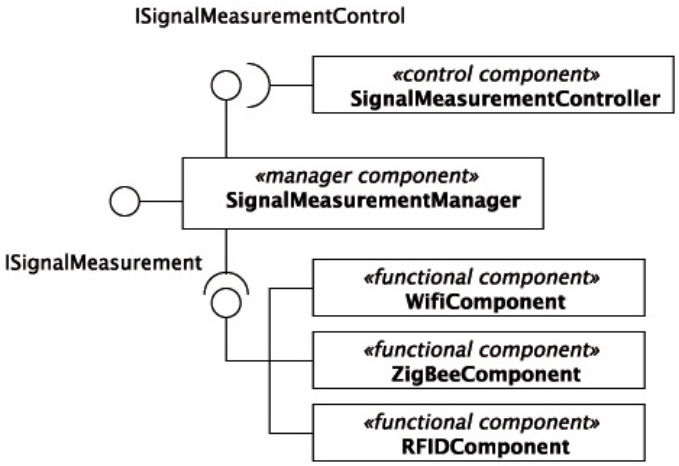

Figura 10.8: Componente de gestión `SignalMeasurementManager`.

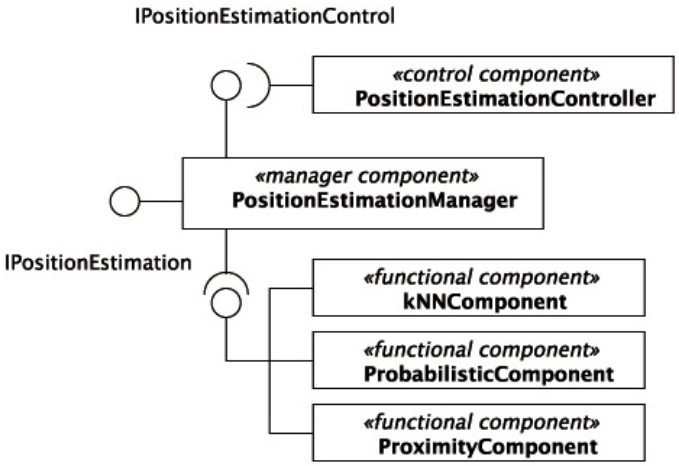

Figura 10.9: Componente de gestión `PositionEstimationManager`.

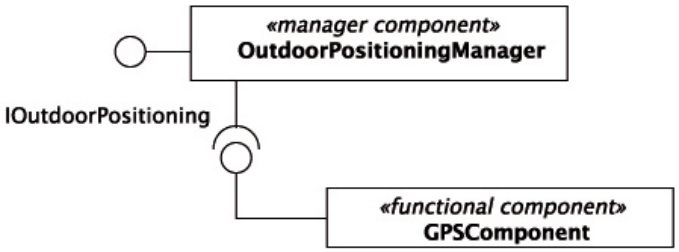

Figura 10.10: Componente de gestión `OutdoorPositioningManager`.

- `IndoorPositioningManager`, que implementa `IIndoorPositioning`, encargado de gestionar los componentes responsables de realizar el posicionamiento en interiores.

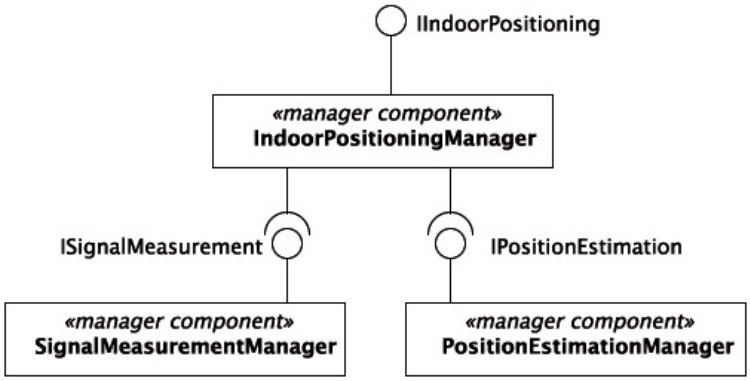

Figura 10.11: Componente de gestión `IndoorPositioningManager`.

- `PositioningManager`, que implementa `IPositioning` y `IPositioning-Control`, encargado de gestionar los componentes responsables de realizar el posicionamiento.

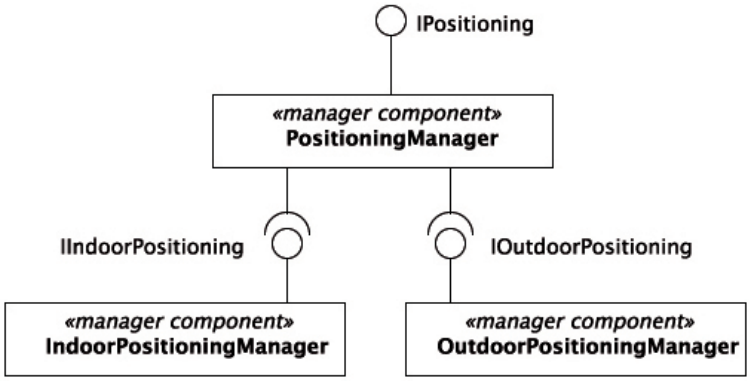

Figura 10.12: Componente de gestión `PositioningManager`.

Nótese que los componentes de gestión creados por la aplicación de esta regla de transformación implementan las correspondientes interfaces funcionales y de control generadas por esta misma regla, de la forma que se presentó en secciones anteriores.

Componentes de control

De manera similar, los componentes de control se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a cada uno de los *goals*. Así, como resultado de la aplicación de esta regla, obtendremos los siguientes componentes:

Componentes de control generados

- `SignalMeasurementController`, que implementa `ISignal MeasurementControl` y controla a `SignalMeasurementManager`.

- `PositionEstimationController`, que implementa `IPositionEstimationControl` y controla a `PositionEstimationManager`.
- `PositioningController`, que implementa `IPositioning Control` y controla a `PositioningManager`.

Nótese que con la aplicación de esta regla de transformación habrían aparecido también los componentes de control `IndoorPositioningControl` y `OutdoorPositioningControl`; sin embargo, y como se discutió anteriormente, se ha optado por eliminar estos componentes del diseño al carecer de utilidad puesto que no hay posibilidad de variación en los componentes controlados.

Componentes de contexto

Los componentes de contexto se obtienen mediante la aplicación de la regla *De Datos de Contexto a Componente de Contexto* a cada uno de los atributos de contexto presentes en el modelo de requisitos. Así, como resultado, se obtienen los siguientes componentes:

Componentes de contexto generados

- `SignalMACAddressProvider` y `SignalMACAddressConsumer`, encargados de transmitir y recibir, respectivamente, eventos de cambio en la dirección MAC de un dispositivo, que se comunican a través de la interfaz `ISignalMACAddress`.

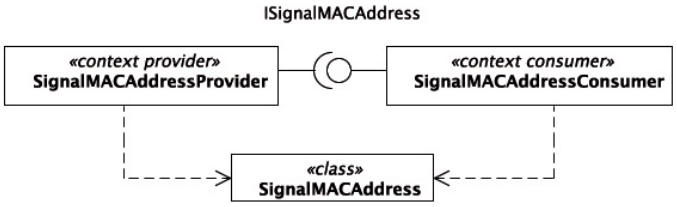

Figura 10.13: Componentes de contexto para `SignalMACAddress`.

- `ReceivedSignalStrengthProvider` y `ReceivedSignal StrengthConsumer`, encargados de transmitir y recibir, respectivamente, eventos de cambio en la intensidad de señal recibida, que se comunican a través de la interfaz `IReceivedSignalStrength`.
- `BatteryLevelProvider` y `BatteryLevelConsumer`, encargados de transmitir y recibir, respectivamente, eventos de cambio en el nivel de la batería de un dispositivo, que se comunican a través de la interfaz `IBatteryLevel`.
- `AlarmProvider` y `AlarmConsumer`, encargados de transmitir y recibir, respectivamente, eventos de alarma, que se comunican a través de la interfaz `IAlarm`.

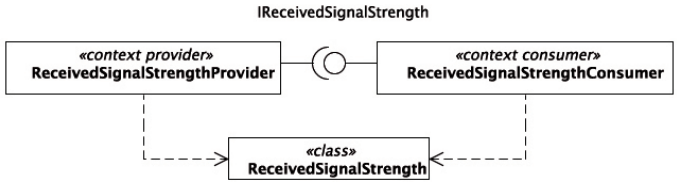

Figura 10.14: Componentes de contexto para `ReceivedSignalStrength`.

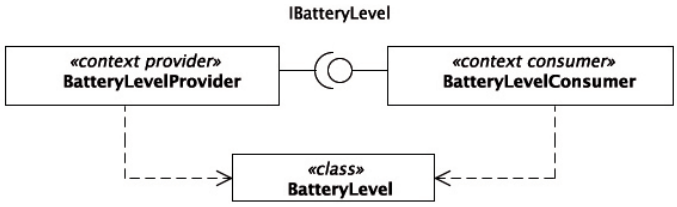

Figura 10.15: Componentes de contexto para `BatteryLevel`.

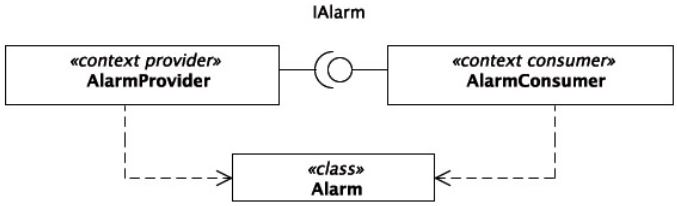

Figura 10.16: Componentes de contexto para `Alarm`.

Componentes de adaptación

Por último, los componentes de adaptación se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a cada uno de los *goals*. Así, como resultado, se obtienen los siguientes componentes:

Componentes de adaptación generados

- `SignalMeasurementAdaptation`, encargado de enviar las adaptaciones al componente `SignalMeasurementController` a través de la interfaz `ISignalMeasurementControl`.
- `PositionEstimationAdaptation`, encargado de enviar las adaptaciones al componente `PositionEstimationController` a través de la interfaz `IPositionEstimationControl`.
- `PositioningAdaptation`, encargado de enviar las adaptaciones al componente `PositioningController` a través de la interfaz `IPositioningControl`.

Nótese que la aplicación de esta regla daría como resultado la aparición de los componentes `IndoorPositioningAdaptation`

y `OutdoorPositioningAdaptation`, pero por las razones mencionadas en secciones anteriores, se ha decidido eliminar dichos componentes del diseño del servicio.

Reglas de adaptación generadas

Finalmente, los componentes de adaptación deben tener las reglas de adaptación correspondientes. Dichas reglas se derivan mediante la aplicación de la regla de transformación *De Contribución a Regla de Adaptación* a cada contribución de una operacionalización a un objetivo (*goal* o *softgoal*), condicionada por la ocurrencia de una situación de contexto. Teniendo en cuenta los resultados obtenidos en la Tabla 9.4, las reglas de adaptación obtenidas por la aplicación de la transformación pueden observarse en la Tabla 10.1.

Regla	**Eventos**	**Condiciones**	**Acciones**
Regla 1	`SignalMAC Address` o `ReceivedSignal Strength`	`MAC == 00:11:22:33:44:55` y `strength >med-high`	`ZigBeeComponent, WifiComponent, kNNComponent, Proximity Component`
Regla 2	`SignalMAC Address` o `ReceivedSignal Strength`	`MAC == 00:11:22:33:44:55` y `strength <= med-high`	`ZigBeeComponent, WifiComponent, GPSComponent, kNNComponent, Proximity Component`
Regla 3	`SignalMAC Address` o `ReceivedSignal Strength`	`MAC != 00:11:22:33:44:55`	`GPSComponent`
Regla 4	`BatteryLevel`	`level <= low`	`ZigBeeComponent, Proximity Component`
Regla 5	`Alarm`	`alarm == on`	`ZigBeeComponent, RFIDComponent, WifiComponent, GPSComponent, kNNComponent, Proximity Component, Probabilistic Component`

Tabla 10.1: Reglas de adaptación generadas por la transformación.

10.2.4 Servicios

Servicios generados

Los servicios necesarios se obtienen mediante la aplicación de la regla *De Goal a Componente de Gestión* a los *goals* de más alto nivel. En este caso, solamente existe uno, el correspondiente a realizar el posicionamiento. Así, como resultado de la aplicación de esta regla, se obtiene `PositioningService`, que contiene `PositioningManager`, el componente que se encarga de realizar el posicionamiento, así como el resto de componentes que se han ido detallando en este capítulo.

10.2.5 Resultado final

Como resultado de la aplicación de las transformaciones propuestas en el Capítulo 8 al modelo de requisitos obtenido en el Capítulo 9, se ha obtenido el diseño mostrado en la Figura 10.17. Además, se ha aplicado el método SCUBI para determinar las interfaces que muestra el servicio, tanto funcionales como de control y de contexto.

10.3 Implementación

Cabe destacar que el diseño obtenido mediante transformación ha sido implementado mediante la aplicación de técnicas de generación de código tal como se indicó en el Capítulo 8 y completado mediante programación manual en aquellos casos en los que no era posible la generación automática de código (e.g. código específico para realizar la medición de una señal de una tecnología concreta o algoritmo para la estimación de una localización). Dicha implementación ha dado como resultado un servicio de localización denominado Sherlock que ha sido o está siendo incorporado y utilizado en varios proyectos de desarrollo de sistemas ubicuos/móviles reales, tales como:

- **Blue Rose:** es un middleware de comunicación de Sistemas Ubicuos que proporciona servicios tales como descubrimiento dinámico de entidades o difusión de eventos. Se pretende que el servicio de localización Sherlock esté integrado en este middleware como servicio básico de soporte.
- **Kora:** es una aplicación móvil de control domótico para Android. El servicio de localización le aporta la ubicación del usuario para que pueda determinar qué elementos cercanos son los que puede controlar desde el dispositivo.

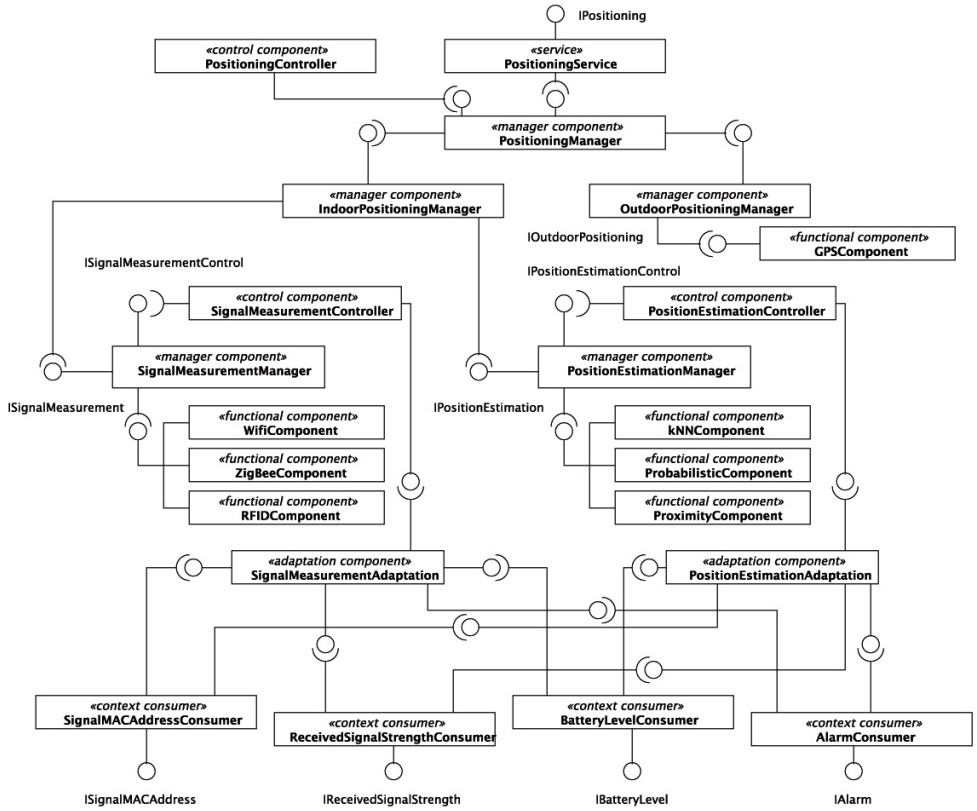

Figura 10.17: Diseño del servicio de localización obtenido por aplicación de las reglas de transformación propuestas por MDUBI.

Sherlock ha sido implementado en el lenguaje de programación Java. Se emplearon dos plataformas distintas en su implementación. Por una parte, se realizó una implementación como un servicio en el sistema operativo Android, haciendo uso de las tecnologías que este sistema proporciona (Wifi y Bluetooth). Por otro lado, se realizó una implementación en Arduino haciendo uso de módulos de tecnología ZigBee. La implementación del servicio fue la misma con la única salvedad de que se sustituyeron los componentes encargados de las tecnologías concretas que se utilizaron.

Además, se incorporó un mecanismo de instalación de *plug-ins* para la incorporación dinámica de nuevos componentes. De esta manera, se estableció la posibilidad de añadir, eliminar o cambiar componentes en tiempo de ejecución, sin necesidad de detener el funcionamiento del sistema de localización.

10.4 Conclusiones

En este capítulo se ha mostrado la aplicación de las reglas de transformación propuestas en MDUBI para derivar de manera semi-automática un modelo de diseño a partir de una especificación de requisitos realizada siguiendo el enfoque de REUBI. El método de diseño SCUBI propuesto complementa la transformación en aquellos casos en los que el resultado generado puede ser refinado para maximizar las propiedades de calidad del diseño.

Este servicio de localización garantiza las propiedades de calidad que se establecieron en la aplicación del método REUBI gracias a tratamiento sistemático de los NFR que se realizó en la etapa de Ingeniería de Requisitos. Es un servicio autoadaptativo, capaz de dar soporte al posicionamiento en interiores y exteriores, y que integra diferentes tecnologías y algoritmos de manera transparente. Además, gracias a la aplicación de REUBI y SCUBI, es posible incorporar nuevas tecnologías o algoritmos mediante su incorporación al modelo de requisitos como una operacionalización nueva que contribuye a satisfacer un *goal*, y su transformación en el componente correspondiente que implementa una interfaz que abstrae su comportamiento.

Parte V

Conclusiones

11 CONCLUSIONES Y TRABAJO FUTURO

Índice

11.1 Conclusiones

Este trabajo de investigación ha tratado de cubrir las carencias existentes en la bibliografía sobre el soporte metodológico para el desarrollo de Sistemas Ubicuos. Se ha realizado un estudio cuidadoso de los métodos de Ingeniería de Requisitos y Diseño para Sistemas Ubicuos, y de los enfoques de Ingeniería dirigida por Modelos, tanto de propósito general como especialmente dedicados a Sistemas Ubicuos. Se ha realizado una propuesta que cubre las etapas de Ingeniería de Requisitos y Diseño, complementada con un conjunto de transformaciones que permiten derivar semiautomáticamente los modelos necesarios para la construcción del software. Por último, se ha mostrado su aplicabilidad mediante el desarrollo de un sistema de posicionamiento híbrido, capaz de dar soporte a la localización en interiores y exteriores, haciendo uso de múltiples tecnologías y métodos de posicionamiento.

Estado de la Técnica

El análisis del estado de la técnica ha servido para aumentar y afianzar el conocimiento sobre diferentes aspectos relacionados con esta tesis doctoral. En cuanto a los métodos de Ingeniería de Requisitos existentes, este análisis ha permitido descubrir las características que presentan habitualmente cada uno de estos métodos, así como detectar carencias importantes en el tratamiento sistemático de los Requisitos No Funcionales. Existen numerosos trabajos cuyo propósito es realizar un tratamiento formal de los requisitos desde las etapas más tempranas del desarrollo de software; sin embargo, muy pocos se centran en conceder un papel fundamental a los Requisitos No Funcionales, los cuales determinan en gran medida la calidad del sistema desarrollado. El *NFR Framework* es el más claro ejemplo de un método de Ingeniería de Requisitos donde se abordan de manera sistemática, y con un

papel fundamental, los Requisitos No Funcionales. No obstante, cuando se ha realizado un estudio de su aplicabilidad al análisis de Sistemas Ubicuos, se han detectado carencias para cubrir los principales rasgos que exhiben los Sistemas Ubicuos, tales como la consciencia del contexto o la variabilidad en la priorización de requisitos.

Por otra parte, se han estudiado métodos de Ingeniería de Requisitos para Sistemas Ubicuos, Dinámicos y Conscientes del Contexto, así como ciertas técnicas de adquisición de requisitos específicas para este ámbito. Dichos métodos abordan en mayor o menor medida algunas de las características especiales de la Computación Ubicua. Particularmente, la mayoría de ellos se centran en el modelado del contexto y su influencia sobre el sistema, y en la adaptación y personalización de estos sistemas dependiendo de las preferencias del usuario. En lo referente a los Requisitos No Funcionales, queda patente la necesidad de realizar un mayor esfuerzo para poder avanzar en la definición y aplicación de estos métodos.

Posteriormente se estudiaron múltiples enfoques para el Diseño de Sistemas Ubicuos, obteniendo como resultado que el paradigma más apropiado para el desarrollo de software es el de la Arquitectura Orientada a Servicios, dadas sus características que facilitan la reutilización de servicios y el descubrimiento dinámico de los mismos, potenciando así la movilidad de los usuarios y la creación de espacios localizados donde pueden ser accedidos. El análisis de las principales propuestas existentes reveló que las metodologías de propósito general ofrecían guías generales basadas en los principios de SOA, que en ocasiones no eran suficientes para guiar al Ingeniero de Software en la tarea de diseño. Además presentaban carencias importantes a la hora de abordar características específicas de la Computación Ubicua. Por otra parte, se analizaron metodologías cuyo fin era el desarrollo de software para Sistemas Ubicuos, pero tras su estudio se observó que su foco principal de atención era la adquisición y el tratamiento del contexto en dichos servicios, en lugar de proponer un método para el diseño general de los servicios que abordase cuestiones tales como la adaptación de los mismos.

De cara a la realización de un proceso más automatizado, se analizaron las técnicas existentes basadas en Ingeniería dirigida por Modelos. El resultado de este análisis demostró que la mayoría de los enfoques se centran en la generación de código o, a lo sumo, en la transformación entre un diseño conceptual y un diseño para la implementación en una plataforma concreta. Ninguno de los enfoques analizados relativos al desarrollo de Sistemas Ubicuos abordaba el problema desde el nivel independiente de

la computación, tratando de derivar modelos de diseño a partir de especificaciones de requisitos.

Por último, se analizaron diferentes propuestas para la especificación de procesos. Se descubrió que el estándar SPEM 2.0 era particularmente apropiado para la definición de procesos de Ingeniería del Software, ya que proporcionaba construcciones específicas para representar las actividades que habitualmente se llevan a cabo en este tipo de procesos. Además, existen herramientas que implementan este estándar, posibilitando la incorporación de nuevos procesos en dichas herramientas para que puedan ser gestionados, y además, facilitando su extensión o modificación.

Propuesta

Del estudio del estado de la técnica se desprende que actualmente existen carencias en las diferentes etapas del ciclo de desarrollo de software para Sistemas Ubicuos. En particular, es necesario un método de Ingeniería de Requisitos para Sistemas Ubicuos que permita al diseñador tratar de manera sistemática los Requisitos No Funcionales. Tomando como objetivos las carencias detectadas durante el análisis de los diferentes métodos, se ha propuesto un método de Ingeniería de Requisitos para Sistemas Ubicuos (*Requirements Engineering for Ubiquitous Systems*, REUBI).

REUBI es un método de Ingeniería de Requisitos basado en objetivos. Define de forma precisa modelos basados en los conceptos de *goal* y *softgoal* para representar y refinar de manera jerárquica los requisitos del sistema que se debe diseñar. Dichos objetivos se representan mediante Grafos de Interdependencia, los cuales muestran las relaciones existentes entre los diferentes objetivos que se deben cumplir en el sistema. En particular, el Grafo de Interdependencia muestra cómo afectaría la toma de diferentes decisiones, arquitectónicas, de diseño, o combinaciones de ellas, a la satisfacción de los objetivos que deben acometerse en el sistema. Además, el método propone el estudio de la influencia del contexto sobre los requisitos. De esta forma, el contexto puede determinar los objetivos que deben satisfacerse en cada momento o cambios en la prioridad que éstos tienen. Por último, el método proporciona un procedimiento de evaluación para comprobar la satisfacción de los requisitos teniendo en cuenta las decisiones de arquitectónicas y de diseño tomadas, la influencia del contexto y la priorización de los objetivos en un momento determinado.

Las principales ventajas de nuestra contribución respecto a otros trabajos realizados en el ámbito de la Ingeniería de Requisitos son:

Ventajas de REUBI

- Representar los requisitos de un Sistema Ubicuo en térmi-

nos de un Grafo de Interdependencia de *goals* y *softgoals* que permite su refinamiento.

- Llevar a cabo un tratamiento sistemático de los requisitos, en especial no funcionales, para que el diseño y desarrollo del software esté guiado por propiedades de calidad y no por los últimos avances tecnológicos.
- Proporcionar un medio para mostrar y analizar los obstáculos que pueden ocurrir y entorpecer la satisfacción de los objetivos.
- Prever el impacto del contexto en los requisitos.
- Incorporar diferentes operacionalizaciones para estudiar el impacto de cada una de ellas en la satisfacción de los requisitos.
- Modelar la existencia de un cambio de prioridades dependiente del contexto en la satisfacción de los requisitos funcionales y no funcionales.
- Evaluar las decisiones tomadas para comprobar la satisfacción de todos los objetivos mediante un procedimiento de evaluación definido.
- Emplear una notación estándar basada en UML.
- Reutilizar el conocimiento generado, tanto para otros desarrollos como futuras versiones del sistema diseñado, mediante la incorporación de nuevas operacionalizaciones al grafo de interdependencia y su estudio sistemático aplicando el procedimiento de evaluación definido.
- Permitir el análisis de otro tipo de sistemas, tales como sistemas dinámicos, adaptativos o conscientes del contexto, ya que el método tiene en cuenta características que pueden encontrarse en este tipo de sistemas.

Adicionalmente, es necesario un método de Diseño para Sistemas Ubicuos que guíe al ingeniero en esta tarea. Partiendo de las carencias detectadas durante el análisis de los diferentes métodos, se ha propuesto un método de Diseño de Servicios basado en Componentes para Sistemas Ubicuos (*Service Component-based Design Method for Ubiquitous Systems*, SCUBI).

SCUBI propone un proceso *bottom-up* en el que se determinen los componentes básicos del sistema y se combinen para dar lugar a estructuras de mayor complejidad que conformen los servicios.

De esta manera, se permite la reutilización tanto a nivel de servicio como a nivel de los componentes que conforman el servicio. Además, el método promueve la incorporación de estructuras que permitan adquirir y manipular el contexto, de manera que en base a cambios ocurridos en el mismo puedan llevarse a cabo adaptaciones en el servicio que permitan el cumplimiento de los requisitos, tanto funcionales como no funcionales, que se esperan del mismo.

Las principales ventajas de nuestra contribución respecto a otros trabajos realizados en el ámbito del Diseño de servicios para Sistemas Ubicuos son:

Ventajas de SCUBI

- Derivar un diseño que garantice las propiedades de calidad a partir de la información obtenida durante la etapa de Ingeniería de Requisitos.
- Proponer un metamodelo que indique cómo deben dividirse los servicios en diferentes componentes, cada uno con responsabilidades concretas y bien definidas.
- Encapsular decisiones de diseño en componentes que puedan ser añadidos, eliminados o intercambiados de manera sencilla y con un impacto mínimo en el resto del sistema.
- Establecer mecanismos de control de la configuración de los componentes y servicios, permitiendo realizar cambios a nivel funcional (operaciones realizadas) y no funcional (propiedades de calidad vinculadas al software).
- Incorporar componentes que permitan la reorganización del software, incluso en tiempo de ejecución, adaptándose a cambios ocurridos en el contexto.
- Incorporar la presencia de tres interfaces diferentes para un servicio, encargadas de la funcionalidad, el control y los cambios según el contexto, respectivamente, y que son implementadas por los componentes correspondientes.
- Promover la reutilización de artefactos (diseño e implementación) tanto a nivel de componentes como a nivel de servicios.

Ventajas de MDUBI

REUBI y SCUBI han sido incorporados dentro de un método orientado a guiar al ingeniero de software en el proceso de desarrollo de software siguiendo un enfoque dirigido por modelos, el cual se ha denominado *A Model-Driven Approach for the Development of Ubiquitous Systems* (MDUBI). A diferencia de otros

enfoques existentes en este ámbito, MDUBI aborda el problema desde el nivel CIM (independiente de la computación), partiendo de una especificación de requisitos conforme al metamodelo de REUBI. El conjunto de reglas de transformación propuestas permite extraer un diseño que cumple con los requisitos expuestos en esta especificación, y que es conforme al metamodelo de SCUBI, constituyendo un modelo en el nivel PIM (independiente de la plataforma). Posteriormente, este modelo basado en SCUBI puede hacerse corresponder con modelos en el nivel PSM (específico de la plataforma) para tratar de obtener implementaciones concretas. En este trabajo se han realizado correspondencias con Java y WSDL para la implementación de los servicios. Finalmente, MDUBI aplica estrategias de generación de código para obtener la implementación del software.

Por último, la propuesta se ha definido conforme al estándar SPEM 2.0 para la especificación de procesos de Ingeniería del Software. Esto presenta los siguientes beneficios:

Ventajas de SPEM

- Puede ser comprendido por un gran número de profesionales al emplearse una notación estándar.
- Puede incorporarse en numerosas herramientas de gestión de procesos que implementan el estándar.
- Puede modificarse o extenderse con facilidad para adecuarse mejor a las necesidades de un proyecto particular, o de un dominio de aplicación concreto, permitiendo la inclusión de nuevas actividades o la sustitución de algunas de las técnicas propuestas por otras que sean de mayor conocimiento para el ingeniero que las aplica.

Como premisas para llevar a cabo una aplicación efectiva de la propuesta se destacan:

- Requiere un aprendizaje del método y de su notación.
- Requiere que el Ingeniero de Software sea una persona experimentada y conozca la variedad de soluciones que existen para dar solución a los diferentes requisitos, y el impacto que dichas decisiones tienen sobre los requisitos no funcionales. No obstante, el método ayuda al ingeniero a involucrarse y no tomar decisiones sin fundamento. El método le obliga a ser sistemático, aprovecha sus conocimientos y le incita a completar su formación, para poder tomar la mejor decisión.

- REUBI puede involucrar un mayor tiempo y esfuerzo que otras técnicas de Ingeniería de Requisitos tradicionales, pero combinada con técnicas de diseño que hacen uso del análisis realizado, puede reducir el tiempo global de desarrollo del proyecto.

Caso de Estudio

Para demostrar la aplicabilidad del método, se ha elegido un caso de estudio que resulta un ejemplo representativo de los Sistemas Ubicuos: el análisis de los requisitos, el diseño y la implementación de un Sistema de Posicionamiento. Más concretamente, se ha realizado el desarrollo de un sistema de localización para ofrecer posicionamiento tanto en interiores como en exteriores. El análisis de la bibliografía sobre el tema revela una amplia variedad de arquitecturas empleadas (basada en red, basada en terminal, asistida por terminal), métodos (triangulación, *dead-reckoning*, proximidad, *scene analysis*) y tecnologías de posicionamiento (infrarrojos, ultrasonidos, radio frecuencia, dispositivos inerciales). Las características del sistema requieren la aplicación de la propuesta, ya que el sistema debe ser diseñado como un servicio, se debe adaptar a interiores y a exteriores y también al contexto que viene dado por las características de las tecnologías o técnicas empleadas en cada instanciación, y que pueden variar durante el uso del sistema. Cada una de estas alternativas presenta diferentes propiedades de calidad, las cuales deben ser cuidadosamente consideradas para determinar su idoneidad para satisfacer los objetivos que se persiguen. El servicio de localización desarrollado, llamado Sherlock, es un sistema real que puede emplearse para dar soporte a otros Sistemas Ubicuos a la vez que constituye una fuente importante de información sobre el contexto del usuario.

La propuesta ha sido aplicada exitosamente para el desarrollo de este sistema, a la vez que, tras su aplicación práctica, se han detectado algunas posibles mejoras y extensiones a tener en cuenta para potenciar la propuesta, las cuales se describen en la siguiente sección como trabajos futuros.

11.2 Trabajo futuro

La propuesta realizada en esta tesis doctoral abarca la mayor parte del ciclo de vida del software; sin embargo, existen aspectos que podrían ser mejorados o extendidos como resultado de su aplicación al desarrollo de otros sistemas. Algunas de las mejoras y extensiones que podrían llevarse a cabo:

Trabajo Futuro

- Definir vistas sobre los modelos propuestos, ya que en

ocasiones pueden alcanzar una gran complejidad y su legibilidad se dificulta.

- Definir un mecanismo de «empaquetado» que agrupe el trabajo realizado en el análisis de parte de los requisitos de un sistema, de manera que puedan ser organizados fácilmente, e incluso reutilizados en otros sistemas de características similares. El uso de catálogos de requisitos es útil para los Ingenieros de Requisitos, ya que pueden analizar qué relaciones habituales existen entre los requisitos no funcionales y qué decisiones se han tomado para satisfacerlos en otros proyectos.

- Estudiar un mayor número de situaciones recurrentes y recopilarlas en términos de patrones de requisitos que pueden ser reutilizados. Así mismo, cabría la posibilidad de estudiar cómo los patrones de requisitos pueden tener asociados sus correspondientes patrones de diseño.

- Estudiar la posibilidad de aplicar, con las necesarias modificaciones y adaptaciones, la propuesta realizada en otros paradigmas, marcos conceptuales y dominios de aplicación, haciendo uso a su vez de las tecnologías asociadas a estos, tales como *Internet of Things* o *Cloud Computing*.

- Aunque existe soporte para la aplicación del método gracias a los plug-ins existentes para el IDE Eclipse, es necesario aumentar este soporte para automatizar y facilitar algunas de las tareas propuestas. Así, sería conveniente contar con un editor que incorpore la notación propuesta y permita aplicar el procedimiento de evaluación, las transformaciones entre modelos o la generación de código de manera semiautomática. La aplicación manual del método puede resultar un proceso costoso y propenso a errores, por lo que contar con una herramienta como ésta podría resultar de gran ayuda. Una posibilidad que facilitaría la integración con el trabajo que ya ha sido realizado consistiría en la generación de una herramienta gráfica haciendo uso del *Graphic Modeling Framework* (GMF) de Eclipse. Esto, junto con los metamodelos implementados en EMF, las reglas de transformación en ATL, y las plantillas de generación de código en Acceleo, podrían exportarse como un plugin que pudiera extender el IDE Eclipse y facilitar la aplicación del proceso.

- Establecer transformaciones de modelos que permitan obtener los tests que verifican la corrección de la implementa-

ción realizada, posibilitando la aplicación de metodologías ágiles basadas en *Test-Driven Development* junto con la propuesta realizada.

- Por último, aunque el método ha sido aplicado con éxito al desarrollo de un Sistema de Posicionamiento, el cual ha sido construido teniendo en cuenta los resultados obtenidos del estudio, creemos que es necesario aplicarlo a un mayor número de Sistemas Ubicuos para obtener una mayor experiencia sobre su usabilidad y completar aquellos aspectos comunes y particulares a otros sistemas ubicuos.

12 CONCLUSIONS AND FUTURE WORK

Índice

12.1 Conclusions

This research work has aimed to fill the existing lacks in the bibliography about methodological support for the development of Ubiquitous Systems. It has conducted a careful study of the methods for Requirements Engineering and Design for Ubiquitous Systems, and of the approaches based on Model-driven Engineering, both for general-purpose systems and for Ubiquitous Computing; a proposal has been made covering the stages of Requirements Engineering and Design, complemented with a set of transformations that enable the semi automatic derivation of the necessary models to build software; and its applicability has been depicted by means of the development of a hybrid positioning system, capable of supporting localization both indoors and outdoors, making use of multiple technologies and positioning methods.

State of the art

The analysis of the state of the art served to increase and maintain the knowledge about different aspects related to this doctoral thesis. Regarding the existing Requirements Engineering methods, this analysis permitted to discover the usual features of each of these methods, as well as detecting important shortcomings in the systematic assessment of NFRs. Numerous works, whose purpose is to perform a formal treatment of requirements since the earliest stages of the software development process, exist; however, just a few of them give a fundamental role to NFRs, which mostly determine the quality of the developed system. The NFR Framework is the quintessential example of a Requirements Engineering method where NFRs add address systematically and with a major role. Nevertheless, when its applicability to the analysis of Ubiquitous Systems has been tested, several lacks were discovered to fulfill the main features that Ubi-

quitous Systems exhibit, such as context-awareness or variability in the prioritization of requirements.

Besides, some other methods for the Requirements Engineering of Ubiquitous, Dynamic and Context-aware Systems have been studied, as well as requirements acquisition techniques specific for this field. Such methods address to a certain extent some of the special features of Ubiquitous Computing. Particularly, most of them are focused on context modeling and its influence on the system, and on the adaptation and customization of those systems depending on user's preferences. Regarding NFRs, there is a clear need to perform a bigger effort to make an advance in the definition and application of these methods.

Por otra parte, se han estudiado métodos de Ingeniería de Requisitos para Sistemas Ubicuos, Dinámicos y Conscientes del Contexto, así como ciertas técnicas de adquisición de requisitos específicas para este ámbito. Dichos métodos abordan en mayor o menor medida algunas de las características especiales de la Computación Ubicua. Particularmente, la mayoría de ellos se centran en el modelado del contexto y su influencia sobre el sistema, y en la adaptación y personalización de estos sistemas dependiendo de las preferencias del usuario. En lo referente a los Requisitos No Funcionales, queda patente la necesidad de realizar un mayor esfuerzo para poder avanzar en la definición y aplicación de estos métodos.

Later, several approaches for the Design of Ubiquitous Systems where studied, obtaining as a result that the most suitable paradigm for the development of software is the Service Oriented Architecture, given its characteristics that ease the reuse of services and their dynamic discovery, enhancing the users' mobility and the creation of spaces with localized access to the services. The analysis of the main proposals revealed that general-purpose methodologies offer just general guidelines based on SOA principles, which oftentimes are not enough to guide the Software Engineer in the Design task, and furthermore, the presented important shortcomings to address specific features of Ubiquitous Computing. On the other hand, software development methodologies for Ubiquitous Systems were studied, but their analysis showed that their main focus was the acquisition and management of context in those services, instead of proposing a method for the general design of some aspects such as their adaptation.

Towards the proposition of a more automated process, Model-driven Engineering techniques were analyzed. The result of this study revealed that most of the approaches are focused on the code generation or, at most, in the transformation between a conceptual design and an implementation design in a concrete

platform. None of the studied approaches for the develeopment of Ubiquitous Systems addressed the problem from the computation independent level, trying to derive design models from requirements specifications.

Last, different proposals for the specification of processes were considered. It was discovered that the SPEM 2.0 standard was particularly suitable for the definition of Software Engineering Processes, since it provides specific constructions to represent the usual activities in this kind of processes. Moreover, there are tools that implement this standard, enabling the incorporation of new processes into those tools in order to be managed and also, easing their extension or modification.

Proposal

From the study of the state of the art, we could infer that currently there are some lacks and shortcomings in the different stages of the software development cycle for Ubiquitous Systems. In particular, a Requirements Engineering method is needed in order to allow the designer to systematically deal with NFRs. Taking the observed problems as goals to solve, a Requirements Engineering Method for Ubiquitous Systems (REUBI) has been proposed.

REUBI is a goal-based Requirements Engineering method. It accurately defined models based in the concepts of goal and softgoal in order to represent and decompose in a hierarchical manner the requirements of the system under design. Such objectives are represented by means of Interdependency Graph, which depict the existing relationships between different objectives that must be fulfilled in the system. More precisely, the Interdepency Graph shoes how the decision making process to chose combinations of architectural or design alternatives would affect to the satisfaction of the system goals. Furthermore, the method proposes the study of the influence of context on the requirements. In this way, context can determine which objectives have to be accomplished at each moment or changes in the priority they have. Last, the method provides an evaluation procedure to check the satisfaction of the requirements taking into consideration the architectural and design decisions made, the influence of context and the priority of the requirements at a given moment.

The main advantages of our contribution respect to other works in the scope of Requirements Engineering are:

Advantages of REUBI

- Representing the requirements of a Ubiquitous System in terms of an Interdependency Graph of goals and softgoals that depicts their decomposition.
- Performing a systematic treatment of the requirements, specially regarding NFRs, oriented to achieve a quality-driven

design and development process, instead of a technology driven process.

- Providing a means to show and analyze the obstacles that may arise and thwart the satisfaction of the requirements.
- Showing the impact of context in the requirements.
- Modeling the existence of a context-dependent priority change in the satisfaction of both functional and non functional requirements.
- Evaluating the decisions made in order to check the satisfaction of every objective.
- Making use of a standard notation based on UML.
- Reusing the generated knowledge, both for other development projects and future versions of the envisioned system, by means of the incorporation of new operationalizations to the Interdependency Graph and its systematic study with the proposed evaluation procedure.
- Analyzing some other types of systems, such as dynamic, adaptive or context-aware systems, since the method incorporates characteristics that can be found in this kind of systems.

Additionally, a Design method for Ubiquitous Systems to guide the engineer in this task is needed. Stemming from the detected problems during the analysis of the different methods, a Service Component-based Design Method for Ubiquitous Systems (SCUBI) has been proposed.

SCUBI proposes a bottom-up process where the basic components of the systems are determined and combined to create more complex structures to build services. In this way, reuse is promoted both at service and component levels. Moreover, the method encourages the incorporation of structures to acquire and manage context, in such a way that changes occurred in context can lead to adaptations in the service in order to fulfill the requirements, functional and non functional, that are expected.

The main advantages of our contribution respect to other Service Design methods for Ubiquitous Systems are:

Advantages of SCUBI

- Deriving a design that guarantees the quality properties starting from the information acquired during the stage of Requirements Engineering.

- Proposing a metamodel that depicts how services should be split in components, each one of them with well-defined responsibilities.
- Encapsulating design decisions into components that can be added, removed or exchanged easily and with a minimum impact in the rest of the system.
- Establishing control mechanisms for the configuration of components and services, enabling to perform changes to a functional level (operations performed by the system) and non functional (quality properties).
- Incorporating structures that allow software reorganization, even in runtime, adapting to contextual changes.
- Incorporating the presence of three different interfaces for a service, in charge of functionality, control and contextual changes, respectively.
- Promoting the reutilization of artifacts both at service and component levels.

Advantages of MDUBI

REUBI and SCUBI have been incorporated into a process to guide the software engineer in the development of software following a model-driven approach, which has been named A Model-Driven Approach for the Development of Ubiquitous Systems (MDUBI). Unlike other existing approaches in this field, MDUBI addresses the problem from the CIM level (computation independent), starting from a requirements specification conforming the REUBI metamodel. The set of proposed transformation rules allows to extract a design that fulfills the requirements contained in this specification, and which conforms to the SCUBI metamodel, leading to a model at the PIM level (platform independent). Afterwards, the SCUBI model can be mapped to models at the PSM level (platform specific) in order to achieve a concrete implementation. In this case, mappings to Java and WSDL for the implementation of services have been proposed. Finally, MDUBI applies strategies for code generation in order to obtain a software implementation.

Last, but not least, the proposal has been defined conforming to the SPEM 2.0 standard for the specification of Software Engineering processes. This has the following benefits:

- It can be understood by a big number of professionals since it uses a standard notation.

- It can be incorporated to a number of process management tools that implement this standard.

- It can be modified or extended easily in order to better fit the needs of a particular project, or a concrete application domain, enabling the incorporation of new activities or the substitution of some of the proposed techniques by others which are more familiar to the engineer who applies them.

In order to achieve an effective application of the proposal, we can highlight:

- It requires a learning process to know the method and notation.

- It requires the software engineer to be an experienced person and to know the variety of existing solutions to fulfill the different requirements, and the impact those solutions have on the NFRs. Nevertheless, the method helps the engineer to get involved and not making unfounded decisions. The method forces him/her to be systematic, takes advantage of his/her knowledge and promotes to complete his/her formation in order to make the most suitable decision.

- REUBI can involve a longer time and effort than other Requirements Engineering techniques, but combined with design techniques that make use of the performed analysis, it can reduce the overall development time.

Case Study

In order to demonstrate the applicability of the method, a case study of a representative example of Ubiquitous System has been chosen: the analysis of the requirements, design and implementation of a Positioning System. More precisely, the development of a localization system to provide positioning both indoors and outdoors has been made. The analysis of the bibliography about this topic reveals a wide variety of architectures (network-based, terminal-based, terminal-assisted), methods (triangulation, dead-reckoning, proximity, scene analysis) and positioning technologies (infrared, ultrasound, radio-frequency, inertial devices). Each one of these alternatives has different quality properties, which must be carefully considered to determine their suitability to satisfy the pursued goals. The developed positioning system, named Sherlock, is a real system that can be employ to support other Ubiquitous Systems at the same time that constitutes an important source of information about the user's context.

The proposal has been successfully applied to the development of this system, and after its practical application, some improvements and extensions have been detected in order to enhance the proposal, which are described in the next section as future work.

12.2 Future Work

The proposal contained in this doctoral thesis embraces most of the stages of the software development lifecycle; however, there are aspects that could be improved or extended as a result of its application to the development of other systems. Some of the improvements and extensions that could be performed are:

Future Work

- Defining views over the proposed models, since in some situations they can get very complex and their readability becomes difficult.
- Defining a packaging mechanism to group the work done in the analysis of part of the requirements of a system, in a way that they could be easily organized and even reused in other systems with similar characteristics. The use of requirement catalogues is useful for Requirements Engineers, since they can analyze the usual relationships between NFRs and which decisions have been made to satisfy them in other projects.
- Studying a greater number of recurring situations and compiling them in terms of requirements patterns which can be reused. Moreover, we could study how the requirements patterns may have associated their corresponding design patterns.
- Studying the possibility of applying, with the necessary modifications and adaptations, this proposal in other application fields, such as Internet of Things or Cloud Computing.
- Although there is some technological support for the application of the method thanks to the existing plugins for the Eclipse IDE, it is necessary to augment this support to automatize and ease some of the proposed tasks. Thus, it would be highly convenient to count on a graphical editor that incorporates the proposed notation and allows to apply the evaluation procedure, the model transformations or the code generation strategies in a semi automatic manner. The manual application of this method can be a costly

and error-prone process; therefore, counting on a tool like this could be useful. A possibility to ease the integration of the already done work into a tool would be to generate a graphical modeling tool making use of the Eclipse Graphic Modeling Framework (GMF). This, together with the metamodels implemented in EMF, the transformation rules implemented in ATL, and the code generation templates implemented in Acceleo, could be exported as a plugin to extend the Eclipse IDE and make the application of the process easier.

- Creating model transformations to enable the obtention of tests to verify the correction of the implementation, allowing the application of agile methodologies based on Test-Driven Development together with this proposal.

- Finally, although the method has been successfully applied to the development of a Positioning System, which has been built taking into consideration the results obtained from its study, we believe that it is necessary to apply it to a wider number of Ubiquitous Systems to gain more experience about its usability and complete those common and particular aspects of Ubiquitous Systems.

Parte VI

Apéndices

A PUBLICACIONES DERIVADAS DEL TRABAJO DE INVESTIGACIÓN

Cabe mencionar que como fruto de este trabajo de investigación se han derivado las siguientes publicaciones en diversas revistas de impacto y congresos de reconocido prestigio en el área de la Computación Ubicua y la Ingeniería del Software:

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, Manuel Noguera, José Luis Garrido: **Towards a Reusable Design of a Positioning System for AAL Environments**. *Proceedings of the 1st EvAAL Competition; Communications in Computer and Information Science 2012*.

- Tomás Ruiz-López, Manuel Noguera, María José Rodríguez, José Luis Garrido, Lawrence Chung: **REUBI: A Requirements Engineering Method for Ubiquitous Systems**. *Science of Computer Programming 2012*. JCR index: 1.282

- Carlos Rodríguez-Domínguez, Kawtar Benghazi, Manuel Noguera, José Luis Garrido, María Luisa Rodríguez, Tomás Ruiz-López: **A Communcation Model to Integrate the Request-Response and the Publish-Subscribe Paradigms in Ubiquitous Systems**. *Journal of Sensors 2012*. JCR index: 2.395

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, Manuel Noguera, María José Rodríguez, José Luis Garrido: **Model-Driven Requirements Engineering for Ambient Intelligence Environments**. *Journal of Ambient Intelligence and Smart Enivronments 2013*. JCR index: 1.298

- Carlos Rodríguez-Domínguez, Tomás Ruiz-López, Kawtar Benghazi, Manuel Noguera, José Luis Garrido: **Towards the development of middleware solutions for ubiquitous systems through model-driven engineering techniques**. *Journal of Ambient Intelligence and Smart Environments*. JCR index: 1.298 (pre-seleccionado, bajo revisión)

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, Sergio F. Ochoa, José Luis Garrido: . *Journal of Sensors 2014*. JCR index: 2.395 (pre-seleccionado, bajo revisión)

- Tomás Ruiz-López, José Luis Garrido, Kawtar Benghazi, Lawrence Chung: **A Survey on Indoor Positioning Systems: Foreseeing a Quality Design.** *Distributed Computing and Artificial Intelligence (DCAI 2010)*

- Carlos Rodríguez-Domínguez, Kawtar Benghazi, Manuel Noguera, María José Rodríguez, Tomás Ruiz-López and José Luis Garrido: **Framework de Soporte al Desarrollo Integrado de Sistemas Ubicuos**. *Jornadas de Concurrencia y Sistemas Distribuidos 2011*

- Tomás Ruiz-López, José Luis Garrido, Carlos Rodríguez-Domínguez and Manuel Noguera: **Sherlock: A Hybrid, Adaptive Positioning Service based on Standard Technologies**. *Evaluating AAL Systems through Competitive Benchmarking (EvAAL 2011)*

- Tomás Ruiz-López, Manuel Noguera, María José Rodríguez, José Luis Garrido and Lawrence Chung: **Towards a Systematic Treatment of Non-Functional Requirements in Ubiquitous Systems**. *5th International Symposium of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011)*

- Carlos Rodríguez-Domínguez, Kawtar Benghazi, Manuel Noguera, José Luis Garrido, María Luisa Rodríguez and Tomás Ruiz-López: **Seamless Integration of Communication Paradigms for Ubiquitous Systems**. *5th International Symposium of Ubiquitous Computing and Ambient Intelligence (UCAmI 2011).*

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, Manuel Noguera, María José Rodríguez, José Luis Garrido: **A Model-Driven Approach to Requirements Engineering in Ubiquitous Systems**. *3rd International Symposium on Ambient Intelligence (ISAmI 2012)*

- Carlos Rodríguez-Domínguez, Tomás Ruiz-López, Kawtar Benghazi, José Luis Garrido, Manuel Noguera: **Designing a Middleware-Based Framework to Support Multiparadigm Communications in Ubiquitous Systems**. *3rd International Symposium on Ambient Intelligence (ISAmI 2012)*

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, María José Rodríguez, José Luis Garrido: **Mecanismos de Adaptación basados en Propiedades de Calidad: Un Caso de Estudio de un Servicio de Localización**. *XIII Congreso Internacional de Interacción Persona Ordenador (Interacción 2012)*.

- Rutvij Mehta, Tomás Ruiz-López, Lawrence Chung, Manuel Noguera: **Selecting among alternatives using dependencies: an NFR approach**. *28th Symposium on Applied Computing, Requirements Engineering Track (SAC 2013)*.

- Tomás Ruiz-López, Manuel Noguera, María José Rodríguez, José Luis Garrido: **Requirements systematization through pattern application in Ubiquitous Systems**. *4th International Symposium on Ambient Intelligence (ISAmI 2013).*

- Tomás Ruiz-López, José Luis Garrido, Sam Supakkul, Lawrence Chung: **A Pattern Approach to Dealing with NFRs in Ubiquitous Systems**. *CAiSE Forum 2013*.

- Carlos Rodríguez-Domínguez, Tomás Ruiz-López, Kawtar Benghazi, Manuel Noguera, José Luis Garrido: **A Model-Driven Approach to the Development of Middleware Technologies for Ubiquitous Systems**. *9th International Conference on Intelligent Environments (IE 2013)*.

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, Manuel Noguera, María José Rodríguez, José Luis Garrido: **Towards a Component-based Design of Adaptive, Context-sensitive Services for Ubiquitous Systems**. *8th Workshop on Artificial Intelligence Techniques for Ambient Intelligence (AITAmI 2013)*.

- Álvaro Monares, Sergio F. Ochoa, José A. Pino, Tomás Ruiz-López, Manuel Noguera: **Using Unconventional Awareness Mechanisms to Support Mobile Work**. *Jornadas Chilenas de Computación 2013*.

- Tomás Ruiz-López, Carlos Rodríguez-Domínguez, María José Rodríguez, Sergio F. Ochoa, José Luis Garrido: **Context-aware Self-adaptations: From requirements specification to code generation**. *7th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2013)*.

- Carlos Rodríguez-Domínguez, Tomás Ruiz-López, José Luis Garrido, Manuel Noguera, Kawtar Benghazi: **Leveraging the Model-Driven Architecture for Service Choreography in Ubiquitous Systems**. *7th International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2013)*.

B IMPLEMENTACIÓN DE LAS REGLAS DE TRANSFORMACIÓN EN ATL

```
module ValueModel2REUBI;

-- @path ValueModel=/MDUBI/Metamodels/ValueModel.ecore
-- @path REUBI=/MDUBI/Metamodels/REUBI.ecore

create OUT: REUBI from IN: ValueModel;

helper def: getSoftgoalName(enhancer : ValueModel!ValueEnhancer) : String =
    if(enhancer.type = #MAXIMIZE) then
        'Maximize' + enhancer.name
    else
        'Minimize' + enhancer.name
    endif;

rule Value2Goal{
    from
        value : ValueModel!Value
    to
        goal : REUBI!Goal(
            name <- value.name
        )
}
```

Figura B.1: Regla de transformación `Value2Goal` en ATL.

```
rule ValueEnhancer2Softgoal{
    from
        enhancer : ValueModel!ValueEnhancer
    using{
        value : ValueModel!Value = enhancer.enhances;
    }
    to
        softgoal : REUBI!Softgoal(
            topic <- value.name
        )
    do{
        softgoal.name <- thisModule.getSoftgoalName(enhancer);
    }
}
```

Figura B.2: Regla de transformación `ValueEnhancer2Softgoal` en ATL.

```
module REUBI2SCUBI;

-- @path REUBI=/MDUBI/Metamodels/REUBI.ecore
-- @path SCUBI=/MDUBI/Metamodels/SCUBI.ecore

create OUT: SCUBI from IN: REUBI;

helper def: ruleNo : Integer = 0;
helper def: conditionNo : Integer = 0;

helper def: getComparison(c : REUBI!ComparisonType) : SCUBI!Comparison=
    if c = #EQUAL then
        #EQUAL
    else
        if c = #NOT_EQUAL then
            #NOT_EQUAL
        else
            if c = #GREATER_THAN then
                #GREATER_THAN
            else
                if c = #GREATER_EQ_THAN then
                    #GREATER_EQ_THAN
                else
                    if c = #LESS_THAN then
                        #LESS_THAN
                    else
                        #LESS_EQ_THAN
                    endif
                endif
            endif
        endif
    endif;
```

Figura B.3: Función auxiliar.

```
helper context REUBI!Goal def: getSubGoals() : Set(REUBI!Goal)=
    if(self.decomposedInto.oclIsUndefined()) then
        Set{}
    else
        Set{self.decomposedInto.children->asSet(), self.decomposedInto.children->
            collect(c | c.getSubGoals())}->flatten()->asSet()
    }
    endif;
```

Figura B.4: Función auxiliar

```
rule Goal2Service {
    from
        goal : REUBI!Goal(goal.isDecompositionOf.size() = 0)
    using{
        contributions : Sequence(REUBI!Contribution) =
            goal.contributedBy->select(o | (o.type = #MAKE or o.type = #HELP));
        situations : Sequence(REUBI!ContextSituation) =
            contributions->collect(c | c.influencedBy);
        contextParameters : Sequence(REUBI!ContextAttribute) =
            situations->collect(c | c.attributes)->flatten()->asSet()->asSequence();
        contextData : Sequence(REUBI!ContextData) =
            contextParameters->collect(c | c.data)->asSet()->asSequence();

        input : Sequence(REUBI!Resource) = goal.demands;
        output : REUBI!Resource = goal.provides;

        children : Set(REUBI!Goal) = goal.getSubGoals();
    }
    to
        method : SCUBI!Method(
            name <- 'perform' + goal.name,
            parameters <- input->
            collect(i | thisModule.resolveTemp(i, 'parameter')),
            return <- thisModule.resolveTemp(output, 'parameter')
        ),
        interface : SCUBI!FunctionalInterface(
            name <- 'I' + goal.name,
            methods <- Sequence{method}
        ),
        manager : SCUBI!ManagerComponent(
            name <- goal.name + 'Manager',
            provides <- interface,
            demands <- interface,
            manages <- contributions->
                collect(c | thisModule.resolveTemp(c.source, 'component')),
            controlledBy <- controlInterface
        ),
        stringClass : SCUBI!Class(
            name <- 'String'
        ),
        controlInput : SCUBI!Parameter(
            class <- stringClass,
            name <- 'componentID'
        ),
        controlMethod : SCUBI!Method(
            name <- 'setComponentWithID',
            parameters <- controlInput
        ),
        controlInterface : SCUBI!ControlInterface(
            name <- 'I' + goal.name + 'Control',
            methods <- controlMethod
        ),
        controlComponent : SCUBI!ControlComponent(
            name <- goal.name + 'Controller',
            provides <- controlInterface,
            demands <- controlInterface
        ),
        adaptationComponent : SCUBI!AdaptationComponent(
            name <- goal.name + 'Adapter',
            demands <- controlInterface,
            provides <- contextData->collect(c | thisModule.resolveTemp(c, 'consumerInterface')),
            rules <- contributions->collect(c | thisModule.resolveTemp(c, 'adaptationRule'))
        ),
        service : SCUBI!Service(
            name <- goal.name + 'Service',
            functionalInterface <- interface,
            controlInterface <- Set{controlInterface, children->
                collect(c | thisModule.resolveTemp(c, 'controlInterface'))}->flatten()->asSet(),
            }
            contextInterface <- Set{adaptationComponent.provides, children->
                collect(c | thisModule.resolveTemp(c, 'adaptationComponent').provides)}->
                flatten()->asSet()
            }
        )
}
```

Figura B.5: Regla de transformación `Goal2Service` en ATL.

```
rule Goal2FunctionalInterface {
    from
        goal : REUBI!Goal(goal.isDecompositionOf.size() > 0)
    using{
        contributions : Sequence(REUBI!Contribution) =
            goal.contributedBy->select(o | (o.type = #MAKE or o.type = #HELP));
        situations : Sequence(REUBI!ContextSituation) =
            contributions->collect(c | c.influencedBy);
        contextParameters : Sequence(REUBI!ContextAttribute) =
            situations->collect(c | c.attributes)->flatten()->asSet()->asSequence();
        contextData : Sequence(REUBI!ContextData) =
            contextParameters->collect(c | c.data)->asSet()->asSequence();

        input : Sequence(REUBI!Resource) = goal.demands;
        output : REUBI!Resource = goal.provides;
    }
    to
        method : SCUBI!Method(
            name <- 'perform' + goal.name,
            parameters <- input->collect(i | thisModule.resolveTemp(i, 'parameter')),
            return <- thisModule.resolveTemp(output, 'parameter')
        ),
        interface : SCUBI!FunctionalInterface(
            name <- 'I' + goal.name,
            methods <- Sequence{method}
        ),
        manager : SCUBI!ManagerComponent(
            name <- goal.name + 'Manager',
            provides <- interface,
            demands <- interface,
            manages <- contributions->collect(c | thisModule.resolveTemp(c.source, 'component')),
            controlledBy <- controlInterface
        ),
        stringClass : SCUBI!Class(
            name <- 'String'
        ),
        controlInput : SCUBI!Parameter(
            class <- stringClass,
            name <- 'componentID'
        ),
        controlMethod : SCUBI!Method(
            name <- 'setComponentWithID',
            parameters <- controlInput
        ),
        controlInterface : SCUBI!ControlInterface(
            name <- 'I' + goal.name + 'Control',
            methods <- controlMethod
        ),
        controlComponent : SCUBI!ControlComponent(
            name <- goal.name + 'Controller',
            provides <- controlInterface,
            demands <- controlInterface
        ),
        adaptationComponent : SCUBI!AdaptationComponent(
            name <- goal.name + 'Adapter',
            demands <- controlInterface,
            provides <- contextData->collect(c | thisModule.resolveTemp(c, 'consumerInterface')),
            rules <- contributions->collect(c | thisModule.resolveTemp(c, 'adaptationRule'))
        )
}
```

Figura B.6: Regla de transformación `Goal2FunctionalInterface` en ATL.

```
rule Group2ManagerComponent {
    from
        group : REUBI!Group
    using{
        contributions : Sequence(REUBI!Contribution) =
            group.contributesTo->select(o | (o.type = #MAKE or o.type = #HELP));
        goals : Sequence(REUBI!Goal) = contributions->collect(c | c.target)->asSet();
        operationalizations : Sequence(REUBI!Operationalization) = group.contains;
    }
    to
        component : SCUBI!ManagerComponent(
            name <- group.name + 'CombinatorComponent',
            provides <- goals->collect(g | thisModule.resolveTemp(g, 'interface'))->asSet(),
            demands <- goals->collect(g | thisModule.resolveTemp(g, 'interface'))->asSet(),
            manages <- operationalizations->collect(o | thisModule.resolveTemp(o, 'component'))
        )
}
```

Figura B.7: Regla de transformación `Group2ManagerComponent` en ATL.

```
rule Operationalization2FunctionalComponent {
    from
        oper : REUBI!Operationalization(not oper.oclIsKindOf(REUBI!Group))
    using{
        contributions : Sequence(REUBI!Contribution) = oper.contributesTo->
                select(o | (o.type = #MAKE or o.type = #HELP));
        goals : Sequence(REUBI!Goal) = contributions->collect(c | c.target)->asSet();
        groups : Sequence(REUBI!Group) = oper.containedBy;
    }
    to
        component : SCUBI!FunctionalComponent(
            name <- oper.name + 'Component',
            provides <- Set{goals->collect(g | thisModule.resolveTemp(g, 'interface'))->asSet(),
                            groups->collect(g | thisModule.resolveTemp(g, 'component').provides)
                            ->flatten()->asSet()}->flatten()->asSet()
        )
}
```

Figura B.8: Regla de transformación `Operationalization2FunctionalComponent` en ATL.

```
rule Resource2Class {
    from
        res : REUBI!Resource
    using{
        fields : Sequence(REUBI!Field) = res.attributes;
    }
    to
        cls : SCUBI!Class(
            name <- res.name,
            attributes <- fields->collect(f | thisModule.resolveTemp(f, 'field'))
        ),
        parameter : SCUBI!Parameter(
            name <- 'a' + res.name,
            class <- cls
        )
}
```

Figura B.9: Regla de transformación `Resource2Class` en ATL.

```
rule Field2Field {
    from
        attr : REUBI!Field(not attr.type.oclIsUndefined())
    to
        field : SCUBI!Field(
            name <- attr.name,
            class <- thisModule.resolveTemp(attr.type, 'cls')
        )
}
```

Figura B.10: Regla de transformación `Field2Field` en ATL.

```
rule PlainField2Field {
    from
        attr : REUBI!Field(attr.type.oclIsUndefined())
    to
        cls : SCUBI!Class(
            name <- attr.plaintype
        ),
        field : SCUBI!Field(
            name <- attr.name,
            class <- cls
        )
}
```

Figura B.11: Regla de transformación `PlainField2Field` en ATL.

```
rule ContextData2ContextConsumer {
    from
        data : REUBI!ContextData
    to
        eventClass : SCUBI!Class(
            name <- data.name + 'Event'
        ),
        eventParameter : SCUBI!Parameter(
            class <- eventClass,
            name <- 'a' + data.name
        ),
        consumerMethod : SCUBI!Method(
            name <- 'on' + data.name + 'Received',
            parameters <- eventParameter
        ),
        consumerInterface : SCUBI!ContextInterface(
            name <- 'I' + data.name,
            methods <- consumerMethod
        ),
        consumer : SCUBI!ContextConsumer(
            name <- data.name + 'Consumer',
            provides <- consumerInterface,
            demands <- consumerInterface
        )
}
```

Figura B.12: Regla de transformación `ContextData2ContextConsumer` en ATL.

```
rule ContextAttribute2Condition {
    from
        attribute : REUBI!ContextAttribute
    using{
        data : REUBI!ContextData = attribute.data;
    }
    to
        condition : SCUBI!Condition(
            name <- 'Condition' + thisModule.conditionNo,
            class <- thisModule.resolveTemp(data, 'eventClass'),
            comparison <- thisModule.getComparison(attribute.comparison),
            value <- attribute.value
        )
    do{
        thisModule.conditionNo <- thisModule.conditionNo + 1;

    }
}
```

Figura B.13: Regla de transformación `ContextAttribute2Condition` en ATL.

```
rule Contribution2AdaptationRule {
    from
        contribution : REUBI!Contribution(not contribution.influencedBy->oclIsUndefined())
    using{
        situation : REUBI!ContextSituation = contribution.influencedBy;
        attributes : Sequence(REUBI!ContextAttribute) = situation.attributes;
        data : Sequence(REUBI!ContextData) = attributes->collect(a | a.data)->asSet()->asSequence();
        operationalization : REUBI!Operationalization = contribution.source;
    }
    to
        adaptationRule : SCUBI!Rule(
            name <- 'Rule' + thisModule.ruleNo + '-' + situation.name,
            events <- data->collect(d | thisModule.resolveTemp(d, 'eventClass'))->asSet()->asSequence(),
            conditions <- attributes->collect(a | thisModule.resolveTemp(a, 'condition')),
            actions <- adaptationAction
        ),
        adaptationAction : SCUBI!Action(
            name <- 'Action' + thisModule.ruleNo,
            component <- thisModule.resolveTemp(operationalization, 'component')
        )
    do{
        thisModule.ruleNo <- thisModule.ruleNo + 1;
    }
}
```

Figura B.14: Regla de transformación `Contribution2AdaptationRule` en ATL.

ÍNDICE DE FIGURAS

BIBLIOGRAFÍA

[AAH+97] Gregory Abowd, Christopher Atkeson, Jason Hong, Sue Long, Rob Kooper, and Mike Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3:421–433, 1997. (Citado en la página 241.)

[AC03] Mehmet Aksit and Zièd Choukair. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 84–89. IEEE, 2003. (Citado en la página 117.)

[AFN] O. Ardaiz, F. Freitag, and L. Navarro. On service deployment in ubiquitous computing. In *WORKSHOP ON UBIQUITOUS COMPUTING AND COMMUNICATIONS, PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT*, volume 1. Citeseer. (Citado en las páginas 28 y 44.)

[AGKO04] Lauri Aalto, Nicklas Gothlin, Jani Korhonen, and Timo Ojala. Bluetooth and wap push based location-aware mobile advertising system. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 49–58, 2004. (Citado en la página 241.)

[AP98] A. I. Antón and C. Potts. The use of goals to surface requirements for evolving systems. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 157–166. IEEE Computer Society, 1998. (Citado en las páginas 62, 63, 87 y 315.)

[APT+09] Sofia Aparicio, Javier Perez, Paula Tarrio, Ana Bernardos, and Jose Casar. An indoor location method based on a fusion map using bluetooth and wlan technologies. In *International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008)*, volume 50 of *Advances in Soft*

Computing, pages 702–710. 2009. (Citado en la página 241.)

[Ars04] Ali Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 2004. (Citado en las páginas 31, 46, 107 y 109.)

[AYL11] Daniel Avrahami, Michael Yeganyan, and Anthony LaMarca. The danger of loose objects in the car: challenges and opportunities for ubiquitous computing. In *Proceedings of the 13th international conference on Ubiquitous computing*, pages 173–176. ACM, 2011. (Citado en las páginas 27 y 42.)

[BBB+05] Michael Beisiegel, Henning Blohm, Dave Booz, J Dubray, Adrian Colyer, Mike Edwards, Don Ferguson, Bill Flood, Mike Greenberg, Dan Kearns, et al. Service component architecture. building systems using a service oriented architecture. *Whitepaper [online]*, pages 1–31, 2005. (Citado en las páginas 31, 46, 108 y 109.)

[BCZ05] D. M. Berry, B. H. C. Cheng, and J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *In 11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ)*, 2005. (Citado en las páginas 60 y 82.)

[BDR07] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007. (Citado en la página 142.)

[Bec03] R. Beckwith. Designing for ubiquity: The perception of privacy. *IEEE pervasive computing*, pages 40–46, 2003. (Citado en las páginas 245 y 255.)

[BH01] A.F. Blackwell and R. Hague. Autohan: An architecture for programming the home. 2001.

[BLP05] T. Buchholz and C. Linnhoff-Popien. Towards realizing global scalability in context-aware systems. *Location-and Context-Awareness*, pages 15–17, 2005.

[BMC+09] Ruben Blasco, Alvaro Marco, Roberto Casas, Alejandro Ibarz, Victoria Coarasa, and Angel Asensio. Indoor localization based on neural networks

for non-dedicated zigbee networks in aal. In *Bio-Inspired Systems: Computational and Ambient Intelligence*, volume 5517 of *Lecture Notes in Computer Science*, pages 1113–1120. 2009. (Citado en la página 241.)

[BPS10] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy goals for requirements-driven adaptation. *Requirements Engineering, IEEE International Conference on*, 0:125–134, 2010. (Citado en las páginas 66, 68, 87, 89, 90 y 315.)

[BS93] V. Bellotti and A. Sellen. Design for privacy in ubiquitous computing environments. In *Proceedings of the third conference on European Conference on Computer-Supported Cooperative Work*, pages 77–92. Kluwer Academic Publishers, 1993. (Citado en la página 255.)

[CCAT12] Sophie Chabridon, Denis Conan, Zied Abid, and Chantal Taconet. Building ubiquitous qoc-aware applications through model-driven software engineering. *Science of Computer Programming*, 2012. (Citado en las páginas 31, 46 y 100.)

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17, 2003. (Citado en la página 96.)

[CJJA04] Hae Don Chon, Sibum Jun, Heejae Jung, and Sang Won An. Using rfid for accurate positioning. *Journal of Global Positioning Systems*, 2004. (Citado en las páginas 237 y 241.)

[CNYM00] L. Chung, B.A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000. (Citado en las páginas 27, 30, 43, 45, 73, 74, 88, 89, 147, 167 y 315.)

[Coa12] Workflow Management Coalition. Xml process definition language (xpdl). http://www.xpdl.org, 2012. (Citado en las páginas 104 y 105.)

[Con01] World Wide Web Consortium. Web services description language (wsdl). http://www.w3.org/tr/wsdl, 2001. (Citado en la página 110.)

[Con04a] World Wide Web Consortium. Owl-s: Semantic markup for web services, 2004. (Citado en la página 110.)

[Con04b] World Wide Web Consortium. Web ontology language (owl) http://www.w3.org/tr/owl-features/, 2004. (Citado en la página 104.)

[Con07a] UWA Consortium. Ubiquitous web applications, 2007. (Citado en las páginas 31, 46, 85 y 87.)

[Con07b] World Wide Web Consortium. Simple object access protocol (soap). http://www.w3.org/tr/soap/, 2007. (Citado en la página 110.)

[CSBW09] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 468–483. 2009. (Citado en las páginas 68, 69, 87, 89, 90 y 315.)

[DBS+02] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, JC Burgelman, et al. Reading list for context-aware personalization. *Communications of the ACM*, 45(2):102–104, 2002. (Citado en las páginas 28 y 44.)

[DCCC05] Jim Dowling, Eoin Curran, Raymond Cunningham, and Vinny Cahill. Using feedback in collaborative reinforcement learning to adaptively optimize manet routing. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 35(3):360–372, 2005. (Citado en la página 115.)

[DDMvL97] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: an environment for goal-driven requirements engineering. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97. ACM, 1997. (Citado en las páginas 31, 46, 63, 64, 87 y 315.)

[Dey01] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001. (Citado en la página 111.)

[DMS98] C Drane, M Macnaughtan, and C Scott. Positioning gsm telephones. *IEEE Communications Magazine*, 36(4):46–54, 59, 1998. (Citado en la página 235.)

[dOdPdSB09] Raphael Pereira de Oliveira, Antonio Francisco do Prado, Wanderley Lopes de Souza, and Mauro Biajiz. Development based on mda, of ubiquitous applications domain product lines. In *Computer and Information Science, 2009. ICIS 2009. Eighth IEEE/ACIS International Conference on*, pages 1005–1010. IEEE, 2009. (Citado en las páginas 31, 46 y 100.)

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20:3–50, 1993. (Citado en la página 63.)

[EAK06] Abdelkarim Erradi, Sriram Anand, and Naveen Kulkarni. Soaf: An architectural framework for service definition and realization. In *Services Computing, 2006. SCC'06. IEEE International Conference on*, pages 151–158. IEEE, 2006. (Citado en las páginas 107 y 109.)

[Erl08] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008. (Citado en las páginas 30, 45, 106, 108 y 109.)

[EWA06] Christian Emig, Jochen Weisser, and Sebastian Abeck. Development of soa-based software systems-an evolutionary programming approach. In *Telecommunications, 2006. AICT-ICIW'06. International Conference on Internet and Web Applications and Services/Advanced International Conference on*, pages 182–182. IEEE, 2006. (Citado en las páginas 108 y 109.)

[Fer06] A. Ferscha. Context aware systems. In *Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*, pages 1–2. ACM, 2006. (Citado en las páginas 28, 43 y 234.)

[FPQ+10] Xavier Franch, Cristina Palomares, Carme Quer, Samuel Renault, and François De Lazzer. A metamodel for software requirement patterns. In *Requirements Engineering: Foundation for Software Quality*, pages 85–90. Springer, 2010. (Citado en la página 86.)

[Fra13] Xavier Franch. Software requirement patterns. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1499–1501. IEEE, 2013. (Citado en la página 86.)

[GCH+04] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004. (Citado en la página 114.)

[Gei] J. Geier. Multipath, a potential wlan problem. http://www.wifiplanet.com/tutorials/article.php/1121691. (Citado en la página 237.)

[GFDT05] K. Gratsias, E. Frentzos, V. Delis, and Y. Theodoridis. Towards a taxonomy of location based services. *Web and Wireless Geographical Information Systems*, pages 19–30, 2005. (Citado en la página 233.)

[Gli08] Martin Glinz. On non-functional requirements. *Requirements Engineering, IEEE International Conference on*, 0:21–26, 2008. (Citado en las páginas 59, 60 y 315.)

[GMFFGS10] Iván García-Magariño, Rubén Fuentes-Fernández, and Jorge J Gómez-Sanz. A framework for the definition of metamodels for computer-aided software engineering tools. *Information and Software Technology*, 52(4):422–435, 2010. (Citado en la página 105.)

[Gro03a] Object Management Group. Common warehouse metamodel (cwm). http://www.omg.org/spec/cwm/, 2003. (Citado en la página 95.)

[Gro03b] Object Management Group. Model driven architecture (mda). http://www.omg.org/mda/, 2003. (Citado en las páginas 30, 45, 95 y 99.)

[Gro05a] Object Management Group. Meta object facility (mof). http://www.omg.org/mof/, 2005. (Citado en las páginas 95 y 101.)

[Gro05b] Object Management Group. Unified modeling language (uml). http://www.uml.org/, 2005. (Citado en la página 95.)

[Gro06] Object Management Group. Acceleo. www.acceleo.org, 2006. (Citado en la página 99.)

[Gro08a] Object Management Group. Mof model to text. http://www.omg.org/spec/mofm2t/, 2008. (Citado en las páginas 98 y 99.)

[Gro08b] Object Management Group. Software and systems process engineering metamodel (spem). http://www.omg.org/spec/spem/, 2008. (Citado en las páginas 101 y 105.)

[Gro11a] Object Management Group. Object constraint language (ocl). http://www.omg.org/spec/ocl, 2011. (Citado en la página 100.)

[Gro11b] Object Management Group. Query-view-transformation (qvt). http://www.omg.org/spec/qvt/, 2011. (Citado en la página 97.)

[Gro11c] Object Management Group. Xml metadata interchange (xmi). http://www.omg.org/spec/xmi/, 2011. (Citado en la página 95.)

[Gro13] Object Management Group. Business process model and notation (bpmn). http://www.omg.org/spec/bpmn, 2013. (Citado en la página 101.)

[GSB+08] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H.C. Cheng, and D. Hughes. Goal-based modeling of dynamically adaptive system requirements. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:36–45, 2008. (Citado en las páginas 31, 46, 82, 87 y 90.)

[GSB+13] Avik Ghose, Priyanka Sinha, Chirabrata Bhaumik, Aniruddha Sinha, Amit Agrawal, and Anirban

Dutta Choudhury. Ubiheld: ubiquitous healthcare monitoring system for elderly and chronic patients. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, pages 1255–1264. ACM, 2013. (Citado en las páginas 27 y 43.)

[Her10] Heinrich Herre. General formal ontology (gfo): A foundational ontology for conceptual modelling. In *Theory and Applications of Ontology: Computer Applications*, pages 297–345. Springer, 2010. (Citado en las páginas 104 y 105.)

[HF04] W. Heaven and A. Finkelstein. Uml profile to support requirements engineering with kaos. *IEEE Proceedings - Software*, 151:10–27, 2004. (Citado en las páginas 64, 65 y 315.)

[HMOG08] A. Haller, M. Marmolowski, E. Oren, and W. Gaaloul. A process ontology for business intelligence. *Digital Enterprise Research Institute (DERI)*, pages 1 – 16, 2008. (Citado en las páginas 104 y 105.)

[HOK06] Armin Haller, Eyal Oren, and Paavo Kotinurmi. m3po: An ontology to relate choreographies to workflow models. In *Services Computing, 2006. SCC'06. IEEE International Conference on*, pages 19–27. IEEE, 2006. (Citado en las páginas 104 y 105.)

[INR08] INRIA. Atlas transformation language. http://www.eclipse.org/atl/, 2008. (Citado en la página 98.)

[(IS07a] Information Society Technologies (IST). The service oriented process. *CBDi Journal*, 2007. (Citado en las páginas 107 y 109.)

[(IS07b] Information Society Technologies (IST). Super business process ontology method, 2007. (Citado en las páginas 104 y 105.)

[JB04] J.B. Jrgensen and C. Bossen. Executable use cases: Requirements for a pervasive health care system. *IEEE Softw.*, 21:34–41, 2004. (Citado en las páginas 31, 46, 84 y 87.)

[JKC04] H.W. Jung, S.G. Kim, and C.S. Chung. Measuring software product quality: A survey of iso/iec 9126. *Software, IEEE*, 21(5):88–92, 2004.

[JL02] X. Jiang and J.A. Landay. Modeling privacy control in context-aware systems. *Pervasive Computing, IEEE*, 1(3):59–63, 2002. (Citado en la página 256.)

[Jon05] Steve Jones. A methodology for service architectures, 2005. (Citado en las páginas 108 y 109.)

[Kae] K. Kaemarungsi. Design of indoor positioning systems based on location fingerprinting technique. (Citado en la página 240.)

[Kaw09] Nobuo Kawaguchi. Wifi location information system for both indoors and outdoors. In *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, volume 5518 of *Lecture Notes in Computer Science*, pages 638–645. 2009. (Citado en la página 241.)

[KBM+02] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: web presence for the real world. *Mob. Netw. Appl.*, 7:365–376, October 2002.

[KD07] Jeffrey O Kephart and Rajarshi Das. Achieving self-management via utility functions. *Internet Computing, IEEE*, 11(1):40–48, 2007. (Citado en la página 113.)

[KEW06] L. Kolos-Mazuryk, P. Eck, and R. Wieringa. A survey of requirements engineering methods for pervasive services, 2006.

[Kja07] Mikkel Kjaergaard. A taxonomy for radio location fingerprinting. *Lecture Notes in Computer Science*, 4718:139–156, 2007. (Citado en la página 235.)

[KKFS10] Gerd Kortuem, Fahim Kawsar, Daniel Fitton, and Vasughi Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010. (Citado en las páginas 27 y 42.)

[KKJ+09] N. Koay, P. Kataria, R. Juric, P. Oberndorf, and G. Terstyanszky. Ontological support for managing non-functional requirements in pervasive healthcare. *Hawaii International Conference on System Sciences*, 0:1–10, 2009.

[KKP03] SH Kim, S.W. Kim, and HM Park. Usability challenges in ubicomp environment. *the Proceeding of International Ergonomics Association (IEA'03)(Seoul, Korea*, 2003.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. (Citado en las páginas 77, 78 y 315.)

[Kolo5] L. Kolos-Mazuryk. Development of a requirements engineering method for pervasive services. University of Waterloo, School of Computer Science, 2005. (Citado en las páginas 31, 46 y 87.)

[KPP10] Konstantinos Kakousis, Nearchos Paspallis, and George Angelos Papadopoulos. A survey of software adaptation in mobile and ubiquitous computing. *Enterprise Information Systems*, 4(4):355–389, 2010. (Citado en las páginas 30, 45 y 111.)

[KPv05] L. Kolos-Mazuryk, G.J. Poulisse, and P.A.T. van Eck. Requirements engineering for pervasive services. In *Second Workshop on Building Software for Pervasive Computing. Position Papers.*, pages 18–22, 2005.

[KR12] James F Kurose and Keith W Ross. *Computer networking*. Pearson Education, 2012. (Citado en la página 116.)

[KW04] Jeffrey O Kephart and William E Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12. IEEE, 2004. (Citado en la página 114.)

[LA06] C. López and H. Astudillo. Explicit architectural policies to satisfy nfrs using cots. In *Satellite Events at the MoDELS 2005 Conference*, pages 227–236. Springer, 2006. (Citado en la página 137.)

[Lano2] M. Langheinrich. A privacy awareness system for ubiquitous computing environments. *UbiComp*

2002: Ubiquitous Computing, pages 315–320, 2002. (Citado en la página 245.)

[LB03] James A Landay and Gaetano Borriello. Design patterns for ubiquitous computing. *Computer*, 36(8):93–95, 2003. (Citado en la página 86.)

[LBS09] B. Lee, R. Ballagas, and M. Stone. Designing for latency: In search of the balance between system flexibility and responsiveness. 2009.

[LDBL07] H. Liu, H. Darabi, P. Banerjee, and J. Liu. Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(6):1067–1080, 2007. (Citado en las páginas 233, 237, 244, 248 y 252.)

[Lea96] David B Leake. *Case-based reasoning: Experiences, lessons and future directions*. MIT press, 1996. (Citado en la página 115.)

[LL05] T.N. Lin and P.C. Lin. Performance comparison of indoor positioning techniques based on location fingerprinting in wireless networks. In *Wireless Networks, Communications and Mobile Computing, 2005 International Conference on*, volume 2, pages 1569–1574. IEEE, 2005. (Citado en las páginas 234 y 245.)

[Mar99] Natalia Marmasse. commotion: a context-aware communication system. In *CHI '99 extended abstracts on Human factors in computing systems*, CHI EA '99, pages 320–321, 1999. (Citado en la página 241.)

[McC93] Steve McConnell. Code complete: a practical handbook of software construction (redmond, wa, 1993. (Citado en la página 86.)

[MCY99] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Commun. ACM*, 42:31–37, 1999. (Citado en las páginas 27, 28, 43, 73 y 88.)

[MDMBG11] Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. The proximity toolkit: Prototyping proxemic interactions in ubiquitous

computing ecologies. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 315–326, New York, NY, USA, 2011. ACM. (Citado en las páginas 26 y 42.)

[Mic07] SUN Microsystems. Soa rq methodology - a pragmatic approach, 2007. (Citado en las páginas 107 y 109.)

[Mit] K Mittal. Service oriented unified process, soup (2006). (Citado en las páginas 107 y 109.)

[MSKC04] Philip K McKinley, Seyed Masoud Sadjadi, Eric P Kasten, and Betty HC Cheng. A taxonomy of compositional adaptation. *Rapport Technique numéroMSU-CSE-04-17, juillet*, 2004. (Citado en la página 117.)

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46. ACM, 2000. (Citado en la página 58.)

[NM99] C. Ncube and N. A.M. Maiden. Pore: Procurement-oriented requirements engineering method for the component-based systems engineering development paradigm. *International Workshop on Component-Based Software Engineering*, 1999. (Citado en las páginas 75 y 87.)

[NS99] Brian D Noble and Mahadev Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999. (Citado en la página 114.)

[OAS02] OASIS. Universal description, discovery and integration (uddi). https://www.oasis-open.org/committees/uddi-spec/faq.php, 2002. (Citado en la página 110.)

[OAS06] OASIS. Service oriented architecture. http://www.oasis-open.org/committees/soa-rm/, 2006. (Citado en la página 76.)

[OFR+12] Avner Ottensooser, Alan Fekete, Hajo A Reijers, Jan Mendling, and Con Menictas. Making sense

of business process descriptions: An experimental comparison of graphical and textual notations. *Journal of Systems and Software*, 85(3):596–606, 2012. (Citado en la página 104.)

[OGA+06] Shumao Ou, Nektarios Georgalas, Manooch Azmoodeh, Kun Yang, and Xiantang Sun. A model driven integration architecture for ontology-based context modelling and context-aware application development. In *Model Driven Architecture–Foundations and Applications*, pages 188–197. Springer, 2006. (Citado en la página 100.)

[Onto4] Web Services Modeling Ontology. Web services modeling language (wsml). http://www.wsmo.org, 2004. (Citado en la página 104.)

[oTFIG] Massachussets Institute of Technology. Fluid Interfaces Group. http://ambient.media.mit.edu/.

[PB05] Davy Preuveneers and Yolande Berbers. Semantic and syntactic modeling of component-based services for context-aware pervasive systems using owl-s. In *First International Workshop on Managing Context Information in Mobile and Pervasive Environments*, pages 30–39. Citeseer, 2005. (Citado en las páginas 31, 46, 108 y 109.)

[PCN06] Sara Passone, Paul WH Chung, and Vahid Nassehi. Incorporating domain-specific knowledge into a genetic algorithm to implement case-based reasoning adaptation. *Knowledge-Based Systems*, 19(3):192–201, 2006. (Citado en la página 115.)

[PJKF02] S. Ponnekanti, B. Johanson, E. Kıcıman, and A. Fox. Designing for maintainability, failure resilience, and evolvability in ubiquitous computing software. *Submission to Operating Systems Design and Implementation*, 2002.

[PJKF03] S.R. Ponnekanti, B. Johanson, E. Kiciman, and A. Fox. Portability, extensibility and robustness in iros. In *Pervasive Computing and Communications, 2003.(PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 11–19. IEEE, 2003.

[PKC01] P. Prasithsangaree, P. Krishnamurthy, and P. K. Chrysanthis. On indoor position location with wireless lans. In *13th IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC 2002)*, pages 720–724, 2001. (Citado en las páginas 233, 240 y 241.)

[PLM02] K. Pahlavan, X. Li, and J.P. Makela. Indoor geolocation science and technology. *IEEE Communications Magazine*, pages 112–118, 2002. (Citado en la página 236.)

[Pot99] C. Potts. Scenic: A strategy for inquiry-driven requirements determination. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering*, pages 58–65. IEEE Computer Society, 1999. (Citado en las páginas 71 y 87.)

[PTA94] C. Potts, K. Takahashi, and A. I. Antón. Inquiry-based requirements analysis. *IEEE Softw.*, 11:21–32, 1994. (Citado en la página 71.)

[PVDH06] Michael P Papazoglou and Willem-Jan Van Den Heuvel. Service-oriented design and development methodology. *International Journal of Web Engineering and Technology*, 2(4):412–442, 2006. (Citado en las páginas 31, 46, 108 y 109.)

[RBE08] Romain Rouvoy, Mikaël Beauvois, and Frank Eliassen. Dynamic aspect weaving using a planning-based adaptation middleware. In *Proceedings of the 2nd workshop on Middleware-application interaction: affiliated with the DisCoTec federated conferences 2008*, pages 31–36. ACM, 2008. (Citado en la página 117.)

[RDFRR06] Oriana Riva, Cristiano Di Flora, Stefano Russo, and Kimmo Raatikainen. Unearthing design patterns to support context-awareness. In *Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on*, pages 5–pp. IEEE, 2006. (Citado en la página 86.)

[RDM03] Cliff Randell, Chris Djiallis, and Henk Muller. Personal position measurement using dead reckoning. *Wearable Computers, IEEE International Symposium*, 0:166, 2003. (Citado en la página 239.)

[RDS07] Ervin Ramollari, Dimitris Dranidis, and Anthony JH Simons. A survey of service oriented development methodologies. In *The 2nd European Young Researchers Workshop on Service Oriented Computing*, page 75, 2007. (Citado en la página 107.)

[RGL05] Gustavo Rossi, Silvia Gordillo, and Fernando Lyardet. Design patterns for context-aware adaptation. In *Applications and the Internet Workshops, 2005. Saint Workshops 2005. The 2005 Symposium on*, pages 170–173. IEEE, 2005. (Citado en la página 86.)

[RLGBC10] Tomas Ruiz-Lopez, Jose Luis Garrido, Kawtar Benghazi, and Lawrence Chung. A survey on indoor positioning systems: Foreseeing a quality design. In *Distributed Computing and Artificial Intelligence*, volume 79 of *Advances in Intelligent and Soft Computing*, pages 373–380. 2010. (Citado en las páginas 28, 44, 234 y 254.)

[RSMA02] A. Rashid, P. Sawyer, A. Moreira, and J. Araújo. Early aspects: A model for aspect-oriented requirements engineering. *Requirements Engineering, IEEE International Conference on*, 0:199, 2002. (Citado en las páginas 77 y 88.)

[Sat01] M. Satyanarayanan. Pervasive computing: Vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001.

[Sch99] August-Wilhelm Scheer. Aris-business process frameworks. *STUDIES IN INFORMATICS AND CONTROL*, 8:251–252, 1999. (Citado en las páginas 103, 104 y 105.)

[SFS05] A. Sutcliffe, S. Fickas, and M. M. Sohlberg. Personal and contextual requirements engineering. *Requirements Engineering, IEEE International Conference on*, 0:19–30, 2005. (Citado en las páginas 78, 79, 87, 90 y 315.)

[SHC+10] Sam Supakkul, Tom Hill, Lawrence Chung, Thein Than Tun, and JC Sampaio do Prado Leite. An nfr pattern approach to dealing with nfrs. In *Requirements Engineering Conference (RE), 2010*

18th IEEE International, pages 179–188. IEEE, 2010. (Citado en las páginas 29, 45, 86 y 166.)

[SIK98] Craig Schlenoff, Rob Ivester, and Amy Knutilla. A robust process ontology for manufacturing systems integration. In *Proceedings of 2nd International Conference on Engineering Design and Automation*, pages 7–14, 1998. (Citado en las páginas 103, 104 y 105.)

[SJ99] T.A. Stansell Jr. Mitigation of multipath effects in global positioning system receivers, 1999. US Patent 5,963,582. (Citado en la página 245.)

[SLK06] U. Sandner, J.M. Leimeister, and H. Krcmar. Business potentials of ubiquitous computing. *Managing Development and Application of Digital Technologies*, pages 277–291, 2006. (Citado en las páginas 29 y 44.)

[SM98] A. G. Sutcliffe and N.A.M. Maiden. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24:1072 – 1088, 1998. (Citado en la página 69.)

[SS07] W. Sitou and B. Spanfelner. Towards requirements engineering for context adaptive systems. *Computer Software and Applications Conference, Annual International*, 2:593–600, 2007. (Citado en las páginas 31, 46, 81, 82, 87, 90 y 315.)

[Sut03] A. Sutcliffe. Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 320. IEEE Computer Society, 2003. (Citado en las páginas 70, 71, 87 y 315.)

[SV04] J. Schiller and A. Voisard. *Location-based Services*. Morgan Kaufmann, 2004. (Citado en las páginas 234 y 242.)

[SVP10] Estefanía Serral, Pedro Valderas, and Vicente Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2):254–280, 2010. (Citado en las páginas 31, 46 y 100.)

[TJWW07] W. T. Tsai, Z. Jin, P. Wang, and B. Wu. Requirement engineering in service-oriented system engineering. *E-Business Engineering, IEEE International Conference on*, 0:661–668, 2007. (Citado en las páginas 76 y 87.)

[TNA09] A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos. Handling non-functional requirements in information system architecture design. *Software Engineering Advances, International Conference on*, 0:59–64, 2009. (Citado en la página 88.)

[UI00] B. Ullmer and H. Ishii. Emerging frameworks for tangible user interfaces. *IBM systems journal*, 39(3.4):915–931, 2000.

[VH08] Samyr Vale and Slimane Hammoudi. Context-aware model driven development by parameterized transformation. *Proceedings of MDISIS*, 2008. (Citado en la página 100.)

[Wan96] Pei Wang. The interpretation of fuzziness. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(2):321–326, 1996. (Citado en la página 113.)

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991. (Citado en las páginas 25 y 40.)

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993. (Citado en las páginas 24 y 40.)

[WSB+09] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, and J.M. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. *Requirements Engineering, IEEE International Conference on*, 0:79–88, 2009. (Citado en la página 68.)

[XZZG08] K. Xu, M. Zhu, D. Zhang, and T. Gu. Context-aware content filtering & presentation for pervasive & mobile information systems. In *Proceedings of the 1st international conference on Ambient media and systems*, page 20. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008. (Citado en la página 256.)

[YM98] E.S.K. Yu and J. Mylopoulos. Why goal-oriented requirements engineering. 1998. (Citado en la página 61.)

[Yu97] E. S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. *Requirements Engineering, IEEE International Conference on*, 0:226, 1997. (Citado en las páginas 64, 66, 87, 88, 89 y 315.)

[Zav97] P. Zave. Classification of research efforts in requirements engineering. *ACM Comput. Surv.*, 29:315–321, December 1997. (Citado en la página 58.)

[ZGL02] V. Zeimpekis, G.M. Giaglis, and G. Lekakos. A taxonomy of indoor and outdoor positioning techniques for mobile location services. *ACM SIGecom Exchanges*, 3(4):19–27, 2002. (Citado en la página 236.)

[ZKG04] Olaf Zimmermann, Pal Krogdahl, and Clive Gee. Elements of service-oriented analysis and design. *IBM developerworks*, 2004. (Citado en las páginas 107 y 109.)

[ZME03] Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich. Adaptable mobile applications: Exploiting logical mobility in mobile computing. In *Mobile agents for telecommunication applications*, pages 170–179. Springer, 2003. (Citado en la página 116.)