Tesis Fin de Máster

Aprendiendo del feedback del usuario para mejorar el diseño de la adaptación del nivel de molestia de las interacciones

Juan Vicente Berzosa Tejero

2013

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Director de tesis: Vicente Pelechano y Miriam Gil

Juan Vicente Berzosa Tejero

Aprendiendo del feedback del usuario para mejorar el diseño de la adaptación del nivel de molestia de las interacciones.

Tesis de Máster, septiembre de 2013

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Aprendiendo del *feedback* del usuario para mejorar el diseño de la adaptación del nivel de molestia de las interacciones

Tesis de máster de:

Juan Vicente Berzosa Tejero

Directores de máster:

Vicente Pelechano Ferragud
Miriam Gil Pascual

Miembros del tribunal de evaluación, especialidad de Sistemas de Información:

Vicente Pelechano Ferragud
Matilde Celma Giménez
César Ferri Ramírez

Centro de Investigación en Métodos de Producción de Software ProS
Universitat Politècnica de València
Camino de Vera s/n, Edif. 1F
46022 - Valencia, Spain
Tel: (+34) 963 877 007 (Ext. 83533)
Fax: (+34) 963 877 359
Web: http://www.pros.upv.es

A mi familia.

Agradecimientos

Esta tesis está dedicada, por encima de todo, a mi familia. Su significado, como culmen de mi etapa de estudios superiores, supone el fin de una época de considerables esfuerzos, compartidos por ellos y por mí. De igual forma, espero que ahora todos podamos compartir sus frutos.

También me gustaría agradecer a todas aquellas personas que, de una manera u otra, me han ayudado a desarrollar y llevar a buen término esta tesis:

Agradecer a mis directores de tesis, Vicente Pelechano y Miriam Gil, ya que me brindaron la idea que fundamenta esta tesis y me facilitaron el marco adecuado para realizarla, guiándome en su realización, resolviendo mis dudas y suministrándome una apreciada documentación. También a Ignacio Mansanet por facilitarme el entorno de programación utilizado para el desarrollo de la tesis.

Y deseo dar las gracias muy especialmente a Ángel Ruiz, compañero de este mismo máster, que ha resultado ser una de las personas más desinteresadas que conozco, y la más voluntariosa a la hora de resolver problemas. Unido esto a sus grandes conocimientos en informática, han hecho de él una incalculable fuente de ayuda para la realización de esta tesis.

Juan

Resumen

Nos encontramos en un periodo de transición, siendo participes de un viaje desde el paradigma del *ordenador de sobremesa*, con grandes y pesados equipos de escritorio, hacia la *computación ubicua*, donde podemos disfrutar a cualquier hora y lugar de nuestras aplicaciones, mediante pequeños y manejables dispositivos que bien podrían caber en la palma de nuestra mano.

Este viaje trae consigo una serie de cambios en cuanto a como estas aplicaciones interactúan con nosotros. Ya no hablamos de sentarse frente a un PC en la oficina o en nuestra habitación, y leer el correo, navegar por la web o hacer un uso “consciente” de alguna aplicación. Hoy, diariamente, se plantean cientos de interacciones entre dichos dispositivos y los usuarios, como por ejemplo cuando recibimos un mensaje y nuestro móvil nos lo notifica vibrando, sonando o iluminándose, o cuando nuestro lector de noticias nos avisa de que acaba de suceder una noticia que se adapta a nuestras preferencias. Debido al carácter ubicuo de esa tecnología, el conjunto de interacciones ente dispositivos y usuarios ya no se llevará a cabo siempre desde el mismo lugar, en un entorno “controlado” y siempre según un mismo patrón horario. En la actualidad los usuarios están siempre conectados, pudiendo realizarse la interacción en cualquier lugar y en cualquier momento.

Por otra parte, ya no es solo el usuario quien iniciará la interacción, sino que las aplicaciones y los servicios también pueden iniciarla, demandando la atención del usuario. Por lo tanto, ***es necesario definir una regulación de estas peticiones de atención*** por parte de los

servicios, que deben llevarse a cabo de forma que no demanden excesiva atención, molestando exageradamente al usuario.

Existe por lo tanto una problemática y **una clara necesidad de que los servicios interactúen con los usuarios sabiendo adaptarse a sus preferencias en cada momento, de una forma adecuada** (sin molestarles en exceso).

Partiendo de los trabajos de investigación en el ámbito de la personalización de la interacción entre servicios y usuario desarrollados por el *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València*[2]*,* ***en esta tesina se abordará el desarrollo de un sistema de auto-adaptación del nivel de molestia*** (el grado con que cada servicio se inmiscuye en la mente de los usuarios) ***de los servicios basado en un algoritmo de aprendizaje por refuerzo***. El sistema adaptará el nivel de molestia de cada servicio a las necesidades de los usuarios en cada momento, más allá de una configuración inicial en tiempo de diseño, o de unas preferencias seleccionadas por medio de un perfil determinado.

Parafraseando el título de esta tesis, y como detallaremos durante el transcurso de esta documentación, ***el sistema aprenderá del feedback del usuario para mejorar el diseño de la adaptación del nivel de molestia de las interacciones***.

[1] http://www.pros.upv.es
[2] http://www.upv.es

Índice de contenidos

Índice de figuras

Índice de tablas

Índice de listados

"Intelligence is the ability to adapt to change."

- Stephen Hawking (Físico Inglés, 1942)

CAPÍTULO 1

Introducción

El concepto de *computación ubicua*, definido por *Mark Weiser* a finales de la década de los 80 y popularizado en 1991 con su trabajo "*The computer for the 21st century*" [1], implica redefinir la forma en que los usuarios interactúan con los ordenadores: en cualquier momento y en cualquier lugar, de una manera casi transparente. Posiciona al ordenador en un segundo plano y se centra en las personas, en los usuarios, de manera que no sean estos quienes se tienen que adaptar a los ordenadores, sino que los ordenadores son los que deben adaptarse a las necesidades de los usuarios en cada momento. Por tanto, esta emergente tecnología representa un nuevo paradigma de interacción muy alejado del paradigma del *ordenador de sobremesa* (denominada también "*de escritorio*"), prevaleciente hasta hace poco tiempo, donde el usuario era quien debía iniciar la interacción, de una manera activa y plenamente consciente, focalizando completamente su atención.

Con las tecnologías ubicuas, los usuarios interactúan mediante pequeños y manejables dispositivos móviles, como pueden ser los *smartphones*, las *tablets* o los nuevos *smartwatches,* convirtiendo el acceso a los ordenadores en algo sencillo y habitual, accesible para todo tipo de usuarios que utilizan sus servicios interaccionando de modo muchas veces inapreciable, centrándose en sus quehaceres en lugar de en la interacción en sí misma. Ahora, a cualquier dispositivo electrónico, ya sea un coche, una televisión, una nevera, lavadora o

cualquier otro electrodoméstico, se le puede integrar un procesador ARM o similar que le permita interactuar con el usuario, ofreciéndole un valor añadido.

A su vez, este uso creciente de estos dispositivos, provoca un mayor número de aplicaciones y servicios disponibles para ellos, como atestiguan los mercados de aplicaciones *Google Play store* o *App store*, ambos con más de 50 billones de descargas a fecha de agosto del 2013 [2][3], con una tendencia creciente, como se puede observar en la figura 1.

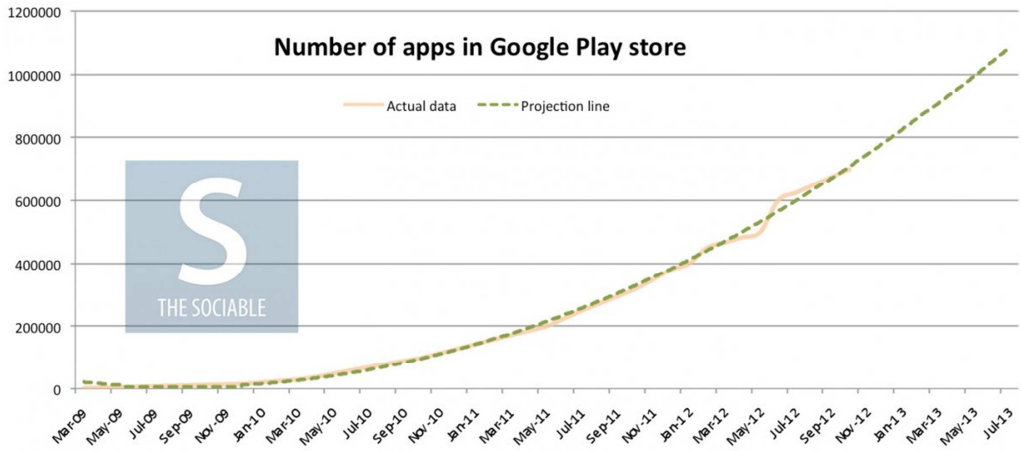

Figura 1 – Curva de crecimiento de aplicaciones en *Google Play store.*

Todos estos servicios interaccionan con los usuarios mediante algún tipo de notificación, como puede ser mediante mensajes visuales, señales acústicas, vibración, etc. Debido al comentado carácter ubicuo, donde el uso de los servicios se realiza desde diferentes entornos, con diferentes características ambientales, de visibilidad y sonoridad, y al citado aumento de servicios (y por tanto de sus notificaciones), hay que prestar especial atención en prevenir que todas estas notificaciones no sobrecarguen a los usuarios, llegando incluso a la molestia [19][20].

Existe por tanto una problemática y una clara necesidad de personalizar la manera en que los servicios se comunican con los

usuarios, de tal modo que sepan adaptarse a sus necesidades en todo momento, mejorando el diseño de la interacción usuario-servicio de las aplicaciones.

Como una solución práctica a esta problemática, ***esta tesina aborda el desarrollo de un sistema basado en un algoritmo de aprendizaje por refuerzo que, en base a la respuesta del usuario frente a una interacción con un servicio*** **(***en adelante* ***feedback*****), adaptará dicha interacción a las necesidades del usuario en cada momento**.

El resto de este capítulo está organizado de la siguiente forma: En el apartado 1.1 se expondrá la motivación de esta tesis. El apartado 1.2 se presentarán sus objetivos. En el apartado 1.3 se extenderá la problemática presentada en la introducción referente a la interacción usuario-servicios, mientras que en el apartado 1.4 se presentará la solución propuesta para dicha problemática. El apartado 1.5 presenta el contexto en que se desenvuelve el trabajo de esta tesis. Finalmente, el apartado 1.6 muestra un resumen de la estructura de esta documentación.

1.1 Motivación

El auge de las tecnologías ubicuas conlleva nuevas formas de acceder a la información, en cualquier momento y desde cualquier lugar, como por ejemplo desde los teléfonos inteligentes, las *tablets* o los *wereable computers (*como las *Google Glass* y los *smartwatches)*, de manera sencilla y cómoda. A diferencia del paradigma de interacción del *ordenador de sobremesa*, donde se presuponía una plena atención por parte de los usuarios que estaban usando sus servicios en un entorno “controlado” (las condiciones ambientales como la iluminación o el ruido eran siempre similares y podían regularse), la

tecnología ubicua supone un cambio en la forma, el tiempo y el lugar en que los usuarios interactúan con los servicios.

Si se añadiese la característica de la ubicuidad pero se continuara con el mismo paradigma de sobremesa, un servicio podría por ejemplo querer interactuar con un usuario que está en medio de una reunión importante de trabajo, interrumpiéndola, o podría por ejemplo querer notificar algo en mitad de la noche, demandando la atención de un usuario que se encuentra descansando. Del mismo modo, podría querer avisar a un usuario de un hecho importante de una forma preestablecida anteriormente, por ejemplo mediante una notificación acústica, aun cuando el usuario puede haber cambiado a un ambiente muy ruidoso donde sería imposible que se percatase de dicha notificación, o bien mediante un aviso por escrito, mientras el usuario está en un ambiente demasiado iluminado como para poder reparar en ello. De ahí la importancia de que ***en el paradigma de la computación ubicua, son los servicios los que se tienen que adaptar a los usuarios***, y no al contrario, teniendo que modificar la forma en que interactúan con los usuarios dependiendo de nuevos factores como en qué lugar, en qué momento y en qué situación va a desarrollarse la interacción, pues ahora las preferencias y las necesidades de los usuarios pueden cambiar con cada combinación de estos factores.

Además, por el comentado auge en la utilización de servicios, es fácil imaginar un escenario donde se puede llegar a saturar a los usuarios a base de notificaciones. Imaginemos por ejemplo un hipotético escenario donde nuestras *Google Glass* nos avisan de una cita importante para hoy mientras nuestro *smartwatch* no cesa de vibrar indicándonos que el móvil que tenemos en otra habitación acaba de recibir un mensaje proveniente de la nevera que nos aconseja que es el momento de hacer la compra porque está a punto de acabarse la leche, mientras nuestra *Smart TV* nos está sugiriendo grabar un nuevo programa que se ajusta a nuestras preferencias… todo al mismo

tiempo y multiplicado por muchísimos más servicios. Es fácil percibir que, de este modo, ***un entorno ubicuo podría llegar a ser molesto para los usuarios, si estos fueran interrumpidos constantemente de una forma poco apropiada***.

Además, esta problemática se agrava si sucede en un entorno de trabajo, ya que ***las interrupciones, además de ser molestas, pueden afectar drásticamente a la productividad, a la falta de atención y***, por consiguiente, ***a la propensión a cometer errores***. Por ejemplo, existen estudios que demuestran que una simple interrupción de 15 segundos puede provocar que una persona pierda de su mente parte de la lista de tareas que debía llevar a cabo [6]. Y no hablamos simplemente de molestias y un descenso en la productividad. Como se comenta en [5], "…*para pilotos, conductores, soldados y doctores, errores de falta de atención pueden ser absolutamente peligrosos*".Por lo tanto, ***debe existir una forma sistemática de evitar las interacciones intrusivas***, proveyendo un soporte para ayudar a controlar dichas interacciones, evitando que los usuarios puedan ser molestados frecuentemente y sean así capaces de centrarse en sus tareas [4].

Además, en la *computación ubicua* ***los servicios deben comportarse de una manera "considerada"*** (de un modo respetuoso, y no tosca y desconsideradamente como lo hacen ahora), requiriendo la atención del usuario estrictamente cuando sea necesario, ya que ***la atención del usuario es un recurso valioso y limitado*** [5]. Por ejemplo, un servicio de antivirus que se comportase de modo adecuado sabría que cuando descargue una nueva actualización de su base de datos de virus, no debería notificárselo al usuario si este está por ejemplo hablando con otras personas o está en medio de una conferencia, o si es de noche y todo está en silencio (sobre todo si lo hace acústicamente, como el archiconocido antivirus de "*la base de datos de virus, ha sido actualizada…*"). En el diseño de sistemas ubicuos, la atención humana se convierte en el recurso más

crítico, por encima de otros recursos como el procesador, la memoria o el ancho de banda [22], existiendo la necesidad de diseñar la interacción de los servicios de forma que ***se debe ser capaz de percibir la situación de atención de los usuarios y adaptar la interacción en base a ella***.

Sin embargo, debe existir un equilibrio entre una interacción no intrusiva y respetuosa y una interacción totalmente transparente e invisible, donde los servicios no requieran interactuar con el usuario ni le notifiquen nada en absoluto, ya que entonces los usuarios pueden advertir la no deseada sensación de que su entorno está fuera de su control [21].

Por lo tanto, a la hora de diseñar un sistema ubicuo, para tener en cuenta todos estos factores comentados, ***necesitamos diseñar un sistema donde los servicios tengan un nivel de molestia*** (el grado con que cada servicio se inmiscuye en la mente de los usuarios) ***adecuado***, ***acorde a la atención del usuario en cada momento***, consiguiendo que los servicios se comporten de una manera respetuosa y satisfactoria para el usuario. Como este nivel apropiado de molestia dependerá del contexto del usuario, la implementación de soluciones predefinidas en tiempo de diseño no son adecuadas, y deberá ser posible una adaptación a dicho contexto en tiempo de ejecución.

1.2 Objetivos de la tesis

Dentro del ámbito de las *tecnologías ubicuas*, la gestión de la interacción entre servicios y usuarios es un campo en desarrollo que todavía no ha sido plenamente explorado. Como manifiestan los ejemplos presentados en el apartado anterior, es necesaria una investigación más profusa hasta encontrar una solución al problema de

que dicha interacción se realice de una manera satisfactoria para el usuario, acorde con sus necesidades y preferencias, siempre cambiantes a lo largo del tiempo.

¿Cómo conseguir esta “interacción satisfactoria”, que sepa adaptarse a las necesidades en cada momento? Esta tesis tendrá como objetivo responder a esta pregunta, desarrollando un sistema que logre adaptar el ***nivel de molestia*** de los servicios (el grado con que cada servicio se inmiscuye en la mente de los usuarios), aprendiendo las preferencias de los usuarios gracias a un algoritmo de aprendizaje.

Así, ***el objetivo principal de esta tesis es desarrollar un sistema que aprenda del feedback de los usuarios para adaptar los modelos de interacción de los servicios*** (definidos en tiempo de diseño), de forma que puedan soportar los cambios que se suceden en las preferencias de los usuarios ***a lo largo del tiempo***. Concretamente, el sistema ajustará en tiempo de ejecución el diseño inicial de molestia, haciendo uso de una aproximación de aprendizaje por refuerzo, donde se explota el propio *feedback* de los usuarios ante las notificaciones de los servicios para adaptar el nivel de molestia de estos de una manera satisfactoria. Además, también será objetivo del sistema aportar una interfaz de usuario por medio de la cual los usuarios puedan recibir las notificaciones y devolver su *feedback*.

Como está diseñado e implementado este sistema, el *framework* del espacio de molestia sobre el que se desarrolla y los principios del algoritmo de refuerzo en que se basa, serán conceptos y cuestiones analizadas y respondidas detalladamente en los sucesivos capítulos.

1.3 Establecimiento del problema

Como se ha analizado durante los apartados anteriores, con la *computación ubicua* existe una creciente utilización de servicios, en cualquier lugar y en cualquier momento, desde distintos entornos, con diferentes particularidades (ambientales, de visibilidad, sonoridad, etc.). Esta característica ubicua trae consigo la necesidad de buscar una interacción satisfactoria con el usuario, adaptándose en cada momento a sus necesidades, cambiantes en el tiempo, de forma que estas notificaciones no sobrecarguen a los usuarios, llegando incluso a la molestia [19][20]. Por tanto, se evidencia la existencia de una problemática y una clara necesidad de que los servicios sepan adaptarse a las necesidades de los usuarios en cada momento.

¿Cómo adaptar el espacio de adaptación de molestia, establecido en tiempo de diseño, para soportar estos cambios en las preferencias del usuario a lo largo del tiempo? Esta será la pregunta que se responderá en el siguiente apartado, cuando se presente la solución propuesta.

1.4 Solución propuesta

Con el propósito de resolver la problemática planteada en el apartado anterior, el *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València*[2] ha realizado una serie de estudios en el ámbito de la auto-adaptación de preferencias en el contexto de la computación ubicua.

Partiendo de sus avances en la línea de investigación sobre la adaptación de la molestia/intromisión de la interacción de servicios por

[1] http://www.pros.upv.es
[2] http://www.upv.es

medio del *feedback* del usuario, esta tesis complementa dichas investigaciones con el desarrollo de un sistema que implementa un algoritmo de aprendizaje capaz de conseguir la adaptación del nivel de molestia en tiempo de ejecución, supliendo así la necesidad presentada en forma de "*trabajo futuro*" en [15].

Para su desarrollo, se seguirá una estrategia de ***aprendizaje por refuerzo*** [13], en la que el sistema aprende gracias al *feedback* suministrado por el usuario mediante el propio uso del sistema, tanto con el *feedback* explícito (cuando el usuario indica que le gusta/no le gusta una interacción), como con el suministrado implícitamente (por ejemplo, cuando un servicio notifica algo al usuario mediante una señal acústica y el usuario silencia el volumen inmediatamente, supuestamente porque le ha molestado, luego el servicio debería disminuir su nivel de molestia). Concretamente, se definirá un sistema de aprendizaje de la molestia producida por los servicios en su interacción, para auto-adaptar dicha molestia a las preferencias y necesidades del usuario y maximizar la satisfacción de este.

La implementación de este sistema será capaz de auto-adaptar el espacio de molestia (*Obtrusiveness Space*, un espacio constituido por los diferentes niveles de molestia por donde puede transitar un servicio, teniendo asignado cada nivel de molestia un grado de interacción diferente) de los servicios introducidos en el sistema, acorde al *feedback* del usuario. Si el *feedback* es positivo, significará que le complace la forma en que el servicio está interactuando, y por lo tanto el nivel de molestia del servicio no debe ser modificado, mientras que si el *feedback* es negativo, habrá que ir afinando su nivel de molestia.

1.5 Contexto de la tesis

El trabajo presentado en esta tesis se desarrolla en el contexto de la *computación ubicua*, concretamente dentro del marco de la *auto-adaptación de preferencias en dispositivos ubicuos*, complementando los trabajos de investigación llevados a cabo en este ámbito por el *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València*[2].

En particular, se engloba en el contexto de la adaptación del espacio de molestia en la interacción entre servicios y usuarios, desarrollando un sistema basado en un algoritmo de aprendizaje por refuerzo que, partiendo de un espacio de molestia asignado a un servicio en la etapa de diseño, auto-adapte en tiempo de ejecución el nivel de molestia del servicio en dicho espacio (de forma que los servicios interactúen con los usuarios satisfactoriamente, acorde a las preferencias y necesidades de estos en cada momento).

1.6 Estructura de la documentación

La Tesina está dividida en nueve grandes capítulos, incluido este capítulo introductorio. A continuación se detalla brevemente el contenido de cada uno de los capítulos siguientes, como guía de la organización del resto de la tesis:

Capítulo 2: Ofrece un repaso general a algunas de las tecnologías que han estado implicadas en este proyecto.

Capítulo 3: Introduce los conceptos principales relacionados con el trabajo presentado en esta tesis, para dar al lector los conocimientos básicos necesarios para su mejor comprensión.

[1] http://www.pros.upv.es
[2] http://www.upv.es

Capítulo 4: Este capítulo introduce otros trabajos relacionados con el aprendizaje y adaptación de preferencias a partir del *feedback* de usuario.

Capítulo 5: Describe el sistema desarrollado en esta tesis, empezando por presentar el *framework* que lo fundamenta, siguiendo por las formas de captar el *feedback* del usuario, y terminando por analizar el algoritmo de aprendizaje por refuerzo en que se basa el sistema.

Capítulo 6: Presenta el análisis de nuestro proyecto, los requisitos que debe cumplir, su funcionalidad por medio de casos de usos, el modelado conceptual del sistema y los diagrama de secuencia y de actividades.

Capítulo 7: Muestra como se ha realizado el diseño y la implementación del sistema, presentando su arquitectura, el interfaz del sistema, las estructuras de datos empleadas y la implementación del algoritmo de aprendizaje.

Capítulo 8: Muestra un ejemplo de uso del sistema, con un caso de estudio bajo distintos escenarios, para demostrar la funcionalidad del sistema desarrollado, así como su forma de uso. También se mostrarán los resultados de una evaluación experimental basada en un ejemplo.

Capítulo 9: Resume, a modo de conclusiones, todo el trabajo presentado en esta tesis, remarcando las ideas principales. Enumera también unas líneas de ampliación de este trabajo en forma de trabajos futuros.

CAPÍTULO 2

Tecnologías involucradas

A lo largo de este capítulo se va a presentar el contexto tecnológico que ha envuelto el desarrollo de nuestro proyecto. El primer elemento introducido será Eclipse, la plataforma de desarrollo sobre la que se ha implementado nuestra aplicación, y del que se presentaran sus principales características junto a una visión de su Arquitectura.

Se presentarán también dos *plugins* para Eclipse, *WindowBuilder,* un editor de interfaces gráficas, gracias al cual hemos podido convertir nuestro proyecto de una aplicación de consola en una gráfica, y EMF, *framework* que se ha utilizado para construir los modelos XMI de los que nuestro sistema carga los datos al arrancar.

Posteriormente se presentará el lenguaje de programación utilizado en nuestro desarrollo, Java, centrándose en su versión Java SE que ha sido la que se ha utilizado en este proyecto.

Por último, se introducirá el framework OSGi, utilizado para facilitar la lectura de los modelos EMF.

2.1 Eclipse

Eclipse[1] es un entorno de desarrollo integrado (un IDE, por sus siglas en inglés), de código abierto y multiplataforma. Es una potente y completa plataforma de programación, originalmente desarrollada para *Java* pero que, gracias a su ampliación con *plugins,* se ha extendido a otros lenguajes de programación como *C/C++* o *Python*. Debido a su enorme flexibilidad, ha dado lugar a otros IDEs especializados (como *intelliJ IDEA* o *Android Studio*).

Algunas de sus características más importantes son:

- Es un entorno multiplataforma, altamente personalizable.
- Posee un completo editor de código, con características como "*ContextAssist*" (similar al "*intellisense*" de *Visual Studio*), que hace sugerencias automáticamente para completar órdenes, tipos, etc., o "*Code highlighting*", con la cual resalta mediante colores sintaxis y estructuras del código. Permite buscar y reemplazar palabras y encontrar código duplicado y "malas prácticas", ayudando a conseguir código de calidad, así como el uso de "*code folding*", con el que podemos replegar a una sola línea un bloque de código.
- Hace uso de *perspectivas*, cada una de las cuales proporciona una serie de vistas adecuadas al desarrollo de un tipo de tarea especifico.
- Dispone de asistentes (*wizards*), que asisten a la hora de realizar tareas, como por ejemplo la creación de proyectos, de clases, una refactorización, un test, etc.
- Mediante la posibilidad del uso de *plugins*, ofrece integración con herramientas externas, como por ejemplo *JUnit* (herramienta

[1] http://www.eclipse.org/

para realizar pruebas unitarias al código) o *Ant* (herramienta de construcción de proyectos).

- Es un entorno con abundante documentación.

2.1.1 Arquitectura Eclipse

Basándose en la experiencia con el predecesor de Eclipse, *Visual Age* para Java de IBM, los arquitectos diseñaron una arquitectura para eclipse fácilmente extensible. En la figura 2, se muestran los principales componentes de esta arquitectura [9], siendo los siguientes:

- Plataforma Eclipse (*Eclipse Platform*): es el componente central de Eclipse, y es un *framework* genérico para la construcción de entornos de desarrollo.
- Herramientas de Desarrollo Java (o JDT, siglas de *Java Development Tools*): Completo y potente entorno de desarrollo Java, construido usando la propia plataforma Eclipse.
- Entorno de Desarrollo de Plugins (o PDE, siglas de *Plug-in Development Environment*): proporciona un entorno para la creación de *plugins* para Eclipse.

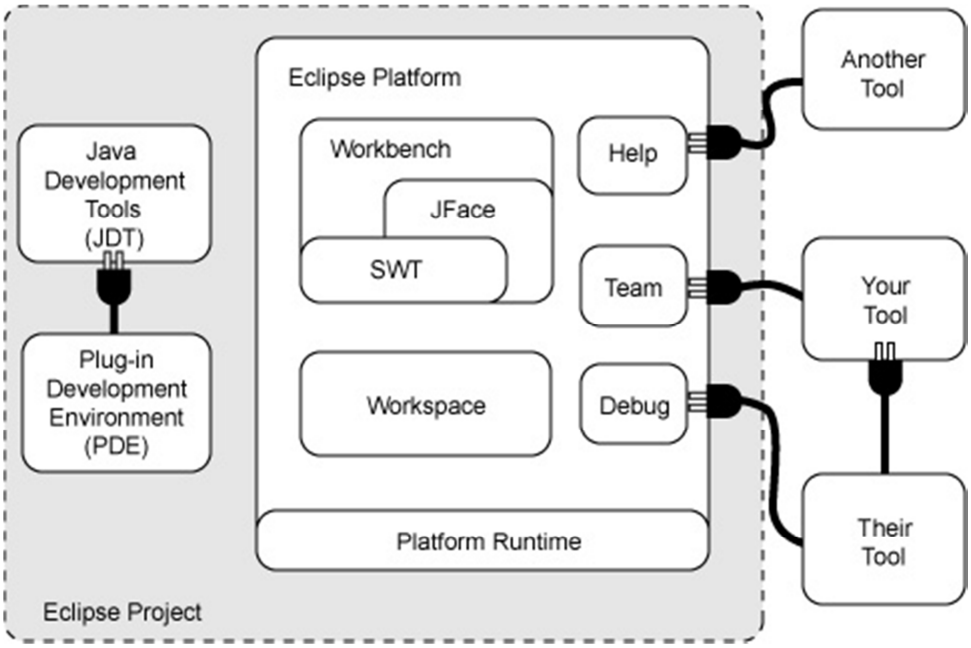

Figura 2 – Arquitectura Eclipse [8]

Nuestro desarrollo ha hecho uso del componente JDT para desarrollar el código de nuestra aplicación en Java (lenguaje de programación que abordaremos en el siguiente apartado).

2.1.2 WindowBuilder

WindowBuilder[1] es el editor estándar de Eclipse para interfaces gráficas de usuario (o GUI, siglas de *Graphical User Interface*), que permite usar componentes gráficos de java basados en *Swing* o *Awt* para generar GUIs a un alto nivel, facilitando al usuario la creación de formularios e interfaces gráficas (de manera muy similar a como se hace con el *Windows Forms* de *Visual Studio*).

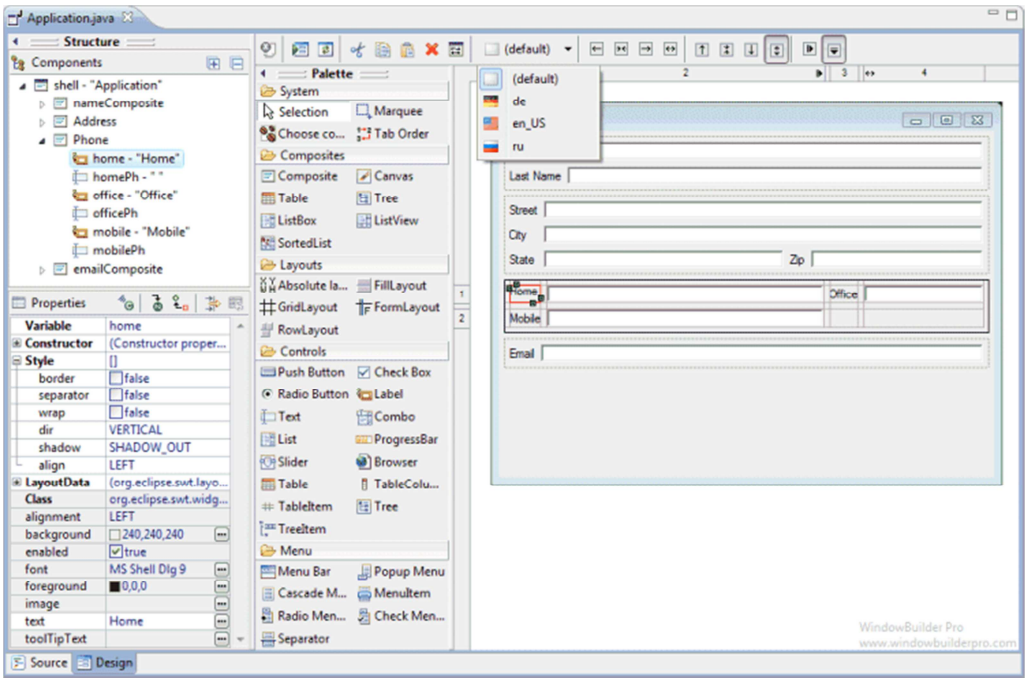

Figura 3 – Pantalla de *WindowBuilder*

Permite añadir controles a la interfaz gráfica usando *drag-and-drop* (arrastrar y soltar) y añadirles manejadores de eventos (permitiendo desarrollar una programación dirigida por eventos). Como se observa en la figura 3, también ofrece facilidades a la hora de modificar las

[1] http://www.eclipse.org/windowbuilder/

propiedades de los elementos gráficos, utilizando un editor de propiedades.

Está construido en forma de *plugin* para Eclipse, aunque en las últimas versiones de este ya se está incorporando por defecto. El *plugin* crea un árbol de sintaxis abstracta para navegar por el código fuente y utiliza GEF[1] (un *framework* de edición grafica) para visualizar y gestionar la presentación visual.

El código generado, no requiere de ninguna biblioteca adicional para ser compilado y ejecutado, pudiendo usarse sin necesidad de tener *WindowBuilder* instalado.

2.1.3 EMF

EMF[2] (siglas de *Eclipse Modeling Framework*) es un *framework* de modelado y facilidad de generación de código que sirve para construir aplicaciones dirigidas por modelos (basadas en un modelo de datos estructurado).

Partiendo de una especificación del modelo descrita en XMI, EMF suministra herramientas para producir un conjunto de clases Java del modelo, un conjunto de clases que permiten su visualización y un editor básico. Los Modelos pueden especificarse usando Java anotado, documentos XML o UML creado con herramientas de modelado como *Rational Rose*, y después ser importados a EMF.

Una vez se tenga modelada la aplicación, EMF se encargará de generar código sin errores y totalmente utilizable, basándose en su modelo.

[1] http://www.eclipse.org/gef/
[2] http://www.eclipse.org/modeling/emf/

En nuestro desarrollo se utilizaron modelos EMF para importar los servicios disponibles en un sistema preexistente al nuestro, de manera que nada más arrancar nuestra aplicación tuviera disponibles dichos servicios para poder operar con ellos ajustando sus niveles de molestia (asignados en la etapa de diseño).

2.2 Java

Java[1] es, en primera instancia, un lenguaje de programación orientado a objetos, y, en segunda, una potente, robusta, segura y universal plataforma de programación con una amplia y diversa colección de librerías integradas entre sí, que permiten desarrollar aplicaciones informáticas actuales, con un alto grado de complejidad y funcionalidad [36].

Algunas de sus principales características son las siguientes:

- Es un lenguaje orientado a objetos de baja complejidad.
- Es interpretado: se compila y seguidamente se ejecuta sobre la máquina virtual de Java (o JVM, siglas de *Java Virtual Machine*).
- Las aplicaciones desarrolladas con Java son portables entre múltiples plataformas.
- Las aplicaciones desarrolladas con Java son robustas, ya que la JVM se encarga de manejar la memoria (sin posibilidad de accesos ilegales), es totalmente tipificado, etc.
- Las aplicaciones son seguras por varios motivos, entre ellos que la JVM protege contra los virus y contra aplicaciones hostiles.
- Permite programación de hilos, mejorando así las prestaciones y funcionalidad de determinadas aplicaciones.
- Permite el diseño y desarrollo de sistemas con arquitecturas complejas de programas.

[1] http://www.oracle.com/es/technologies/java/overview/index.html

2.2.1 Java Platform Standard Edition

Como hemos comentado en el apartado anterior, Java no solo es un lenguaje de programación, también es una plataforma de desarrollo, que, según las necesidades y el área en que se esté desarrollando, permite centrarse y utilizar solamente ciertas herramientas de la plataforma. Por lo tanto, dependiendo de estas necesidades, podremos elegir entre los tres tipos de ediciones de Java existentes, cada una orientada a un área de desarrollo especifica:

- *Java Platform, Standard Edition* (**Java SE**): Orientada al desarrollo de aplicaciones independientes de la plataforma.
- *Java Platform, Enterprise Edition* (**Java EE**): Orientada al desarrollo de aplicaciones de entorno empresarial.
- *Java Platform, Micro Edition* (**Java ME**): Orientada a dispositivos con capacidades restringidas, como móviles o dispositivos embebidos.

Concretamente, para el desarrollo de nuestra aplicación, se ha hecho uso de la versión *Java SE 1.7*. En la figura 4 se muestran los principales componentes de esta versión, entre los cuales podemos destacar el componente de las bibliotecas de interfaz de usuario, que hemos utilizado a través del diseñador de interfaces gráficas *WindowBuilder*, presentado en el apartado anterior.

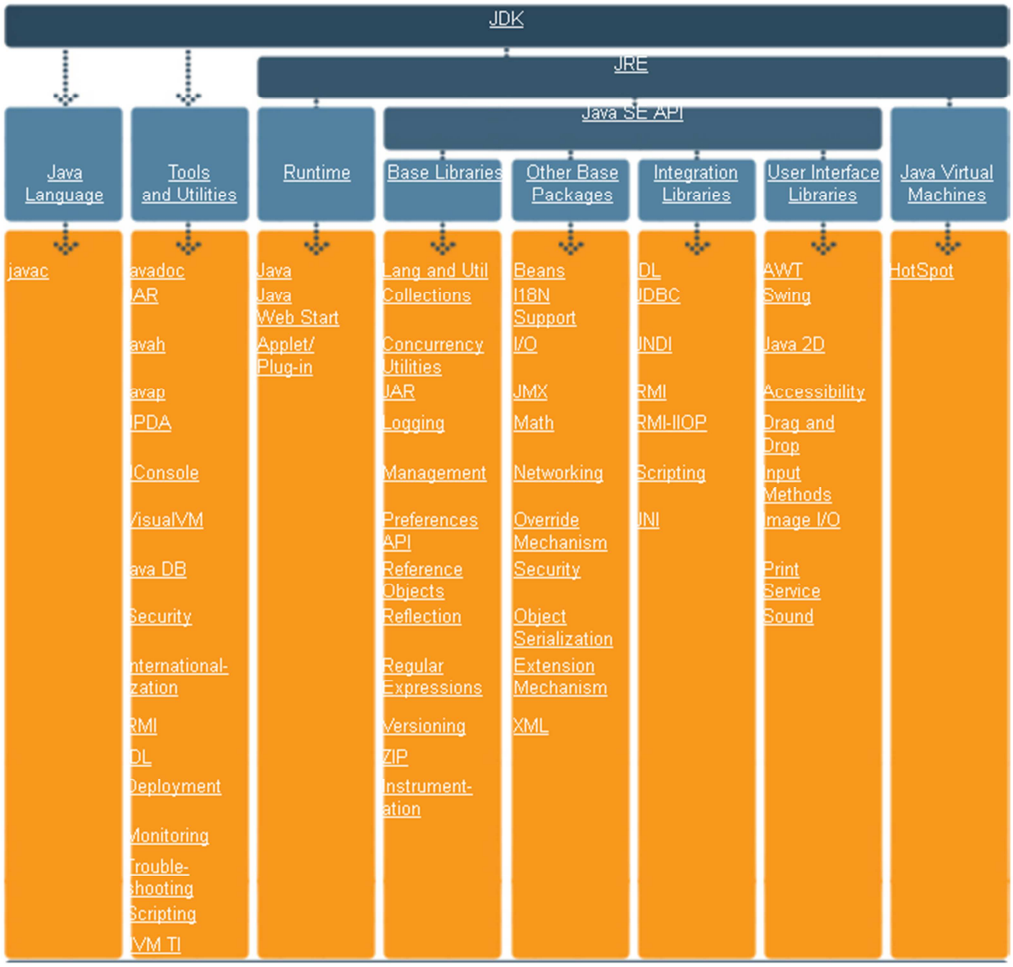

Figura 4 – Diagrama conceptual con los componentes de Java SE

El lenguaje Java de Java SE, es un lenguaje de programación orientado a objetos (OO) con componentes de alto nivel (como por ejemplo el recolector de basura o la clase *String*). Produce código independiente de la plataforma, precompilado en forma de ***bytecode***, que será ejecutado por la máquina virtual de Java (o JVM, siglas de *Java Virtual Machine*).

2.2.2 Tecnología OSGi

OSGi (siglas de *Open Services Gateway Initiative*), es un *framework* para Java, que establece las formas de crear módulos y la manera en que estos interactuarán entre sí en

tiempo de ejecución. OSGI intenta solventar los problemas del tradicional "*classloader*" de la máquina virtual y de los servidores de aplicaciones Java. En OSGI, cada módulo tiene su propio classpath separado del resto de classpaths de los demás módulos [7].

Este *framework* proporciona a los desarrolladores un entorno orientado a servicios y basado en componentes, ofreciendo estándares para manejar los ciclos de vida del software. Sus principales características son:

- Es un sistema de módulos para la plataforma java.
- Se trata de una arquitectura orientada a servicios.
- Los servicios pueden ser registrados y consumidos dentro de la JVM.
- Es dinámico. La instalación, arranque, parada, actualización y desinstalación de *bundles* (aplicaciones empaquetadas en ficheros jar, que se despliegan en una plataforma OSGi) se realiza dinámicamente en tiempo de ejecución, sin tener que detener por completo la plataforma.

Resumiendo estas características, OSGi proporciona un marco de trabajo Java de uso general, seguro y administrado que soporta el despliegue dinámico de aplicaciones *bundles* [29].

Como se mostrará posteriormente en el apartado 7.1.2.3 sobre el nivel de persistencia, nuestro sistema ha utilizado la tecnología OSGi para facilitar el acceso al modelo EMF desde el cual se cargarán los servicios con los que se interactuará.

CAPÍTULO 3

Conceptos básicos

Como se comentó durante el capítulo de introducción, esta tesis aborda el desarrollo de un sistema basado en un algoritmo de aprendizaje por refuerzo, para la auto-adaptación del nivel de molestia en la interacción de servicios. Con el fin de introducir los fundamentos en que se apoya nuestra aproximación y proporcionar una base de fondo para su mejor comprensión, a lo largo de este capítulo vamos a presentar una serie de conceptos básicos necesarios para entender el trabajo realizado en esta tesis.

En el apartado 3.1, se ampliará la definición de un concepto ya introducido en los capítulos anteriores, el concepto de *molestia* o *intromisión* (comúnmente referenciada mediante su nombre en inglés *obtrusiveness*), presentando el *framework* para interacciones implícitas en que está basado [6], el *espacio de adaptación de molestia* y el concepto de *nivel inicial de molestia* [14].

Ya que esta tesis hace uso de conceptos de las disciplinas de *Interacción persona-Computador*, Computación en función del Contexto y la Computación respetuosa (más conocidas por sus respectivos nombres en inglés, *Human-Computer Interaction (en adelante HCI)*, *Context-Aware Computing* y *Considerate Computing*), estando su ámbito englobado dentro de la intersección de las tres, tal como muestra la figura 5, también se presentarán los conceptos básicos de cada una de ellas [16].

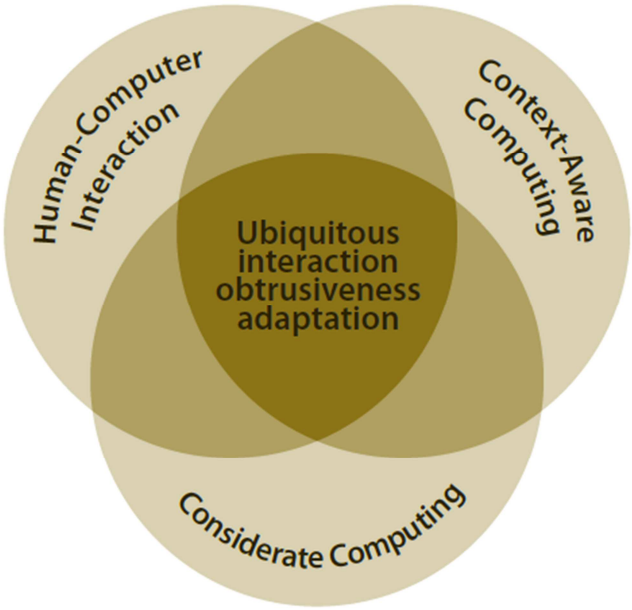

Figura 5 – Disciplinas relacionadas con esta trabajo.

Por lo tanto, en el apartado 3.2 se introduce *HCI*, introduciendo las modalidades de interacción presentes en la interacción del programa implementado en esta tesis. El apartado 3.3 proporciona una visión general del *Context-Aware Computing* y presenta la técnica de *aprendizaje por refuerzo*, base del algoritmo implementado. Por último, el apartado 3.4 introduce el concepto del paradigma de la C*onsiderate Computing*, volviendo a remarcar, tal como se hizo en la introducción, los problemas que conllevan las interrupciones por parte de los servicios.

3.1 Molestia/intromisión (obtrusiveness)

Para definir el espacio de adaptación de molestia y la configuración de la molestia inicial de los servicios, este trabajo hace uso del *framework* conceptual presentado en el trabajo sobre el ***diseño de interacciones implícitas*** [6].

Como ilustra la figura 6, este *framework* define dos dimensiones para caracterizar las interacciones implícitas:

Iniciativa:

Atendiendo a la dimensión de iniciativa, podemos caracterizar las interacciones en ***reactivas*** (el usuario es quien inicia la interacción, bien explícitamente o bien implícitamente) y ***proactivas*** (los servicios inician la interacción, suministrando información no solicitada).

Atención:

Atendiendo a la dimensión de atención, podemos especificar que las interacciones pueden darse en ***primer plano*** (el usuario será plenamente consciente de la interacción) o en ***segundo plano*** (la interacción pasará desapercibida para el usuario).

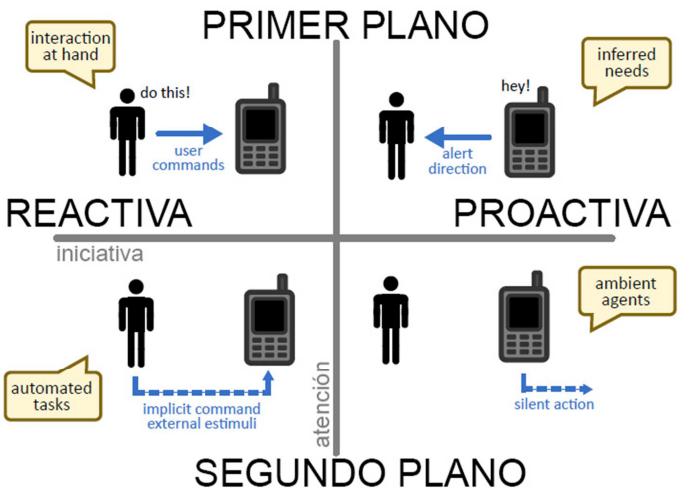

Figura 6 – Framework conceptual de diseño de interacciones implícitas.

Los diseñadores subdividen lógicamente cada uno de los ejes en tantas partes como necesiten para definir los niveles de molestia deseados. Por ejemplo, el espacio de adaptación de molestia de la figura 7, ha dividido su eje de iniciativa en dos, reactivo y proactivo, y su eje de atención en tres, ***Invisible*** (el usuario no percibirá la interacción), *Ligeramente apreciable* (***slightly-appreciable***) (el usuario puede no percibir la interacción, a menos que haga algún esfuerzo) y *Plenamente consciente* (***completely-aware***) (el usuario será plenamente consciente de la interacción, aun incluso cuando esté realizando otras tareas en el sistema).

3.1.1 Espacio de adaptación de molestia

El *framework* conceptual utilizado por nuestra solución sitúa cada una de las dimensiones comentadas arriba en un eje 2D. En el eje vertical sitúa la *iniciativa* y en el eje horizontal la *atención*, conformando una especie de matriz de niveles de molestia denominada ***espacio de adaptación de molestia*** (comúnmente referenciada mediante su nombre en inglés, *obtrusiveness adaptation space*).

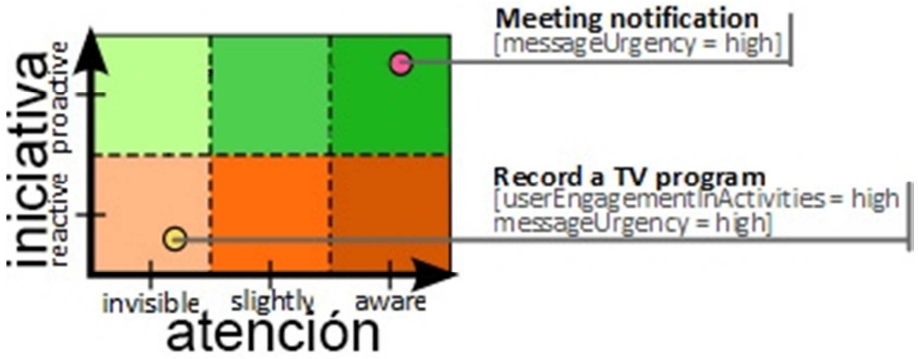

Figura 7 – Espacio de adaptación de molestia.

Cada uno de los niveles de este *espacio de adaptación de molestia* tendrá asociada una configuración de interacción diferente. En la etapa de diseño del sistema, los diseñadores no solo decidirán qué servicios son necesarios para cada usuario (acorde a su perfil), sino que caracterizarán cada uno de estos servicios dentro de este *espacio de*

adaptación de molestia, en una posición conforme con la forma de iteración deseada por el usuario, según sus necesidades y preferencias iniciales. Por ejemplo, como ilustra la figura 7, un usuario puede preferir inicialmente que el servicio encargado de grabar programas de televisión interactúe en modo *invisible (en segundo plano)*, para no ser molestado por ningún mensaje referente a la grabación de un programa, y a la vez que interactúe de manera reactiva, para que sea el usuario el que decide iniciar o no la interacción (decirle al programa que y cuando tiene que grabar), mientras prefiere que su servicio de notificación de citas le avise automáticamente (por sí mismo) cada vez que tenga información que notificar, y en *primer plano*, de modo que siempre sea totalmente consciente para el usuario, incluso aunque este estuviera haciendo alguna otra tarea en el sistema.

3.1.2 Adaptación de las preferencias a lo largo del tiempo

Los hábitos de los usuarios evolucionan a través del tiempo [12]. Los niveles de molestia, seleccionados inicialmente en tiempo de diseño para cada servicio según las preferencias de cada usuario, deben adaptarse a las necesidades y preferencias del usuario en cada momento (no solo a las que tenía en un principio), para así garantizar su satisfacción, ya que dichas preferencias pueden cambiar debido a un cambio en su estilo de vida (por ejemplo si ha conseguido un nuevo trabajo, o si ha vuelto a estudiar, etc.) o un cambio en sus necesidades.

Como ejemplo, imaginemos un usuario desempleado, que opina que el servicio de *atención sanitaria* (*healthcare*) es muy importante para él, por lo que lo tiene configurado para estar en el nivel máximo de atención (el nivel *aware*), donde recibirá un aviso sonoro cada vez que reciba una notificación de este servicio (por ejemplo, que tiene cita con el dentista, o con su médico de cabecera para la revisión anual). Como está desempleado y no asiste a clases, no le molesta recibir esas

notificaciones. Sin embargo, supongamos que ahora ha conseguido un trabajo, pasando la mayor parte del día rodeado de más personas, y no quiere que dichas personas sean conscientes de las notificaciones de ese servicio de atención sanitaria (además de por educación hacia los demás, porque este tipo de información es muy privada). Por lo tanto, este nivel de molestia para el servicio tendría que ser adaptado a otro nivel que requiere menos atención, como ilustra la figura 8:

Figura 8 – Ejemplo de adaptación de nivel de molestia.

Para hacer frente a este problema y adaptar la molestia de interacción de los servicios a las nuevas preferencias y situaciones, el sistema debería como mínimo permitir a los usuarios configurar sus nuevas preferencias para cada servicio explícitamente por medio de una interfaz. Sin embargo, tal como hemos comentado anteriormente en la introducción, al existir en las tecnologías ubicuas un número creciente de servicios, y al mismo tiempo la tarea de configurar cada servicio puede ser muy tediosa, los usuarios olvidan a menudo establecer o cambiar las configuraciones, dando como resultado multitud de interrupciones no deseadas [11] que, como ya expusimos en el *apartado 1.1*, pueden producir efectos negativos más allá de la propia molestia [5][6].

Por lo tanto, en este trabajo se propone auto aprender estas preferencias y adaptar el diseño inicial automáticamente, de manera que se mantenga la satisfacción del usuario. El sistema lo logrará aprendiendo de la reacción del usuario (de su *feedback*) después de recibir una notificación en un nivel de molestia específico, tal como se detallará durante el capítulo 5.

3.2 Human-Computer Interaction (HCI)

La disciplina *Human-Computer Interaction* (en adelante HCI), estudia la interacción entre humanos (los usuarios) y los ordenadores [23] con el objetivo de mejorarla, optimizando la *usabilidad* de los ordenadores y haciéndolos más receptivos ante las necesidades de los usuarios.

3.2.1 Modalidades de interacción

En términos generales, desde el punto de vista de la intención del usuario, podemos observar dos modalidades de interacción:

Una en la que el usuario interactúa conscientemente con el sistema (explícitamente), y otra en que el usuario interactúa subconscientemente (implícitamente), existiendo una graduación para cada uno de los modos. La figura 9 muestra un modelo que explica el concepto de interacción explícita e implícita entre humanos y computadores.

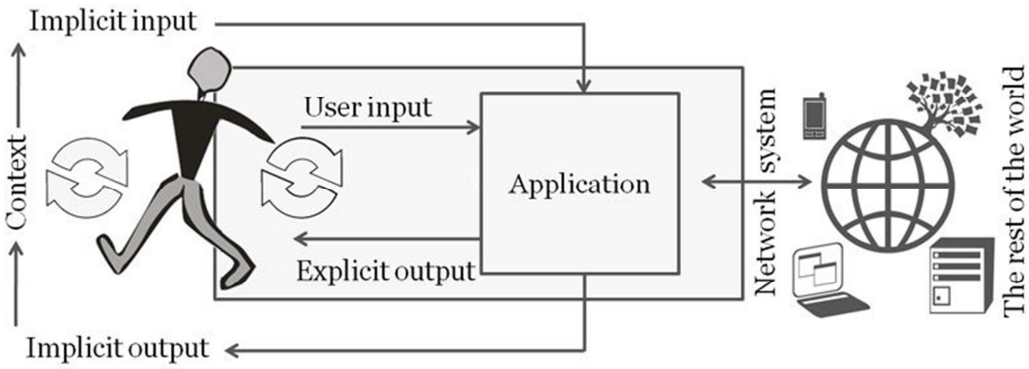

Figura 9 – Modelo de interacción explicita e implícita en HCI [25]

Interacción explicita:

Este tipo de interacción se produce cuando el usuario establece la interacción con plena intencionalidad, por medio de comandos de consola, o pulsando en algún botón/opción desde una interfaz

gráfica, por medio de comandos de voz, etc., pero siempre siendo su intención “consciente” establecer una interacción.

Interacción implícita:

El segundo modo de interacción, la *interacción implícita* [24], apunta a la interacción que ocurre de manera subconsciente para el usuario (como por ejemplo cuando en la comunicación entre personas se añade inconscientemente una información contextual, como los gestos, el lenguaje corporal y voz). El usuario ejecuta una acción, que denota una preferencia, lo que puede ser interpretado como una interacción.

Aunque los cerebros humanos pueden captar muy bien este tipo de información, en HCI esta clase de interacción es difícil de capturar y está sujeta a errores, ya que una acción subconsciente del usuario puede ser entendida como una preferencia cuando en realidad no lo es.

3.3 Context-Aware Computing

El *Context-aware Computing* es un paradigma de computación donde la información del entorno de los usuarios, su ***contexto*** [10] (cualquier información que pueda ser utilizada para caracterizar su situación, su localización, el medio ambiente, etc.), es aprovechada para anticipar las necesidades inmediatas de estos, logrando que los servicios se anticipen proactivamente a las necesidades de los usuarios en cada situación particular, de manera que les proporcionen una asistencia adecuada [25].

De esta manera, los computadores tienden a “desvanecerse”, a convertirse en unos objetos que existen en “el fondo” y a los que no debemos prestar atención (como por ejemplo hoy en día no se presta atención al tendido eléctrico que nos suministra electricidad), pero que nos ayudan a realizar nuestras tareas cotidianas [1].

3.3.1 Aprendizaje por refuerzo (*Reinforcement Learning RL*)

Como se ha visto en el apartado 3.1.2, los hábitos de los usuarios evolucionan a través del tiempo [12], implicando la necesidad de adaptar los niveles de molestia iniciales de cada servicio para que se adecuen a estos nuevos contextos. El *aprendizaje por refuerzo (RL,* de su nombre en inglés *Reinforcement Learning)* [13] está considerada como la solución para conseguir *feedback* sobre el contexto del usuario de manera cualitativa.

El aprendizaje por refuerzo es un enfoque donde un agente (en nuestro trabajo estaríamos refiriéndonos al servicio) actúa en un entorno y aprende de sus experiencias previas, de manera que adapta su comportamiento para maximizar la suma de las “recompensas” (refuerzos) que espera recibir en el futuro por su elección de acciones. La recompensa/castigo normalmente es un valor comprendido entre -1 y 1, donde 1 se corresponde con satisfacción total, -1 desaprobación y 0 es "no opinión".

La idea es que los agentes auto-aprendan a comportarse de manera óptima en base a querer maximizar la cantidad de recompensas que esperan obtener en el futuro, sin necesidad de que otro agente experto les guie indicándoles como hacerlo.

En un funcionamiento normal de aprendizaje por refuerzo, como el que ilustra la figura 10, un agente aprende políticas de toma de decisiones efectivas por medio de un proceso de ensayo y error interactuando con el sistema.

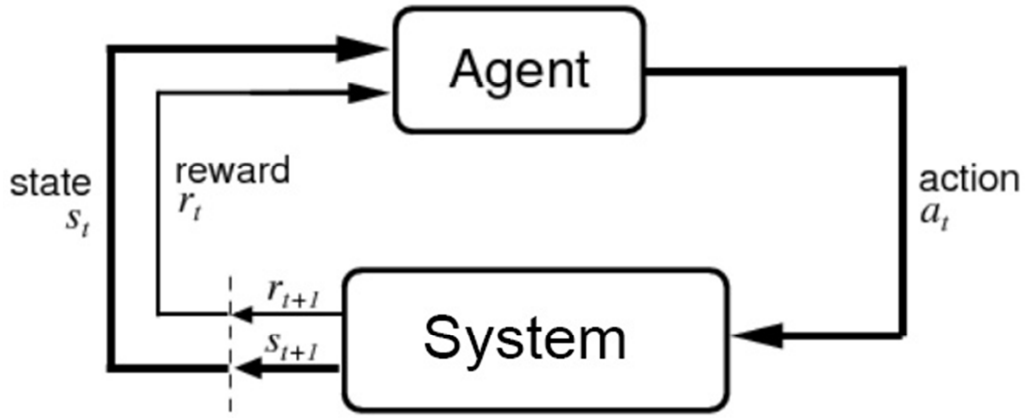

Figura 10 – Ciclo de interacción Agente-Sistema en RL.

Suponiendo que el tiempo avanza en pasos discretos (t1, t2=t1+1, t3=t2+1,…), cada interacción, por orden, consistirá en:

1) Observar el estado actual del sistema (s_t) en el instante t.
2) Llevar a cabo una acción (a_t) de las disponibles en el estado s_t.
3) Recibir una recompensa/castigo (r_t) (un valor numérico entre [-1,1] que el agente querría maximizar), seguido de una transición a un nuevo estado s_{t+1}.

Posteriormente, en el apartado *5.3.3*, presentaremos y analizaremos detenidamente el método de algoritmo de aprendizaje por refuerzo ***Q-Learning***, que sirve de base para la implementación desarrollada en esta tesis.

3.4 Considerate Computing

Tal como se expuso en el apartado 1.1, debido a que la atención del usuario es un recurso valioso y limitado, en la *computación ubicua* los servicios deben comportarse de una forma "considerada" [5], respetuosa con el usuario, requiriendo su atención solo y exclusivamente cuando sea necesario, y en un modo adecuado y cauteloso y no como lo hacen actualmente de manera tosca y desconsiderada.

Sin embargo, el aumento del exceso de información y la solicitud continua de la atención del usuario hacen las interrupciones una ocurrencia común [26].

Como se comentó en el apartado de la motivación, esta problemática de las interrupciones se agrava si sucede en un entorno de trabajo, ya que las interrupciones, además de ser molestas, inducen a la falta de atención, a la distracción, por lo que aumentan la propensión a cometer errores, y por consiguiente a afectar drásticamente a la productividad. Además no hablamos simplemente de molestias y un descenso en la productividad. Como se comenta en [5]: "…*para pilotos, conductores, soldados y doctores, errores de falta de atención pueden ser absolutamente peligrosos*".

Por lo tanto, ***deben evitarse las interacciones intrusivas***, proveyendo un soporte para ayudar a controlar dichas interacciones, evitando que los usuarios puedan ser molestados frecuentemente y sean así capaces de centrarse en sus quehaceres diarios [4].

3.5 Conclusiones

Durante este capítulo, se han presentado los conceptos básicos necesarios para un mejor entendimiento del trabajo realizado en la tesis. Se ha presentado el concepto fundamental de todo el trabajo, el concepto de ***espacio de adaptación de molestia*** (*obtrusiveness adaptation space*), una especie de matriz de niveles (cada uno con una configuración de interacción diferente) por donde puede transitar el ***nivel de molestia*** (el grado con que un servicio se inmiscuye en la mente del usuario) de los servicios. También se ha presentado el *framework* de ***diseño de interacciones implícitas*****,** en que dicho espacio de adaptación se basa.

Una vez analizado el concepto de *espacio de adaptación de molestia*, su significado y de que estudios proviene, junto con un ejemplo, se ha introducido la idea de que las preferencias y las

necesidades cambian a lo largo del tiempo, y de que por tanto ***se hace necesario ajustar los espacios de adaptación de molestia*** (que se personalizan en tiempo de diseño para un usuario determinado, bajo un contexto concreto) ***a las nuevas necesidades de los usuarios en tiempo de ejecución***, para que los servicios siempre interaccionen con los usuarios de manera satisfactoria para estos.

Por último, se han descrito de manera introductoria tres disciplinas que enmarcan a la solución propuesta en esta tesis, que son ***HCI*** (estudia la interacción entre humanos [los usuarios] y los ordenadores), la ***Considerate Computing*** (indica que los servicios deben comportarse de una forma respetuosa ("considerada") con el usuario, requiriendo su atención solo cuando sea necesario, evitando las interacciones intrusivas) y la ***Context-Aware computing*** (aprovecha la información del contexto de los usuarios para anticipar las necesidades inmediatas de estos, logrando que los servicios se anticipen proactivamente a las necesidades de los usuarios en cada situación particular), dentro de la cual se ha hecho hincapié en exponer las bases del ***aprendizaje por refuerzo (Reinforcement Learning)*** que, posteriormente en el apartado 5.3.3, nos serán de utilidad para analizar el algoritmo de la propuesta de esta tesis.

CAPÍTULO 4

Trabajos relacionados

Introducidos ya los conceptos básicos necesarios para entender nuestro trabajo y comentada la necesidad de que los servicios puedan adaptarse a las preferencias de los usuarios a lo largo del tiempo, en este capítulo de *trabajos relacionados* se van a presentar algunos trabajos desarrollados en este mismo ámbito, el de los mecanismos de aprendizaje automático de las preferencias de los usuarios, específicamente a partir del *feedback* de usuario ante una determinada situación (como por ejemplo ante una sugerencia, o ante un valor tomado para ajustar un elemento del entorno). También analizaremos las semejanzas y diferencias que existen entre todos los sistemas, y finalmente mostraremos una tabla comparativa entre dichos trabajos y el nuestro.

Los trabajos que se van a introducir serán los siguientes:

- *Uso de perfiles de usuario para el filtrado web* [27]:
  Aprender *perfiles de usuario* de internautas, mediante técnicas de clasificación, para proporcionar sugerencias en la navegación web.
- *Uso de agentes inteligentes embebidos en entornos inteligentes para realizar inteligencia ambiental* [30]*:*
  Aprender el comportamiento del usuario, mediante *reglas de lógica difusa,* con el objetivo de automatizar las acciones habituales.

4.1 Uso de perfiles de usuario para el filtrado web

Este trabajo consiste en el desarrollo de *PersonalSearcher*, un "*agente personal*" que guiará a los usuarios mientras navegan por la web. Cada vez que estos realicen una búsqueda, el sistema les sugerirá páginas adaptadas a sus preferencias.

Los ***agentes personales*** (*personal agents*) son programas software que aprenden las preferencias de los usuarios, para poder devolverles una asistencia personalizada de forma proactiva [28].Para proporcionar este asesoramiento personalizado, los agentes se basan en el conocimiento acerca de los usuarios que figura en sus ***perfiles de usuario***, que no son más que la información referente a sus intereses modelada mediante una ***jerarquía*** (en la que en el nivel superior se representan los intereses generales y en el nivel inferior los aspectos particulares de estos). Los agentes construirán estas jerarquías basándose en el tipo de web que los internautas visiten regularmente, a base de recoger de una manera transparente el *feedback* implícito que estos transmiten (el tiempo que ha permanecido en la página, cuanto deslizamiento ha realizado por ella, si la ha añadido a marcadores,…).

PersonalSearcher es el agente personal desarrollado por este trabajo que emplea la técnica de *perfiles de usuario* para guiar al usuario mientras navega por la web e ir obteniendo su *feedback*.

Como podemos observar en su esquema de funcionamiento en la figura 11, los usuarios interactuarán con el *PersonalSearcher* expresando sus necesidades de información en forma de palabras clave. El agente personal, cuando un usuario decida hacer una búsqueda de información, llevará a cabo una búsqueda paralela en los buscadores de internet más populares, y filtrará el extenso listado que

estos buscadores devuelvan en función del *perfil del usuario* (construido con las observaciones previas de los hábitos del usuario).

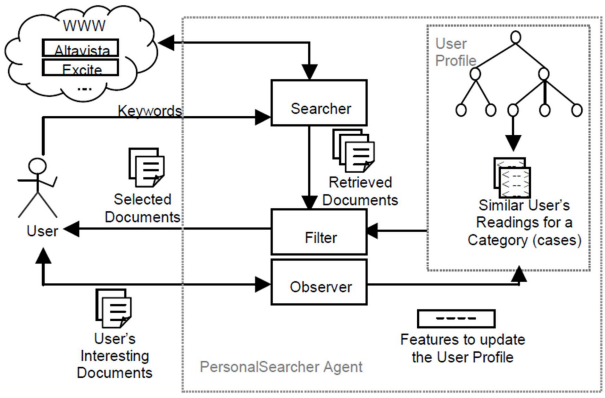

Figura 11 – Esquema de funcionamiento de *PersonalSearcher*

Para entender mejor como funciona *PersonalSearcher*, se va a presentar un ejemplo de uso: Se supondrá que el usuario del ejemplo ha navegado previamente varias veces por páginas sobre los lenguajes de programación Visual Basic y Java y por algunas de deportes como fútbol o deportes de motor.

El perfil de usuario, mostrado en la figura 12, revela como *PersonalSearcher* puede descubrir los temas significativos a partir de las páginas que el usuario ha ido visitando, distinguiendo los diferentes subtemas de los intereses del usuario.

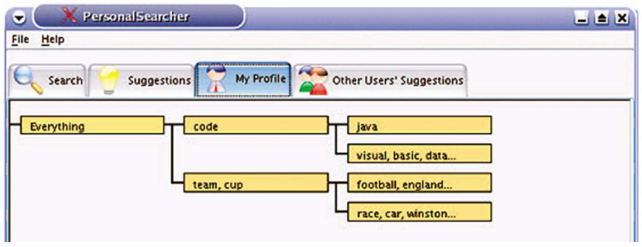

Figura 12 – Perfil de usuario

Para ir construyendo este perfil de usuario, por cada página que lee el usuario *PersonalSearcher* guardará un conjunto de 3 indicadores que estiman el nivel de interés que ha tenido el usuario en dicha página: El tiempo que ha permanecido en ella, cuanto deslizamiento por ella ha

realizado y si la página ha sido añadida a marcadores. Usando estos indicadores, *PersonalSearcher* obtiene las páginas relevantes para el usuario (sin molestarle ni distraerle de su navegación) y con ellas construye su *perfil del usuario* (en forma de una jerarquía de intereses).

Si este usuario realizara ahora una búsqueda en Internet utilizando por ejemplo las palabras clave "lenguajes de programación", el listado de páginas resultantes de la consulta podría incluir páginas sobre cualquier aspecto de la programación, como por ejemplo estudios teóricos sobre programación, información sobre distintos lenguajes de programación, etc., etc. Sin embargo, basándose en el conocimiento sobre los intereses de los usuarios presentes en sus *perfiles de usuario*, *PersonalSearcher* tiene la capacidad de hacer sugerencias. En nuestro ejemplo, tal como se muestra en la figura 13, *PersonalSearcher*, de entre todo el listado obtenido desde varios motores de búsqueda, sugiere dos páginas, en esta ocasión relacionadas con Visual Basic, junto a una indicación grafica de la relevancia esperada de cada una.

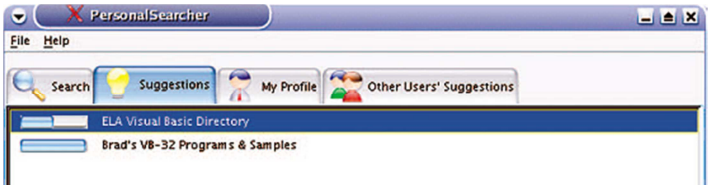

Figura 13 – Listado de sugerencias de PersonalSearcher

En esta ocasión, el agente personal ha descartado páginas sobre otros lenguajes de programación, considerándolas poco interesantes de acuerdo con el perfil del usuario. Así, *PersonalSearcher* determina la conveniencia de sugerir una página web al usuario mediante el cálculo de la relevancia de dicha página en relación con la jerarquía de los intereses del usuario del perfil.

Una vez *PersonalSearcher* ha presentado algunas sugerencias, se observa de nuevo el *feedback* del usuario ante dichas sugerencias para adaptar el perfil de acuerdo a la aprobación (*feedback* positivo) o no (*feedback* negativo) de estas sugerencias.

4.1.1 Evaluación experimental

Finalizamos este apartado sobre el trabajo de filtrado web con sus resultados experimentales. Se estudió un experimento con 10 usuarios evaluando el ***rendimiento del programa*** en base al *feedback* del usuario ante la relevancia de las páginas sugeridas (ya que el objetivo de *PersonalSearcher* es sugerir páginas que sean relevantes).

Cada uno de estos usuarios interaccionaron con *PersonalSearcher* buscando información por la web, y pidiéndole asesoramiento a intervalos regulares (cada vez que los usuarios leían 10 páginas). Cada vez que el agente, de acuerdo a los *perfiles de usuario* que iba creando, sugería páginas web, los usuarios devolvían un *feedback* en relación a las páginas sugeridas. La figura 14 muestra cual fue el rendimiento del programa a lo largo del tiempo, para las medias calculadas con el feedback de los usuarios.

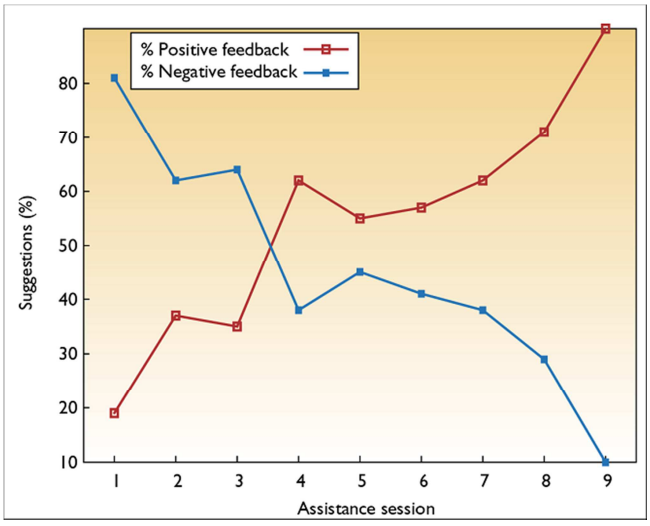

Figura 14 – Gráfico del rendimiento de PersonalSearcher en base al *feedback*

Los resultados de este experimento demostraron como, a medida que se va aprendiendo mejor el perfil del usuario, va aumentando el rendimiento del programa, pues el porcentaje de páginas con *feedback* positivo crece mientras que el de las páginas con *feedback* negativo decrece.

4.2 Uso de agentes inteligentes embebidos en entornos inteligentes para realizar inteligencia ambiental

En este apartado, se presentará un trabajo sobre el uso de *agentes inteligentes* de lógica difusa, embebidos en entornos inteligentes para realizar *inteligencia ambiental*.

Un ***agente inteligente*** es un sistema autónomo, capaz de percibir su entorno (mediante algún tipo de *sensores*), procesar la información percibida de las interacciones de los usuarios con dicho entorno (extrayendo de ella reglas de comportamiento) y actuar sobre dicho entorno (mediante actuadores), controlándolo de manera acorde a dichas reglas. Los usuarios también podrán interactuar con el entorno por medio de los actuadores (por ejemplo, aumentando el nivel de luz en una habitación). El *feedback* implícito resultado de esta interacción permitirá que los *agentes inteligentes* vayan aprendiendo y mejorando sus reglas y comportamiento, adaptándose al entorno en tiempo real.

Como es fácil deducir, es en referencia a estas capacidades de adaptación a las preferencias del usuario donde este sistema y el nuestro tendrán un nexo de unión, más allá de que ambos se muevan en el campo de la computación ubicua.

El *agente inteligente* desarrollado en este trabajo ha sido bautizado con el nombre AOFIS (siglas de *Adaptive Online Fuzzy Inference System)*. AOFIS es una aproximación dirigida por datos y basada en lógica difusa (en adelante *fuzzy*), para extraer reglas de los datos (obtenidos desde sensores cada cierto tiempo y de las interacciones de los usuarios con los actuadores) que sirven para aprender un FLC *(siglas de Fuzzy Logic Control)*, encargado de convertir información de control lingüística en información de control matemática, que modelará el comportamiento del usuario.

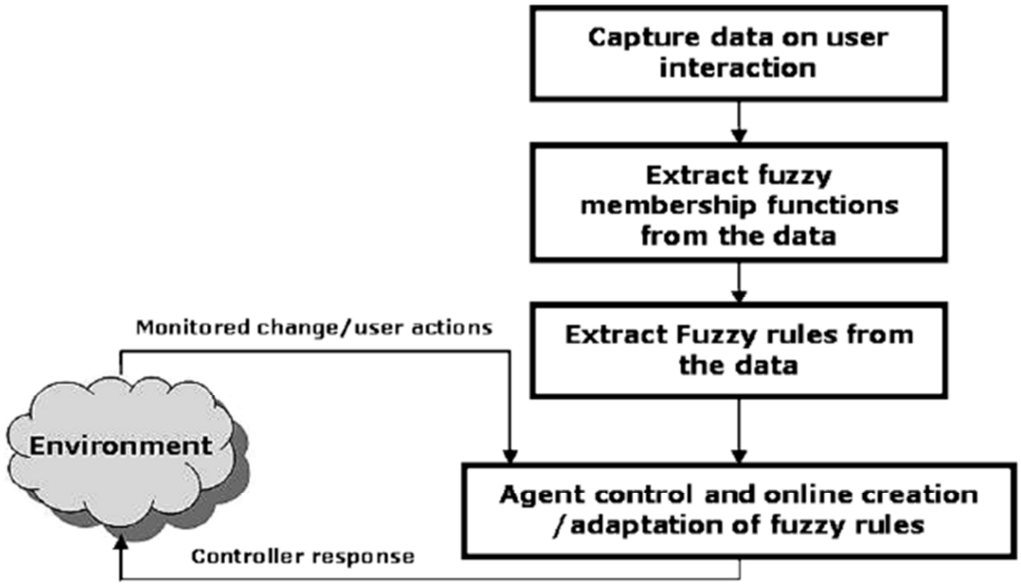

Figura 15 – Diagrama de flujo de las 5 fases de AOFIS

La figura 15 muestra las cinco fases del funcionamiento de AOFIS. Inicialmente, AOFIS monitoriza las interacciones del usuario con el entorno. Cada vez que el usuario cambie la configuración de algún actuador, el agente toma una imagen de las entradas (los datos obtenidos desde los sensores) y de las salidas (los nuevos valores con los que el usuario a modificado el actuador), y va almacenando todos estos datos. Después, en la segunda fase, se cuantifican estos datos en forma de etiquetas lingüísticas (frio, caliente, normal,…) y en la tercera, mediante una versión mejorada del método de Mendel Wang [32], se extraen las reglas *fuzzy* que conforman una tripleta de información, asociando cada par de datos entrada/salida obtenidos en la primera fase con su correspondiente *membership function* (como se ha visto, en forma de etiqueta lingüística) obtenida en la segunda fase.

En la cuarta fase, una vez se tiene toda información, el FLC, encargado de capturar el comportamiento del usuario, ya ha aprendido, y ya puede empezar a controlar el medio ambiente de acuerdo a los deseos del usuario. Dicho control se realizará incidiendo sobre los actuadores (por ejemplo, sobre un actuador que aumente o disminuya el nivel de luz de un lugar) según las reglas aprendidas.

En la última fase, una vez se tienen las reglas que marcan el comportamiento del sistema, el usuario podrá modificar dicho comportamiento, bien para refinarlo o bien porque sus preferencias cambian a lo largo del tiempo. Para este cambio o refinamiento de comportamiento, el usuario simplemente interactúa manualmente con los actuadores (por ejemplo, aumentando el nivel de luz de la habitación). Cuando esto ocurre, el *agente inteligente* modifica sus reglas, o bien añade otras nuevas que se adapten a las nuevas preferencias del usuario.

El proceso de adaptación o creación de nuevas reglas a partir del *feedback* del usuario es el siguiente:

Siempre que el usuario anula alguna respuesta de control del agente accionando por sí mismo manualmente cualquiera de los dispositivos controlados, se registra una imagen del sistema (una "instantánea" de las entradas (los datos obtenidos desde los sensores) y de las salidas (los nuevos valores con los que el usuario a modificado el actuador, en forma de *feedback* implícito), y se la pasa a la rutina de adaptación de las reglas, que se encarga de adaptarlas o crear nuevas.

Esta proceso de adaptación otorga una gran flexibilidad al sistema que, como hemos visto, es capaz de modificar las reglas que el FLC ha aprendido inicialmente (en las primeras etapas de su funcionamiento) para adaptarlo a las preferencias de los usuarios con el paso del tiempo, bien modificando sus reglas o bien creando nuevas (en base al *feedback* del usuario, capturado implícitamente en forma de selecciones/modificaciones que ejerce el usuario en los actuadores).

4.2.1 Evaluación experimental

Finalizamos este apartado sobre la propuesta AOFIS con sus resultados experimentales. Para comprobar su funcionamiento, un usuario permaneció 5 días en un dormitorio inteligente (*iDorm)*, con áreas para realizar diferentes actividades como trabajar, dormir, ver la tele, etc. La *iDorm* estaba compuesta de una gran cantidad de sensores (de nivel de luz interior, de temperatura, de humedad, de nivel de ocupación de la nevera, etc.), dispositivos y procesadores embebidos, conectados entre sí. Era posible controlar directamente los dispositivos del *iDorm* mediante el controlador del *agente inteligente*, accesible de diferentes formas (por ejemplo mediante un móvil, un PC, o incluso desde una nevera inteligente). Ejemplos de lo que se podría controlar son la temperatura de la habitación, el ventilador de la nevera, los focos de luz, las persianas, etc., etc.

El experimento evaluó el ***rendimiento del programa*** en base a lo bien que AOFIS capturó el estado del entorno del *iDorm* y los comportamientos del usuario en este, durante el tiempo que duró la experiencia.

Partiendo del conjunto de reglas *fuzzy* aprendidas en las primeras cuatro etapas de funcionamiento de AOFIS (extraídas de las interacciones del usuario durante los primeros tres días del experimento), el agente inteligente estuvo ejecutando la quinta fase ("*online creation/adaptation de fuzzy rules"*) durante dos días, en los cuales estuvo monitorizando el entorno y las actividades del usuario, y ejecutando las respuestas apropiadas de control basándose en dicho conjunto de reglas aprendidas.

Durante ese tiempo, si era necesario, el usuario en cualquier momento podía sobrescribir y adaptar a sus preferencias las respuestas de control aprendidas por el *agente inteligente*. Por lo tanto, cada vez que el usuario hacia un cambio en los controles, el agente inteligente

recibía la solicitud, ajustaba las reglas aprendidas previamente (o generaba nuevas) y ejecutaba la acción.

El rendimiento de AOFIS puede ser estimado en base al número de correcciones que realiza el usuario ante las respuestas de control del sistema a lo largo del tiempo (en la medida que esto puede reflejar el grado de satisfacción del usuario con las acciones tomadas por el control). La figura 16 muestra un gráfico que representa el número de estas correcciones a lo largo de los dos días de la quinta fase (la fase de adaptación).

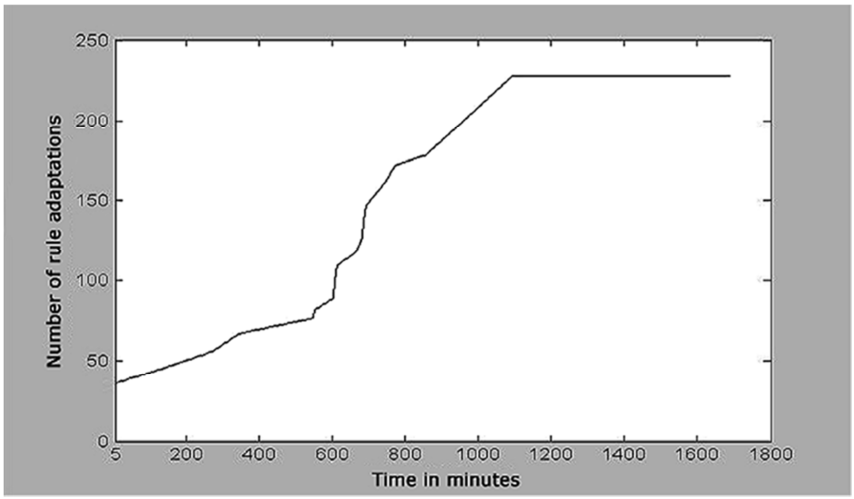

Figura 16 – Gráfico del número de adaptaciones a lo largo del tiempo

En el gráfico se puede observar como la cantidad de adaptaciones se estabiliza a partir de la tarde del segundo día (minuto 1100). Así, el agente fue capaz de aprender de una manera no intrusiva la mayoría de las preferencias del usuario durante dos días, incluyendo comportamientos específicos relacionados con las actividades del usuario, tales como tumbarse en la cama y escuchar música o sentarse en el escritorio para procesar un documento.

4.3 Comparativa de sistemas

Cuando se compara nuestro sistema con las soluciones presentadas en este capítulo, es fácil apreciar ciertas semejanzas en el modo de funcionamiento (a grandes rasgos, todas buscan adaptar la respuesta del sistema a las preferencias del usuario, basándose en el *feedback* de este), así como también diferencias en cuanto a los objetivos particulares de cada una, que desean conseguir y la forma que tienen de hacerlo, los algoritmos que utilizan, su rendimiento, etc. En este apartado trataremos de constatar dichas semejanzas y diferencias, incorporando además una tabla final que muestre fácilmente de manera visual estas similitudes y diferencias.

Como se ha apuntado en el apartado de las semejanzas, se aprecian claramente similitudes en su modo de trabajo:

Las tres son *Free-Model*, es decir, no necesitan un modelo previo que les guíe marcándoles unas normas o reglas de funcionamiento ante determinadas situaciones. Partiendo de un diseño inicial (el *espacio de adaptación de molestia* de los servicios en nuestra solución, o un *perfil de categorías de interés* y un *conjunto de reglas fuzzy* para los casos de *PersonalSearcher* y AOFIS), van adaptándolo según varíen las preferencias y las necesidades de los usuarios a lo largo del tiempo (gracias al *feedback* que se obtiene del usuario).

A un nivel más bajo de detalle, encontramos analogías como la existente entre los conceptos de las *reglas fuzzy* del sistema AOFIS y el concepto de *nivel de molestia* de nuestro sistema, en cuanto a que ambos, dependiendo de como sean adaptados en base a un aprendizaje a raíz del feedback del usuario, provocarán que se interaccione de una forma u otra: Para AOFIS, que un controlador interactúe de determinada forma a la hora de modificar las condiciones del entorno, y para nuestro sistema, que un servicio interactúe de una

manera determinada a la hora de mandar notificaciones al usuario (proactivamente, invisible, etc.).

En el apartado de diferencias, es lógico empezar citando su campo de aplicación, filtrado web para la solución *PersonalSearcher*, control de entornos inteligentes para AOFIS y control de la molestia en la interacción de servicios para nuestra solución. Por añadidura, también serán diferentes sus objetivos. Por ejemplo, el objetivo de AOFIS es que los actuadores del sistema donde se ejecuta actúen de forma acorde con unas reglas *fuzzy*, (aprendidas de los datos capturados en las interacciones de los usuarios al modificar el entorno desde los propios actuadores). En otras palabras, que las condiciones controlables del *entorno inteligente* se accionen de un modo que esté siempre en concordancia con las preferencias del usuario. El objetivo de *PersonalSearcher* es adaptar las sugerencias que brinda a los usuarios en base a sus perfiles, apoyándose en sus navegaciones previas y en el feedback explicito recibido cuando el propio usuario ha puntuado las sugerencias previas.

También difieren en las estructuras utilizadas para representar y almacenar la información. Así, nuestra solución hace uso del concepto de *espacio de adaptación de molestia* (introducido en el apartado 3.1.1) para representar la información de un servicio referente a su nivel de molestia, implementado en forma de mapas ordenados (por la clave que identifica a cada servicio) de matrices bidimensionales. *PersonalSearcher* hace uso del concepto de *perfiles de usuario* para representar información jerarquizada de un usuario en cuanto a sus *categorías de interés*, estando implementado mediante árboles de decisión (específicamente con árboles de clasificación). Por su parte, AOFIS hace uso del concepto de reglas *fuzzy* creadas a partir de los datos de entrada, implementadas en forma de reglas de programación de lógica difusa.

Por último, se va a destacar una diferencia considerable en cuanto al rendimiento de las soluciones y al tiempo que necesitan para aprender los comportamientos de los usuarios. Tanto AOFIS como *PersonalSearcher*, necesitan de una etapa previa de entrenamiento antes de empezar a tener un rendimiento estimable. Como puede apreciarse en sus gráficos de rendimiento de las figura 14 y 16, ambas soluciones pasan por un periodo inicial donde AOFIS necesita de constantes modificaciones por parte del usuario y *PersonalSearcher* tiene mayor feedback negativo que positivo, lo que implica que todavía no se han adaptado a las preferencias del usuario. Nuestro sistema no necesita de esa etapa previa de aprendizaje. Desde su estado inicial, en cuanto se diera el caso de que empieza a recibir *feedback* negativo ante alguna notificación de un servicio, el sistema está capacitado para realizar la adaptación, necesitando solo los pasos necesarios para que el cambio en los valores del espacio de adaptación de molestia (en base al algoritmo *Q-Learning*) provoque a su vez un cambio de nivel de molestia, hasta aprender el nivel correcto (incluyendo unos pequeños saltos de ajuste entre niveles que se dan en este aprendizaje). En otras palabras, nuestro sistema sustituye la gran etapa de aprendizaje inicial de los otros trabajos, por pequeños aprendizajes cada vez que el usuario indique, mediante *feedback* negativo, que desearía cambiar el nivel de un servicio.

Para concluir esta comparativa, en la tabla 1 se muestra un esquema donde se sintetizan de una manera visual las principales características de cada uno de los trabajos relacionados introducidos.

	Filtrado Web	Ambientes Inteligentes	Tesis
Sistema desarrollado	Agente personal (Navegador web) PersonalSearcher	Agente inteligente AOFIS	Sistema para la adaptación del nivel de molestia de los servicios
Objetivo	Mejorar las recomendaciones del sistema, adaptándolas a las preferencias del usuario.	Mejorar el control de un entorno inteligente, adaptándolo a las preferencias del usuario.	Mejorar el nivel de molestia en la interacción servicio-usuario, adaptándola a las preferencias del usuario.
Modo de lograr el objetivo	Aprender el nivel de interés del usuario ante una web, así como aprender el grado de satisfacción del usuario antes las recomendaciones, mediante *feedback*.	Aprender el nivel de satisfacción de un usuario ante determinados valores asignados a los elementos controlables del entorno, mediante *feedback*.	Aprender el nivel de satisfacción del usuario ante el nivel de molestia en la interacción de un servicio, mediante *feedback*.
Tipo de feedback	***feedback* implícito**: Según el tiempo que un usuario pase viendo una web, si la añade en marcadores, etc. ***feedback* explícit**o: Al calificar el usuario las recomendaciones.	***feedback* implícito:** Al modificar el usuario los valores de los elementos controlables manualmente.	**feedback explícito:** Al valorar el usuario el nivel de molestia de la interacción, mediante feedback + o feedback - ***feedback* implícito** Según la reacción del usuario ante dicha notificación.
Free-Model	Si	Si	Si
Algoritmo	Web Document Conceptual Clustering Aprendizaje automático mediante árboles de decisión.	Mendel Wang Aprendizaje difuso (Fuzzy learning)	Q-Learning Aprendizaje por refuerzo (Reinforcement Learning).
Qué obtiene?	Una categorización con los *niveles de interés* del usuario.	Las reglas fuzzy, en las que se basa el sistema para controlar el	Los valores de calidad para combinaciones Q(s,a), con las que se

		entorno inteligente.	conforma el *espacio de adaptación de molestia* de los servicios.
Qué aprende?	Las sugerencias adecuadas para cada búsqueda, en base a las preferencias del usuario.	El nivel adecuado para cada dispositivo controlable de un entorno inteligente, en base a las preferencias del usuario.	El nivel de molestia adecuado para cada servicio, en base a las preferencias del usuario.
Necesaria etapa previa de aprendizaje	Si	Si	No
Conceptos básicos	Perfil de usuario, agente personal.	Agente inteligente, entornos inteligentes.	Nivel de molestia, Espacio de adaptación del nivel de molestia.
Tecnología	Información no disponible	Java	Java, OSGi Services

Tabla 1 – Comparativa de trabajos relacionados

4.4 Conclusiones

El propósito de este capítulo ha sido presentar una breve introducción de trabajos previos relacionados con la propuesta que se desarrolla en esta tesis. Esta presentación ha considerado dos aplicaciones de distinto ámbito: *PersonalSearcher*, en el contexto del filtrado web, y AOFIS, en el de ambientes inteligentes. Nuestra tesis sigue en la línea de ambos trabajos, que a grandes rasgos tienen como objetivo aprender del *feedback* del usuario para conseguir algún un tipo de adaptación por parte del sistema.

Actualmente, existe una investigación muy activa en este tipo de sistemas que pueden adaptarse para satisfacer las preferencias de los usuarios a lo largo del tiempo, en base a su feedback. Sin embargo,

ninguna de estas investigaciones cubría el contexto en el que se desarrolla nuestra propuesta, el de la adaptación del nivel de molestia de los servicios que proliferan en cualquier sistema de hoy en día.

Tal como se explicó en el primer capítulo, ante el creciente aumento de uso de servicios que trae consigo la computación ubicua, era necesario definir una regulación del nivel de molestia de la interacción de estos servicios, de forma que se adaptara a las preferencias del usuario en todo momento. Hasta ahora, esta problemática se había intentado solucionar con otro tipo de métodos como el de la adaptación por reglas de contexto, o con el uso de perfiles de configuración predefinidos, que no aprovechaban las ventajas de una auto-adaptación, ni en cuanto a nivel de comodidad (no es necesario modificar las preferencias servicio a servicio) ni en cuanto a nivel de flexibilidad (no es necesario que existan modelos diseñados previamente que indiquen que nivel de molestia deben tomar los servicios ante determinadas condiciones, lo que provoca que nuestra solución sea mucho más flexible y potente).

Nuestra propuesta resuelve esta problemática adaptando a este ámbito un flujo de trabajo similar al que presentan los trabajos presentados en este capítulo, que consiste en satisfacer las preferencias de los usuarios en base a su feedback, teniendo cada uno de los trabajos su propia filosofía y sus propios métodos para lograrlo, con sus semejanzas y sus diferencias, como se ha demostrado en la comparativa durante este capítulo, destacando que nuestro sistema, al contrario de los demás sistemas presentados en este capítulo, no necesita de una etapa previa de aprendizaje para empezar a ser eficiente.

CAPÍTULO 5

Sistema de auto-adaptación de molestia

Una vez presentada la tesis en si ("el ***qué***", en los apartados 1.1 y 1.4), su propósito ("el ***porqué***", en los apartados 1.2 y 1.3) y su contexto ("el ***dónde***", en el apartado 1.5), así como los conceptos básicos necesarios para poder entender mejor su explicación (en el capítulo 3), a lo largo de este capítulo se va a detallar la forma en que se ha desarrollado el sistema de auto-adaptación de molestia ("el ***cómo***") y se analizará el algoritmo de aprendizaje por refuerzo base de este sistema.

5.1 Diseñando el espacio de adaptación del nivel molestia

Como se ha visto en el apartado 3.1, nuestro trabajo se basa en el *framework* conceptual presentado en el trabajo sobre el ***diseño de interacciones implícitas*** de [6], que define un ***espacio de adaptación del nivel molestia*** (un espacio constituido por los diferentes niveles de molestia por donde puede transitar un servicio), posicionando en un eje en 2D las dimensiones ***iniciativa*** (en la vertical) y ***atención*** (en la horizontal).

Tal como se puede apreciar en la figura 17, nuestro sistema subdivide el eje de iniciativa en dos:

Reactivo: El usuario es quien iniciará la interacción, bien explícitamente o bien implícitamente.

Proactivo: Los servicios iniciarán la interacción, suministrando información no solicitada.

También divide el eje de atención en tres segmentos asociados con las siguientes formas de interacción:

Invisible: El usuario no percibirá la interacción.

Ligeramente apreciable (*slightly-appreciable*): El usuario puede no percibir la interacción, a menos que haga algún esfuerzo.

Plenamente consciente (*completely-aware*): El usuario será plenamente consciente de la interacción, aun incluso cuando esté realizando otras tareas en el sistema.

Figura 17 – Ejemplo de partición de los niveles de iniciativa y atención.

Cada una de las intersecciones de los ejes (conformando una especie de “celda”), se corresponde con un ***nivel de molestia*** que tendrá asociada una configuración de iteración diferente. Así, dependiendo de dónde sitúen los diseñadores un servicio (en qué nivel de molestia), este interaccionara con el usuario de una forma u otra. Por lo tanto, en la etapa de diseño del sistema, los diseñadores, además de decidir qué servicios son necesarios para cada usuario según su perfil, posicionarán cada uno de estos servicios dentro de ese espacio de adaptación de molestia, en un nivel acorde con la forma de

iteración deseada por el usuario, según sus necesidades y preferencias iniciales.

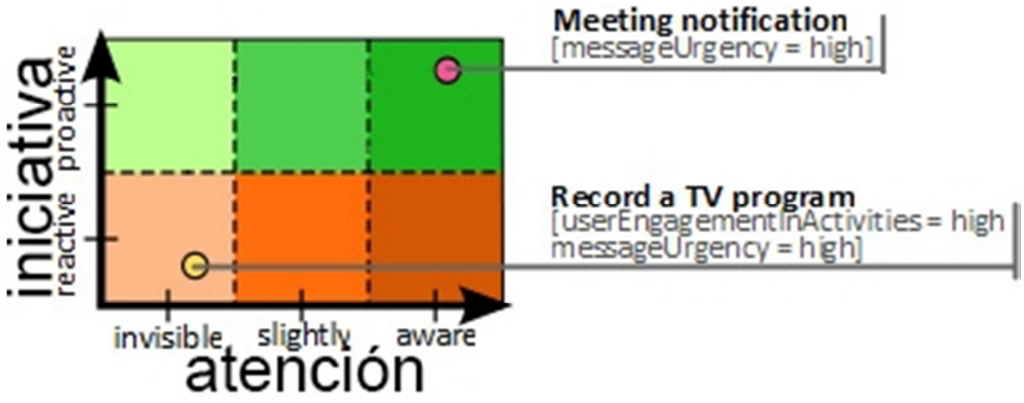

Figura 18 – Ejemplo de diseño de niveles iniciales de molestia.

Por ejemplo, tal como se ilustra en la figura 18, un usuario puede preferir inicialmente que el servicio encargado de grabar programas de televisión, una vez que le ha encomendado una tarea de grabación, no le moleste con ningún tipo de mensaje, que simplemente haga su trabajo en un *segundo plano.* Para este usuario, dicho servicio se posicionaría inicialmente en los niveles de iniciativa *reactiva* y atención *invisible.* Sin embargo, el mismo usuario prefiere que su servicio de notificación de citas le avise automáticamente (por sí mismo) cada vez que tenga información que notificar, y en *primer plano*, de modo que sea totalmente consciente para el usuario, incluso aunque este estuviera haciendo alguna otra tarea en el sistema. Por lo tanto, dicho servicio se posicionaría inicialmente en los niveles de iniciativa *proactiva* y atención *aware.*

5.2 Sistema de auto-adaptación de preferencias

Como se expuso en el apartado 3.1.2, el contexto, las preferencias y las necesidades de los usuarios evolucionan a lo largo del tiempo, siendo necesaria una auto-adaptación del nivel de molestia de cada

servicio (recordemos que asignado en tiempo de diseño) coherente con dichos cambios, de manera que garantice que los servicios interactuarán con el usuario de manera satisfactoria para este.

El sistema de auto-adaptación de preferencias en dispositivos ubicuos del *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València*[2], sistema del que forma parte la implementación de esta tesina, soluciona la adaptación a los cambios de contexto (conseguir por ejemplo que si estoy en una reunión, las notificaciones de un determinado servicio se hagan en modo "*ligeramente apreciable*", pero si estoy a solas el mismo servicio interaccione en modo "*plenamente consciente*") mediante el uso de *reglas de adaptación*, donde se especifica la condición que hace que un servicio cambie de un nivel a otro dependiendo del contexto del usuario.

Sin embargo, el uso de estas reglas de adaptación no resuelve una serie de cuestiones, como por ejemplo:

¿Qué pasa cuando no existen reglas creadas para un nuevo contexto? ¿Qué pasa si para un contexto determinado las reglas de adaptación deberían haber cambiado (porque el contexto ya no el exactamente igual, o porque habría que tener en cuenta nuevos factores)? Es más, el sistema se sigue basando en las preferencias y necesidades presupuestas en tiempo de diseño para cada usuario (según su perfil), para ciertos servicios (asignados también inicialmente al usuario según su perfil) y bajo un determinado contexto. ¿Qué pasa cuando las necesidades y las preferencias del usuario cambien con el tiempo, y esas reglas ya no tasen con ellas? ¿Qué pasa si aparece un nuevo servicio en el sistema?

El sistema debe ser capaz de dar una respuesta a todas estas cuestiones. Debería ser capaz de aprender del usuario, de su

[1] http://www.pros.upv.es
[2] http://www.upv.es

comportamiento interaccionando con el sistema, y saber adaptar el diseño inicial a las nuevas necesidades en tiempo de ejecución, de forma que el usuario siempre esté satisfecho con la interacción.

Con el objetivo de conseguir esta funcionalidad, se ha llevado a cabo la implementación desarrollada en esta tesina. Como estudiaremos en los siguientes apartados, la implementación del sistema se encargará de adaptar los *niveles de molestia* de cualquier servicio (nuevos servicios, o servicios cuya asignación de niveles de molestia en diseño ya no sea satisfactoria, o cuando no se tengan reglas de adaptación para un determinado contexto), aprendiendo del *feedback* de los usuarios ante las interacciones de dicho servicio, mediante un algoritmo de aprendizaje por refuerzo que analizaremos en el punto 5.3.3.

De este modo, si aparecen nuevos servicios en el sistema, o si algún servicio modifica su funcionalidad (puede darse el caso de que al usuario ya no le guste la manera en que dicho servicio interacciona con él, que ya no sea satisfactoria para él la manera en que interacciona la nueva funcionalidad), el propio sistema será capaz de ver como interactúa el usuario con el nuevo servicio o con el servicio que ha modificado su funcionalidad, e ir aprendiendo cuál sería su *nivel de molestia* adecuado.

5.3 La estrategia de aprendizaje por refuerzo

Para llevar a cabo la adaptación del *espacio de adaptación de molestia*, de manera que siempre proporcione una interacción satisfactoria son usuario, el sistema implementado en esta tesis se encargará de controlar y estudiar la interacción (el comportamiento) del usuario con los servicios, para en base al *feedback* de esta interacción ser capaz de aprender el *nivel de molestia* adecuado para cada servicio. Controlará cómo reacciona el usuario ante las notificaciones

de un servicio en cierto *nivel de molestia*, analizando su *feedback* mediante un algoritmo de aprendizaje por refuerzo (RL, *Reinforcement Learning*), para modificar, cuando sea necesario, el nivel de molestia del servicio. De esta manera, dependiendo de su reacción (positiva o negativa), mediante dicho algoritmo RL se deduce respectivamente lo que le gusta y lo que no le gusta, y se van afinando los *niveles de molestia*.

El esquema básico de funcionamiento del sistema, cuyo análisis en profundidad presentaremos en los sucesivos subapartados, es el siguiente:

- Un servicio del sistema manda una notificación al usuario por un *nivel de molestia* determinado.
- Si la interacción que representa dicho *nivel de molestia* le gusta al usuario, este responderá con un *feedback* positivo (recompensa), y en caso contrario responderá con *feedback* negativo (castigo).
    - Si el *feedback* es positivo, significará que le complace la forma en que el servicio está interactuando, y por lo tanto el *nivel de molestia* del servicio no debe ser modificado
    - Si el *feedback* es negativo, el *nivel de molestia* tendrá que ser refinado.
- Después de recibir *feedback* negativo un determinado número de veces (determinado por como el algoritmo RL vaya modificando los valores del *espacio de adaptación de molestia* del servicio), se modificará el *nivel de molestia* de ese servicio, incrementándolo o disminuyéndolo según haya sido el tipo de *feedback* recibido.

5.3.1 Aprendiendo del feedback del usuario

Para conseguir el refinamiento del *nivel de molestia* de los servicios de nuestro sistema, seguimos una estrategia de aprendizaje por refuerzo[13] en la cual nuestro sistema aprende del *feedback* que

manda el usuario en un ***estado de molestia*** (después de haber recibido una notificación por un *nivel de molestia* determinado).

Dependiendo de como haya sido este *feedback*, positivo o negativo, el sistema respectivamente premiará o castigará el actual *estado de molestia*. De esta forma, si una notificación de un servicio mandada en un determinado *estado de molestia*, recibe siempre *feedback* negativo, está indicando que el *nivel de molestia* asignado a ese servicio no es satisfactorio para el usuario, y por tanto el sistema debería cambiarlo.

Por lo tanto, las posibles ***acciones*** que el sistema puede llevar a cabo en cada estado de molestia son:

- ***Mantener***: Mientras reciba *feedback* positivo, mantiene el mismo *nivel de molestia*.
- ***Cambiar***: Si recibe *feedback* negativo, cambia el *nivel de molestia* del servicio para afinar el diseño de su *espacio de adaptación de molestia* satisfactoriamente respecto a las preferencias del usuario.

Para seleccionar la acción que debemos ejecutar en cada estado, ***a cada acción se le asignará un peso***. Inicialmente, la acción mantener del estado al que pertenezca el nivel de molestia inicial de un servicio, se inicializara con un peso mayor que el de la acción cambiar, mientras en los demás estados se hará lo contrario. Después, se define una política de comportamiento (***behaviour policy***) con el objetivo de seleccionar la acción adecuada para un estado determinado, en base al peso de sus acciones (en nuestro sistema, se seleccionará siempre la acción de mayor peso).

El proceso de selección de la acción a ejecutar será el siguiente:

Inicialmente, mientras se vaya recibiendo *feedback* positivo, el sistema irá incrementando el peso de la acción mantener (en base a unos ratios obtenidos mediante las fórmulas del algoritmo RL), por lo que cuando

ejecutemos la política de comportamiento, esta política seleccionará a la acción mantener y se mantendrá el mismo nivel de molestia.

Si empieza a recibirse *feedback* negativo, también en base al mismo algoritmo RL, irá disminuyendo el peso de la acción mantener, y llegará un momento que será inferior al peso de la acción cambiar, por lo que la política de comportamiento seleccionará la acción cambiar, y procederá a adaptar el diseño inicial del *espacio de adaptación de molestia*, ajustando el *nivel de molestia* del servicio.

5.3.2 Obteniendo el feedback

Como hemos comentado anteriormente, será el usuario el que en base a su *feedback* podrá modificar el diseño del *espacio de adaptación de molestia*, en base a sus recompensas (*feedback* positivo) o sus castigos (*feedback* negativo).

Por lo tanto, el sistema debe ofrecer la posibilidad de introducir ese *feedback*, ofreciéndolo de dos formas distintas:

- ***Explícitamente***: El usuario podrá introducir el feedback mediante un interfaz grafico que permita la selección explicita de recompensa (feedback +) o castigo (feedback -). La figura 19 ilustra con un ejemplo este tipo de interfaz.

Figura 19 – Ejemplo de interfaz para *feedback* explícito.

- ***Implícitamente***: El sistema obtendrá el *feedback* implícito del usuario observando su comportamiento. Dado un comportamiento del usuario frente a una notificación, se obtendrá su *feedback* mediante una función semántica que convierta ese comportamiento en una recompensa o un castigo. Por ejemplo, si el usuario quita la voz nada más recibir una notificación acústica, significa que le ha molestado, y por tanto habría que disminuir el *nivel de molestia* del servicio que mandó la notificación, mientras que si se le envían notificaciones y no las mira, puede ser que no esté percibiéndolas, por lo que habría que incrementar el nivel de atención.

El trabajo desarrollado en esta tesis simula la obtención del *feedback* implícito en base "verbos" de acciones posibles ante una notificación ("vista", "no vista", "silenciada", "respondida", "no respondida",…), a los que una función semántica se encargará de convertir en un *feedback* positivo o en uno negativo, indicando en este caso si se debe disminuir o incrementar el nivel de molestia del servicio.

Por ejemplo, si el usuario pone su dispositivo en silencio nada más recibir una notificación sonora ("silenciada"), significa que debe hacerse un decremento en el nivel de atención, mientras que si el usuario no percibe la notificación ("no vista"), significa que debe incrementarse el nivel de atención.

5.3.3 Algoritmo de aprendizaje por refuerzo

Una vez visto como se obtiene el *feedback* del usuario, en este apartado se presentará el algoritmo utilizado por nuestra solución, basado en el aprendizaje por refuerzo mediante el método de ***Q-Learning*** [17]. Este método tiene una función Q que calcula el *valor de calidad* de elegir una *acción* "a" en el *estado de molestia* "s" (denotada por **Q(s,a)**).

Antes de que el aprendizaje haya empezado, Q devuelve un valor fijo, elegido por el diseñador. En nuestro sistema, para un determinado servicio, Q estará inicializada de forma que devuelva un valor mayor a la combinación Q(s,a) cuando s se corresponda con el estado correspondiente al *nivel de molestia* inicial del servicio y "a" se corresponda con la acción mantener (posteriormente, en el apartado 7.2 sobre implementación, se detallará a bajo nivel todo el proceso).

Una vez el sistema de aprendizaje ha comenzado, cada vez que se recibe un *feedback*, nuevos valores de Q(s,a) son estimados con la siguiente fórmula:

$$Q(s, a) \Leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a))$$

A continuación se explica el significado de cada uno de los miembros de la derecha de la fórmula:

- Q(s,a) es el ***valor de calidad*** Q(s,a) en el estado anterior.

- α es la ***tasa de aprendizaje***, un parámetro que pondera las actualizaciones con respecto a los valores anteriores (determina en qué medida la nueva información reemplaza a la antigua).

  Un valor de 0 provocará que el sistema no aprenda nada (mirando la fórmula, se ve que si α=0, al multiplicar por 0 volvería a dar el mismo valor Q(s,a)), mientras que un valor de 1 implicaría que el sistema solo considere el *feedback* más reciente (si α =1, al multiplicar por 1, vemos que a Q(s,a) se anularía con -Q(s,a)).

- γ es un ***parámetro de descuento*** entre 0 y 1 que expresa la importancia de las recompensas futuras.

En nuestro sistema, los valores asignados a α *y* γ *serán:*
α *=0.5;* γ *=0.99;*

- Q(s',a') es el valor futuro esperado (el valor de calidad que tendrá la acción de más peso en el siguiente estado).

- r es el *feedback* observado después de llevar a cabo la acción "a" en el estado s. Este *feedback* podrá ser una recompensa o un castigo.

En nuestro sistema, los valores asignados a r *serán:*
r *= +1, en caso de recompensa (feedback positivo)*
r *= -1, en caso de castigo (feedback negativo)*

Una vez introducida la fórmula para calcular la calidad de elegir una acción "a" en un estado s, pasamos a presentar en la tabla número 2 el algoritmo Q-Learning:

ALGORITMO RL

Requiere:	*tasa de aprendizaje* (α),*parámetro de descuento* (γ), *valor umbral de Q(s,a)* (Δ)
Paso 1	Inicializar s con el estado de molestia inicial, dando en dicho estado un peso mayor a la acción "*mantener*", mientras al resto de estados se le dará mayor peso a la acción "*cambiar*".
Paso 2	**repetir**
Paso 3	**cuando** haya un notificación N_t que enviar:
Paso 4	Elegir la acción "a" en base a la política de comportamiento
Paso 5	Llevar a cabo dicha acción "a"
Paso 6	Mandar N_t por el nuevo estado s'
Paso 7	Observar el *feedback* r ante dicha notificación
Paso 8	Actualizar V(s,a)
Paso 9	s:=s'
Paso 10	**si** r es negativo **entonces**:
Paso 11	Guardar el modo (incrementar/disminuir) del *feedback*
Paso 12	**fin si**
Paso 13	**hasta** fin de proceso de aprendizaje

Tabla 2 – Algoritmo *Q-Learning* modificado

El funcionamiento del algoritmo, cuyos diagramas de secuencia y de actividades se presentarán respectivamente en los apartados 6.6 y 6.7, es el siguiente:

Como tenemos un *espacio de adaptación de molestia* diseñado “a-priori”, asociaremos s al *nivel de molestia* inicial del servicio, dando en dicho estado un peso mayor a la acción “mantener” frente a la acción “cambiar”, mientras que en el resto de estados se hará a la inversa. Esto proporcionará un comportamiento inicial consistente.

Cuando un servicio quiera enviar una notificación, el sistema elegirá una acción del estado s (mantener o cambiar) de acuerdo a una política de comportamiento (en nuestro sistema, de ambas acciones se elegirá siempre a la que tenga un peso mayor). Entonces, dicha acción se llevará a cabo y la notificación será mandada por el nuevo estado de molestia s' (siendo s' otra vez el mismo estado s si la acción elegida fue "mantener", u otro estado diferente si fue elegida la acción "cambiar").

Si la acción a ejecutar es “cambiar”, el sistema ajustará el estado de molestia del servicio, disminuyéndolo o incrementándolo en base al *feedback* recibido.

Cada vez que se recibe *feedback* ante una notificación, el sistema va actualizando los pesos de las acciones para el estado por donde se mandó la notificación, en base a la fórmula de Q(s,a) presentada anteriormente.

El algoritmo RL de la tabla 2, muestra una versión modificada del algoritmo de aprendizaje con el método ***Q-Learning***. Como se puede apreciar en dicho algoritmo, en lugar de actualizar Q(s,a) se actualiza la función V(s,a), una modificación propia de nuestro sistema, que describiremos en el siguiente subapartado.

5.3.3.1 Función V

Pensando en mejorar la respuesta del algoritmo en ambientes ubicuos, nuestro sistema ha introducido una nueva *función V* para establecer un límite al valor que la calidad Q(s,a) puede tomar, con intención de acelerar el aprendizaje y la obtención de una adaptación con pocos ensayos cuando el usuario cambia sus preferencias.

Esta nueva función es la siguiente:

$$V(s,a) = \begin{cases} -\Delta & \text{si} \quad Q(s,a) \leq -\Delta \\ Q(s,a) & \text{si} \quad -\Delta < Q(s,a) < \Delta \\ \Delta & \text{si} \quad Q(s,a) \geq \Delta \end{cases}$$

Donde Δ es el valor umbral que deseamos que la calidad pueda alcanzar. Este valor debe estar acorde con los valores de la *tasa de aprendizaje* (α) y del *feedback* (r), ya que define aproximadamente el número de iteraciones necesarias para aprender un nuevo comportamiento.

Si Δ es elevado, el sistema será menos sensible ante feedback anómalo, pero por el contrario necesitará más tiempo en aprender. Si Δ es bajo, el sistema necesitará menos tiempo en aprender un comportamiento, pero más sensible ante el feedback anómalo.

Por lo tanto, la elección del valor de Δ puede ser diferente para cada usuario, según se quiera priorizar un criterio u otro.

5.4 Conclusiones

En el transcurso de este capítulo se ha presentado el desarrollo del ***sistema de auto-adaptación de molestia***, centro del trabajo de esta tesis. Se ha expuesto como surge su realización a raíz de una necesidad en otra investigación previa, como se ha diseñado en espacio de adaptación en que se basa, y se ha analizado la estrategia

que utiliza para lograr sus objetivos, la estrategia del ***aprendizaje por refuerzo***.

Se han mostrado las posibilidades existentes a la hora de obtener el ***feedback*** de los usuarios, explícita e implícitamente, para posteriormente explicar cómo se emplea dicho *feedback* en el proceso de aprendizaje del nivel adecuado de molestia de los servicios.

Finalmente, se ha visto como llevar a cabo dicho aprendizaje, mediante la comentada estrategia del aprendizaje por refuerzo, presentando el algoritmo de aprendizaje por refuerzo bajo el método ***Q-Learning***, cimiento de todo el trabajo desarrollado en la tesis, así como la implementación de una mejora sobre este método a base de la incorporación de una función (***función v***), que impone un umbral máximo a los valores alcanzables por el algoritmo para el peso de las acciones, de manera que se puede acelerar el aprendizaje y obtener una adaptación con un menor número de ensayos.

CAPÍTULO 6

Análisis del sistema

Una vez introducidos los conceptos básicos, necesarios para la mejor comprensión del trabajo realizado en esta tesis, así como la teoría del sistema a desarrollar, se procederá a mostrar el análisis del sistema, presentando las características requeridas para el proyecto a desarrollar en esta tesis.

Como se detalló en el apartado 1.5, el sistema a desarrollar en esta tesis surge de una proposición de trabajo fin de máster del *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València.* Esta propuesta consistía en desarrollar una implementación de su sistema de auto-adaptación de preferencias analizado en el apartado 5.2.

Esta implementación tenía una serie de requisitos a cumplir en cuanto a tecnologías y funcionalidad, que pasamos a detallar en los siguientes subapartados.

6.1 Requisitos del sistema

De cara a alcanzar las expectativas de la tesis, el sistema debía cumplir una serie de características u objetivos:

1) El sistema debía ser capaz, en tiempo de ejecución, de adaptar el *espacio de adaptación de molestia* de los servicios del sistema a las preferencias de los usuarios, en base al *feedback* de estos últimos.

[1] http://www.pros.upv.es

2) Para lograrlo, el sistema debía implementar el algoritmo de aprendizaje por refuerzo *Q-Learning (*concretamente la versión mejorada que se presentado en el apartado 5.3.3.1.)

3) El sistema debía ser capaz de trabajar con los servicios importados desde un modelo EMF, cuyo metamodelo mostramos en la figura 20.

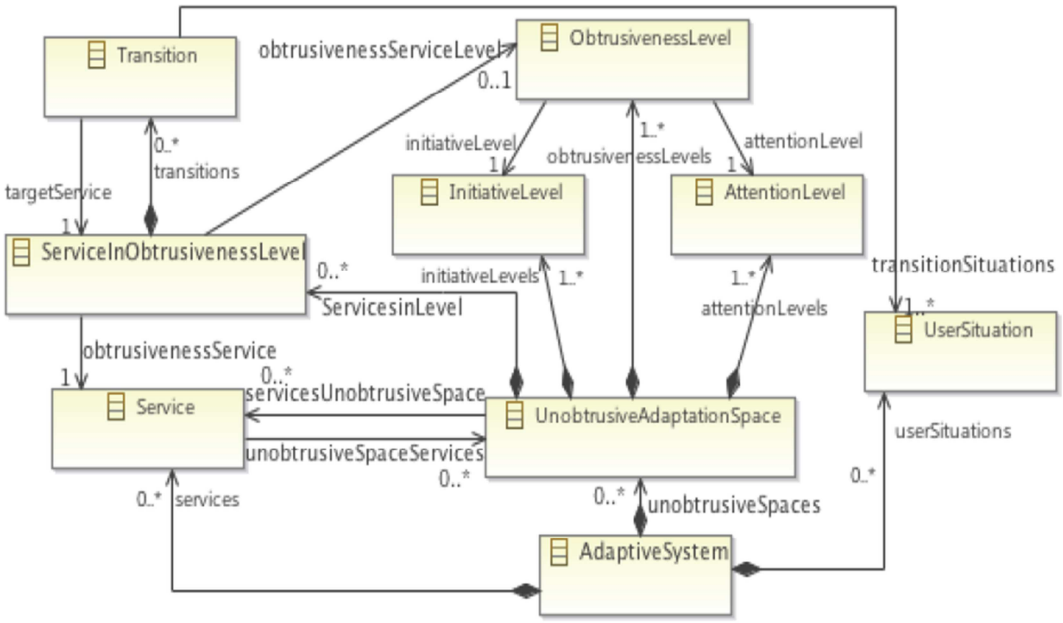

Figura 20 – Metamodelo del modelo de molestia a cargar por el sistema.

4) El sistema ofrecería la posibilidad de crear servicios nuevos, con los que poder trabajar de igual manera que con los servicios cargados.

5) Los servicios podrían tener un *espacio de adaptación de molestia* de diferente tamaño y estructura.

6) Los usuarios podrían elegir el servicio al que deseasen ajustar el nivel de molestia, pudiendo cambiar de servicio en cualquier momento.

7) Los usuarios podrían simular el envío de una notificación por parte del servicio seleccionado.

8) Los usuarios, ante una notificación enviada por un servicio, podrían devolver un *feedback* al sistema.

9) El *feedback* podría ser de dos tipos: Explícito e Implícito.

10) El *feedback* explícito podría ser positivo o negativo.

11) El *feedback* negativo debía ir asociado de un modo: Modo incrementar o modo disminuir el *nivel de atención*.

12) El *feedback* implícito consistiría en la simulación de una acción mediante la elección de unos verbos, que posteriormente una función debería traducir por *feedback* positivo o negativo.

13) El sistema debía almacenar el *feedback* negativo hasta que fuese utilizado para calcular la frecuencia con que se había enviado *feedback* negativo con intención de incrementar el nivel de molestia y la frecuencia con que se había enviado *feedback* negativo con intención de disminuir el nivel de molestia, para poder decidir si incrementar o disminuir el nivel en caso de necesidad de ajuste.

14) El sistema debía proporcionar una interfaz gráfica de usuario que diera acceso a toda la funcionalidad del sistema.

6.2 Actores

En el sistema a implementar solo existirá un único actor, el usuario del sistema, que pasamos a describir en la tabla 3.

ACTOR	Usuario
Descripción	Será el usuario final del sistema, el que ajustará el nivel de molestia de los servicios en base a sus preferencias.
Tipo	Principal
Comentarios	Será el usuario estándar que usará el sistema para seleccionar los servicios, hacerles enviar

	notificaciones y finalmente devolver su feedback.

Tabla 3 – Tabla del actor usuario

6.3 Casos de uso

En este subapartado se mostrarán los casos de uso del sistema, tanto sus modelos como sus especificaciones.

6.3.1 Modelado conceptual

La Figura 21 muestra, a nivel conceptual, el modelo de casos de uso del sistema. En dicha figura se muestran al actor "usuario" de nuestro sistema, así como el conjunto de funcionalidades o acciones que puede realizar, organizado por temática o contexto.

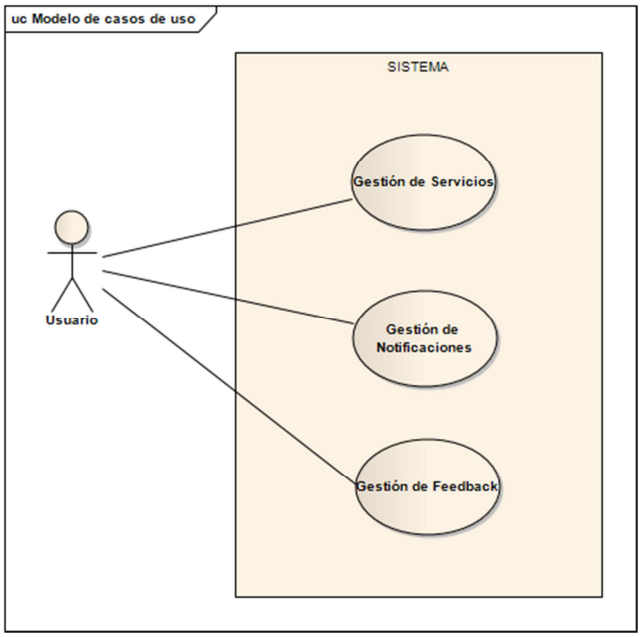

Figura 21 – Modelado conceptual de los casos de uso

En nuestro sistema, la relación entre actores y casos de uso, al existir un único actor, no se rige por cuestiones de privacidad o privilegios, sino que el usuario tiene acceso a toda la funcionalidad del sistema, estando relacionado con todos los casos de uso conceptuales.

6.3.2 Modelado de casos de uso

Los casos de uso especifican las distintas funcionalidades que ofrece el sistema, quién las realiza y para qué. En la figura 22 se presenta el diagrama de casos de uso que tiene relación directa con el actor “usuario” sin considerar al sistema.

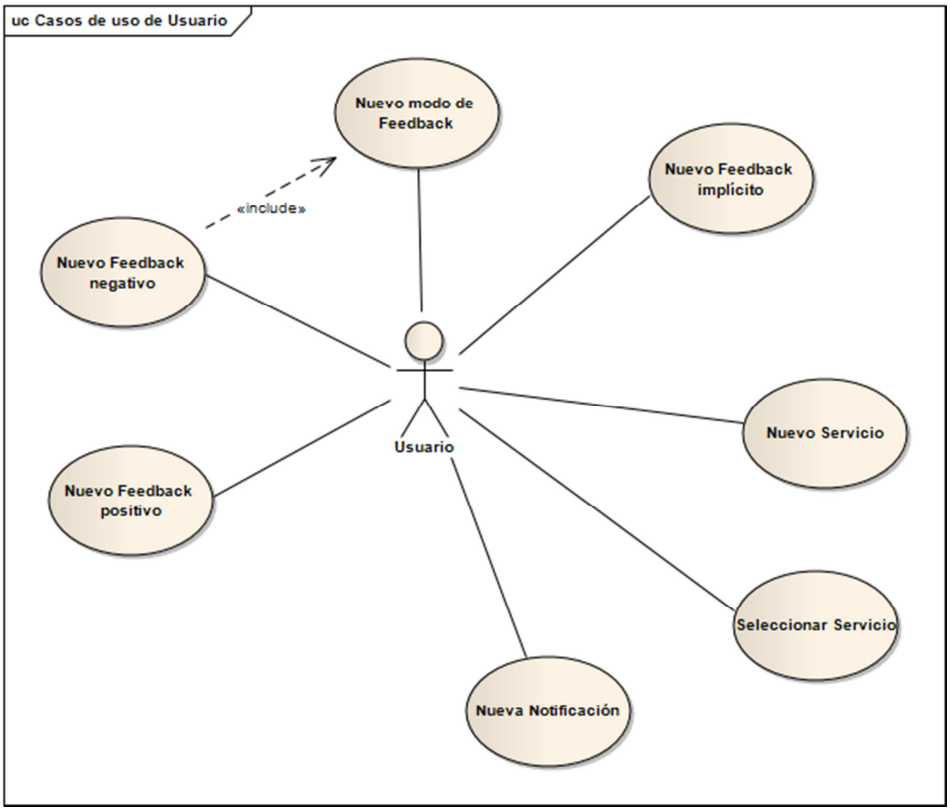

Figura 22 – Casos de uso de del Actor Usuario

6.3.3 Especificación de casos de uso

Nombre	Nuevo servicio
Identificador	001
Actores	Usuario
Resumen	El usuario creará un nuevo servicio en el sistema, con un nivel de molestia inicial, al que podrá posteriormente ajustar el nivel de molestia si cambian sus necesidades.
Dependencias	
Precondición	No debe existir un servicio con el mismo identificador
Postcondición	El nuevo servicio quedará registrado en el sistema
Curso normal	1º El usuario iniciará la aplicación 2º El usuario, dentro de la zona servicios, podrá introducir los datos del nuevo servicio, rellenando una serie de campos 3º Una vez rellenados todos los campos y pulsado el botón de nuevo servicio, el servicio será creado.
Curso alternativo	3º Si hay algún problema con alguno de los datos o bien con

	la creación del nuevo servicio, se notificará con un mensaje de error.
Comentarios	

Nombre	Seleccionar servicio
Identificador	002
Actores	Usuario
Resumen	El usuario seleccionará un servicio, de entre los disponibles en el sistema, para ajustarle su nivel de molestia.
Dependencias	
Precondición	Debe existir algún servicio en el sistema
Postcondición	A partir de su selección, el sistema actuará sobre el servicio seleccionado.
Curso normal	1º El usuario iniciará la aplicación 2º El usuario, dentro de la zona de servicios, podrá seleccionar un servicio de los existentes en el sistema.
Curso alternativo	2º Si hay algún problema con la selección del servicio, se notificará con un mensaje de error.
Comentarios	Al seleccionar un servicio, el usuario automáticamente verá el contenido del espacio de adaptación de molestia de dicho servicio.

Nombre	Nueva notificación
Identificador	003
Actores	Usuario
Resumen	El usuario hará que el servicio seleccionado mande una notificación.
Dependencias	
Precondición	Debe existir un servicio seleccionado
Postcondición	A partir de su selección, el sistema actuará sobre el servicio seleccionado.
Curso normal	1º El usuario iniciará la aplicación 2º El usuario, dentro de la zona de notificaciones, podrá seleccionar mandar una notificación. 3º Se obtendrá un mensaje que muestre con qué nivel de molestia ha sido mandada dicha notificación.
Curso alternativo	3º Si hay algún problema con la el envío de la notificación, se notificará con un mensaje de error.
Comentarios	

Nombre	Nuevo feedback positivo
Identificador	004
Actores	Usuario
Resumen	El usuario devolverá feedback positivo.
Dependencias	
Precondición	Debe haberse recibido una notificación de un servicio.
Postcondición	Se actualizarán los valores del nivel de molestia del servicio que mandó la notificación para la cual se está devolviendo el feedback.
Curso normal	1º El servicio que se encuentra actualmente seleccionado en el sistema, manda una notificación. 2º El usuario, dentro de la zona de feedback explícito, podrá

	seleccionar mandar feedback positivo. 3º Se actualizará el espacio de adaptación de molestia del servicio.
Curso alternativo	3º Si hay algún problema con la el envío del feedback, se notificará con un mensaje de error.
Comentarios	El feedback positivo puede darse de dos formas, bien explícitamente o bien implícitamente (seleccionando una acción)

Nombre	Nuevo feedback negativo
Identificador	005
Actores	Usuario
Resumen	El usuario devolverá feedback negativo.
Dependencias	<<Include>> a 006
Precondición	Debe haberse recibido una notificación de un servicio.
Postcondición	Se actualizarán los valores del nivel de molestia del servicio que mandó la notificación para la cual se está devolviendo el feedback.
Curso normal	1º El servicio que se encuentra actualmente seleccionado en el sistema, manda una notificación. 2º El usuario, dentro de la zona de feedback explícito, podrá seleccionar mandar feedback negativo. 3º El usuario indicará el modo del feedback (incrementar/disminuir) 4º Se actualizará el espacio de adaptación de molestia del servicio.
Curso alternativo	4º Si hay algún problema con el envío del feedback, se notificará con un mensaje de error.
Comentarios	El feedback positivo puede darse de dos formas, bien explícitamente o bien implícitamente (seleccionando una acción)

Nombre	Nuevo tipo de feedback
Identificador	006
Actores	Usuario
Resumen	El usuario indicara el tipo del feedback negativo.
Dependencias	
Precondición	
Postcondición	Se seleccionará el modo del feedback negativo.
Curso normal	1º El usuario se encuentra actualmente dentro de la zona de feedback explícito, queriendo mandar feedback negativo. 2º El usuario, dentro de la zona de feedback explícito negativo, podrá seleccionar el tipo de feedback negativo, entre incrementar/disminuir. 3º El modo del feedback negativo quedará seleccionado.
Curso alternativo	3º Si hay algún problema con la selección del modo, se notificará con un mensaje de error.
Comentarios	El tipo del feedback negativo consiste en indicarle al sistema que le gustaría incrementar el nivel de molestia del servicio para el cual ha mandado el feedback negativo (si selecciona el tipo incrementar) o bien lo contrario (si selecciona el tipo disminuir)

Nombre	Nuevo feedback implícito
Identificador	007
Actores	Usuario
Resumen	El usuario indicara el modo del feedback implícito, seleccionando una acción respuesta a la notificación.
Dependencias	
Precondición	
Postcondición	Se actualizarán los valores del nivel de molestia del servicio que mandó la notificación para la cual se está devolviendo el feedback.
Curso normal	1º El servicio que se encuentra actualmente seleccionado en el sistema, manda una notificación. 2º El usuario, dentro de la zona de feedback implícito, podrá seleccionar mandar una acción como respuesta a la notificación. 3º El sistema convertirá la acción seleccionada en feedback positivo o negativo, según una función semántica.
Curso alternativo	3º Si hay algún problema con el envío del feedback, se notificará con un mensaje de error.
Comentarios	

6.4 Modelado conceptual del sistema

El modelo conceptual de la figura 23 determina cual es el dominio de nuestra aplicación, reflejando la organización e información de todo el sistema. Cada una de las clases de este diagrama representará conceptualmente una o varias clases correspondientes con el diagrama de clases obtenido en la fase de diseño.

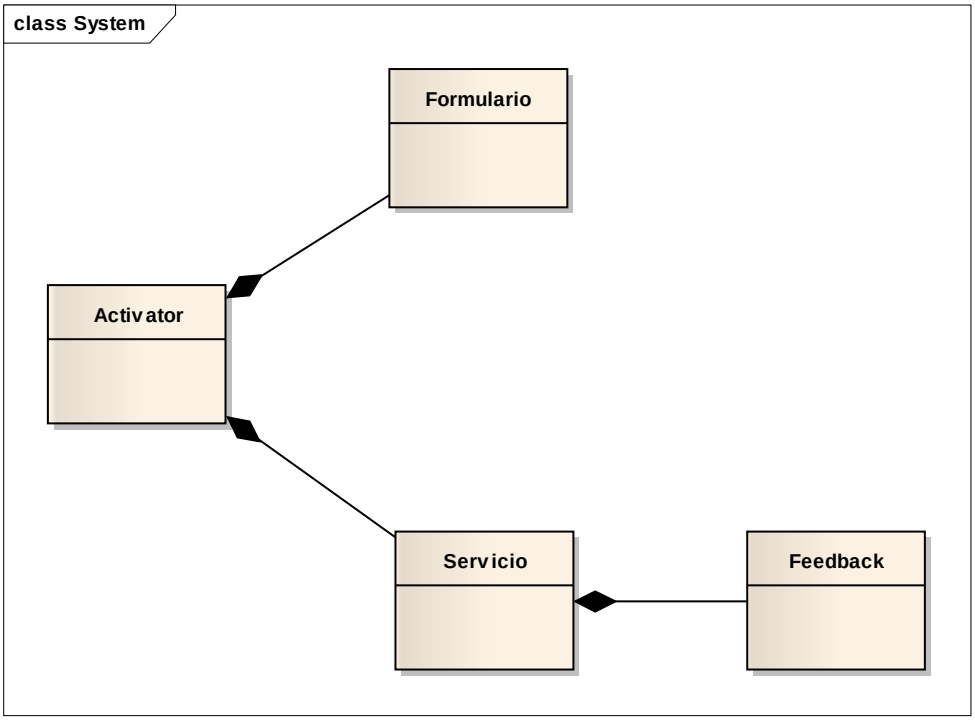

Figura 23 – Modelado conceptual del sistema

6.5 Modelo conceptual de la colección de Feedback

La figura 24 muestra el modelo conceptual de la colección del feedback de usuario. Una de las posibles implementaciones concretas para esta representación mediante listas se podrá ver en el apartado de diseño.

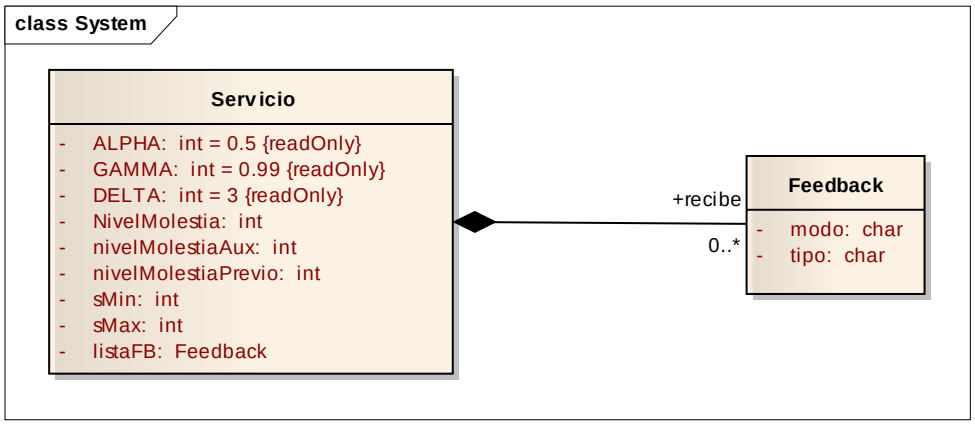

Figura 24 – Modelo conceptual de la colección de Feedback

Cada servicio, cuando interaccione con el usuario mandándole notificaciones, empezará a recibir feedback del usuario, indicándole si le ha gustado o no el nivel de molestia con que ha realizado la interacción. A cada *feedback* se le asigna un modo, que representa si el *feedback* ha sido una recompensa (*feedback* positivo) o un castigo (*feedback* negativo). También, para el caso del *feedback* negativo, se les asigna un tipo, que simboliza que tipo de adaptación de *nivel de molestia* desearían, si del tipo incrementar o del tipo disminuir.

Hay que resaltar el detalle de que cuando se reciba *feedback* negativo, que implica que el usuario no está satisfecho con el nivel con que se ha producido una interacción, unas veces puede estar indicando que el usuario desearía disminuir el nivel de molestia del servicio, mientras otras que desea aumentarlo. Nuestro desarrollo solucionará este detalle almacenando el tipo de adaptación (incrementar/disminuir) de nivel de molestia que el usuario desearía cuando devuelve cada *feedback* negativo, de manera que si se llega a un momento en que el sistema decide que se debe ajustar el *nivel de molestia* del servicio, calculamos la frecuencia de cada tipo de *feedback* almacenado (recordemos que son tipo incrementar o tipo disminuir) y disminuiremos o incrementaremos el nivel de molestia de acuerdo al modo más frecuente (al tipo que tenga la media más alta).

6.6 Diagrama de secuencia

El diagrama de la figura 25 muestra la secuencia de eventos que se sucederán en el sistema, junto a los actores y elementos del sistema que intervienen en ella, desde que se inicia la aplicación hasta que usuario devuelve su feedback.

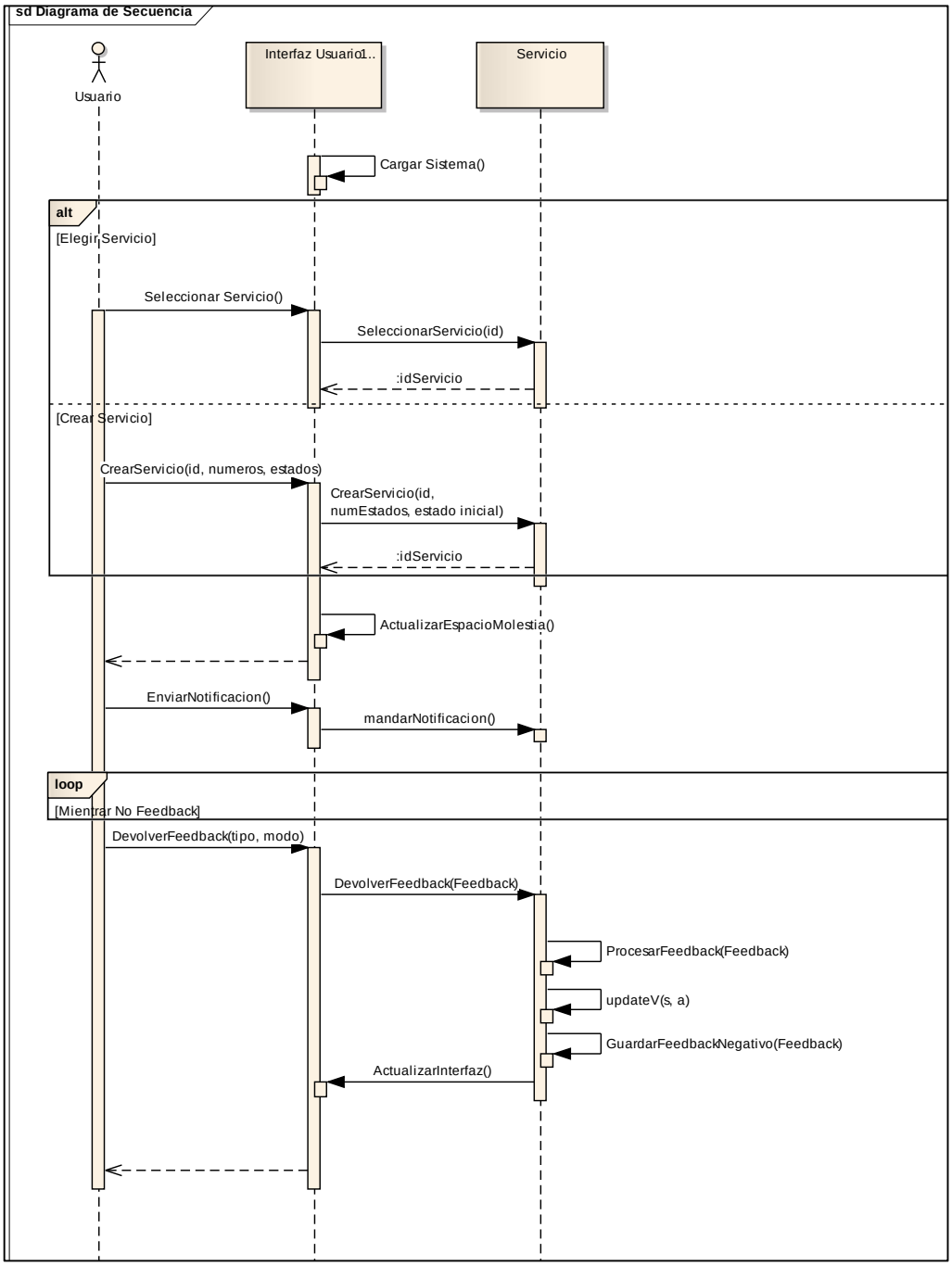

Figura 25 – Diagrama de Secuencia de notificación y devolución de feedback

El diagrama se corresponde con los casos de usos vistos en el apartado 6.3.1, representando un escenario completo de ejecución del sistema, donde primero se cargará el sistema y después el usuario seleccionará un servicio o creará uno nuevo.

Entonces, el sistema esperará hasta que exista una notificación que enviar. Una vez que un usuario hace que el servicio que está seleccionado en el sistema envíe una notificación, el mismo usuario devuelve un *feedback* ante dicha notificación, para que el sistema lo procese y actualice el nivel de molestia del servicio (que lo mantenga el mismo nivel o bien que lo modifique, de acuerdo al *feedback*).

Comparando esta secuencia con el propio algoritmo de aprendizaje por refuerzo presentado en 5.3.3, se puede apreciar un paralelismo total entre dicho algoritmo y la segunda mitad de este diagrama de secuencia.

6.7 Diagrama de actividades

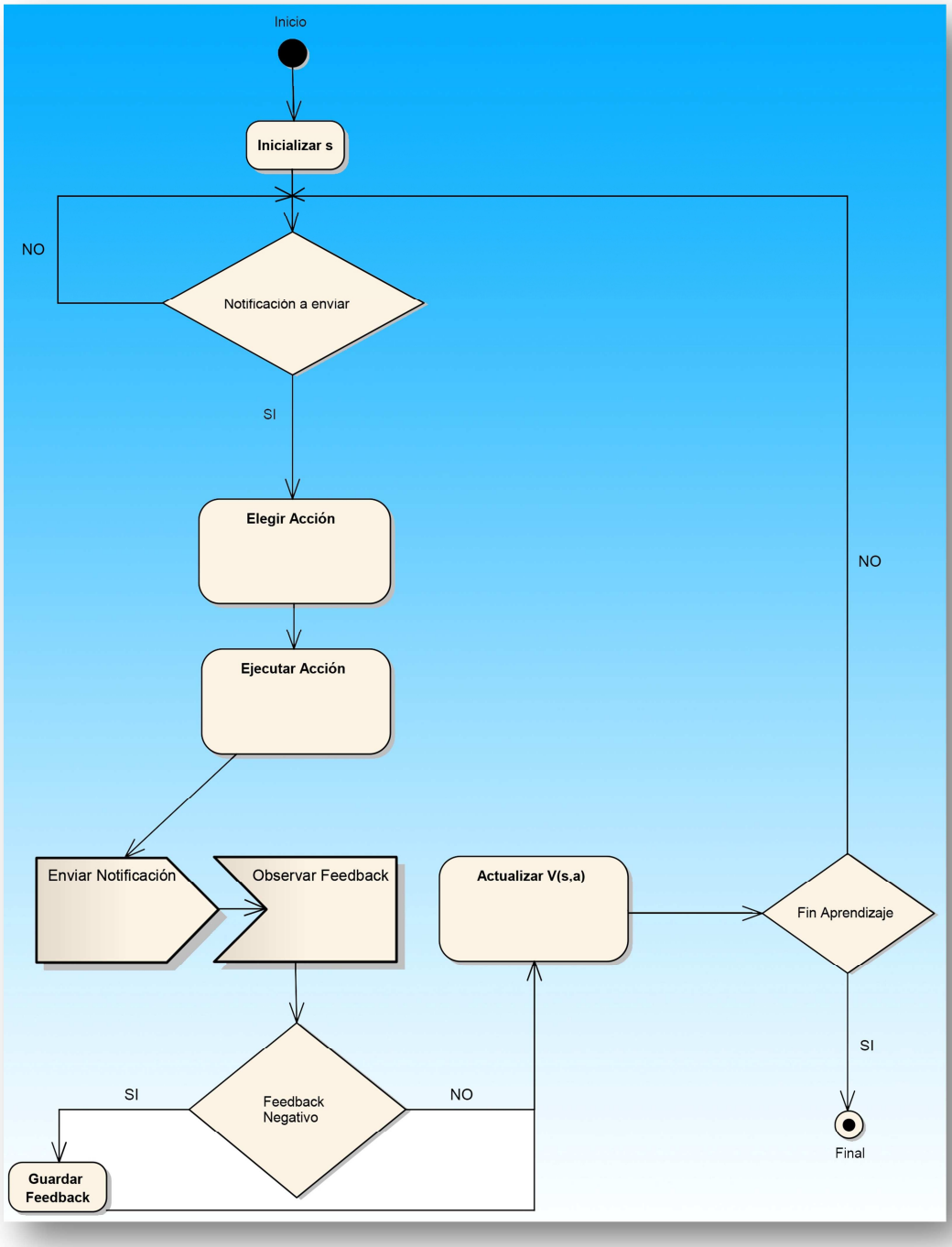

Figura 26 – Diagrama de Actividades

El diagrama de Actividad de la Figura 26 es la representación gráfica de los pasos del algoritmo a implementar, visto en el apartado 5.3.3, pudiendo comprobar fácilmente como existe una correspondencia entre cada actividad y cada uno de los pasos del algoritmo.

CAPÍTULO 7

Diseño e implementación del sistema

Una vez se ha analizado el problema, se mostrará la estrategia de alto nivel que se ha seguido para, partiendo del modelo conceptual del análisis, construir la solución software de nuestro sistema. Se presentará la arquitectura del sistema, el diseño de objetos y las estructuras de datos elegidas, y partiendo del diagrama de secuencia, se estudiará cómo se han diseñado los mensajes entre los distintos elementos del sistema, mostrando un listado con los principales métodos implementados en nuestro sistema.

7.1 Arquitectura del sistema

Para descomponer el desarrollo de nuestro sistema en subsistemas más pequeños y poder llevarlo a cabo en varios niveles, se optado por seguir una estrategia multicapa, concretamente la de la arquitectura de tres capas [38], que muestra la figura 27.

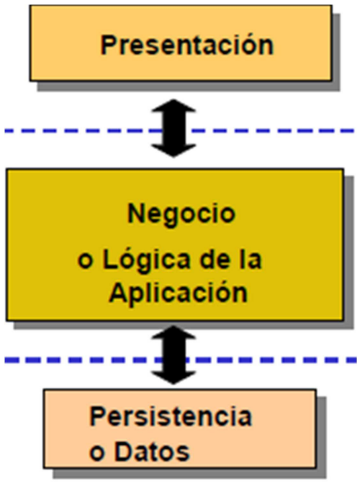

Figura 27 – Arquitectura de 3 capas

En esta arquitectura se aprecian los siguientes niveles:

1) **Nivel de Presentación**: Donde se proporcionará a los usuarios la posibilidad de interaccionar con el sistema a través de un interfaz de usuario, ofreciéndoles una forma de acceder y controlar a los datos y visualizar los resultados.
2) **Nivel de lógica o de negocio**: Donde se implementará completamente el comportamiento de las clases del dominio, especificadas en la fase de modelado conceptual, para proporcionar la funcionalidad del sistema.
3) **Nivel de Persistencia o de datos**: Donde se almacenan los datos y se proporcionan una serie de servicios que permiten a los objetos acceder a estos.

Este tipo de arquitectura ofrece la ventaja de una subdivisión del trabajo, que se hace más sencillo al ser desarrollado por partes. También aporta la ventaja asociada a la independencia de los niveles, por la que las modificaciones se realizarán más fácilmente sin que un cambio en una capa afecte al resto.

7.1.1 Capa de presentación

En los sucesivos subapartados se presentará el interfaz de usuario, mostrando como se presentan los datos hacia el usuario y como el usuario puede lograr interactuar con ellos.

7.1.1.1 Interfaz de usuario

En la figura 28 se observa la interfaz gráfica del sistema desarrollado en la tesis, el cual brindará acceso a toda su funcionalidad. Este interfaz se ha diseñado haciendo uso del editor de interfaces gráficas *WindowBuilder* presentado en el punto 2.1.2

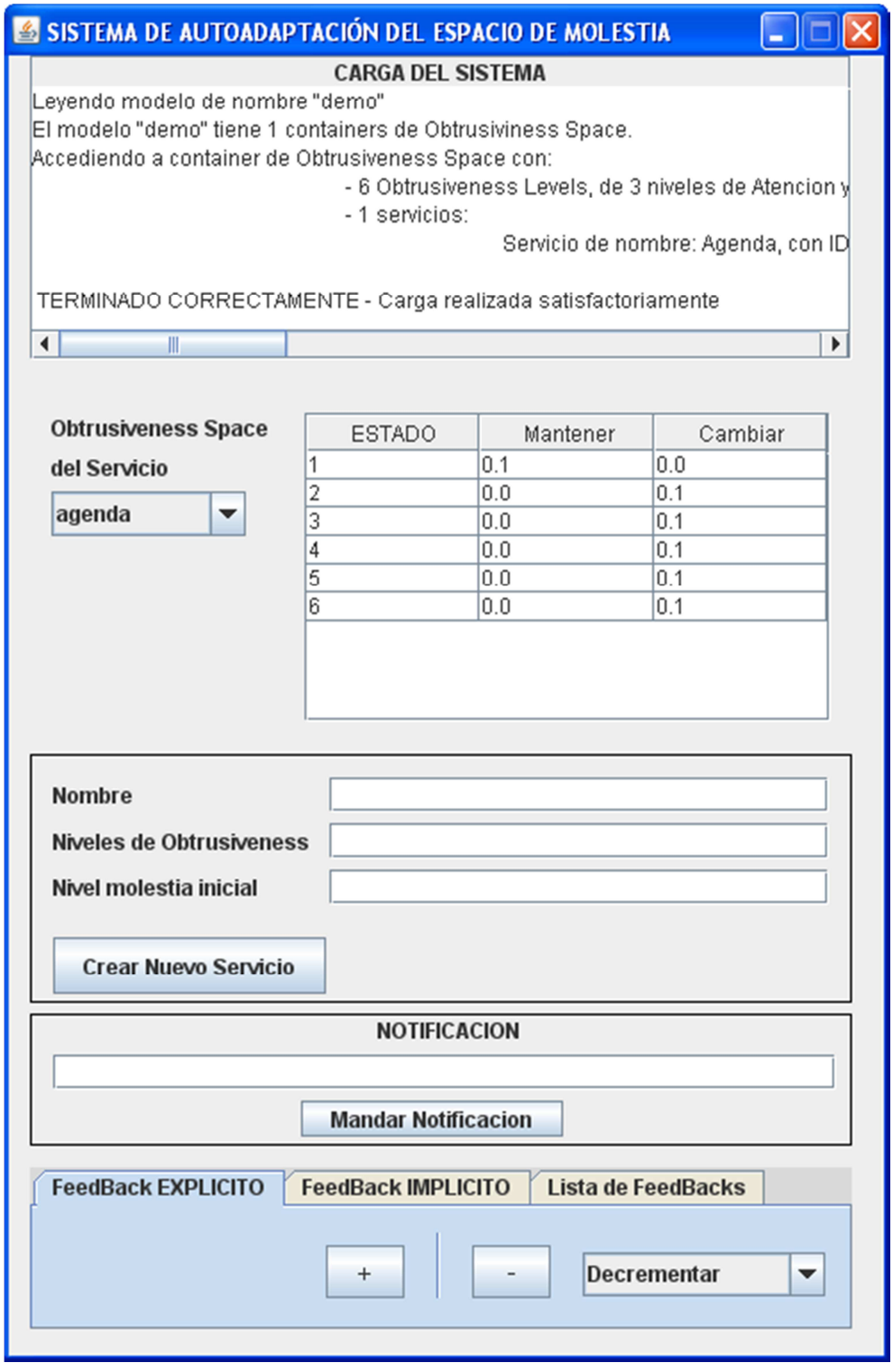

Figura 28 – Interfaz de nuestro sistema.

7.1.1.2 Carga del sistema

La primera zona del interfaz gráfico que observábamos en la figura 28 es un espacio para mostrar la información importada al realizar la carga del sistema que se produce al ejecutar el programa. En ella se muestran la cantidad de espacios de molestia del modelo importado, los servicios que hay en cada uno de esos espacios, los niveles de molestia de estos servicios, etc.

En el apartado 7.1.3 detallaremos como se produce dicha carga del sistema, al hablar de la capa de persistencia.

7.1.1.3 Espacio de adaptación de molestia

A continuación, se observa el área para mostrar el espacio de *adaptación del nivel de molestia* del servicio con el que el sistema esté interactuando en ese momento (como es el caso del servicio "*agenda*" en la figura 29).

Se puede distinguir una tabla donde se aprecian tres campos:

- "ESTADO": Es un índice con el número del estado de molestia que representa esa fila.
- "MANTENER": Es el peso de la acción "mantener" para el estado indicado en el campo "ESTADO".
- "CAMBIAR": Lo mismo que el anterior campo, pero para la acción "cambiar".

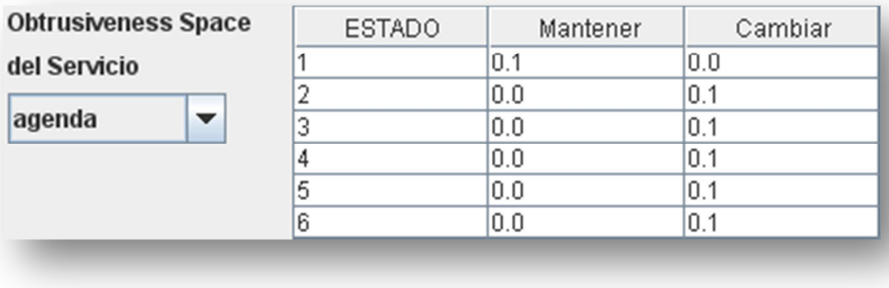

Figura 29 – Espacio de adaptación del servicio agenda.

Veremos el tipo de datos utilizado para implementar el *espacio de adaptación de molestia* de los servicios en el apartado 7.1.2.1 de estructuras de datos.

7.1.1.4 *Creación de nuevos servicios*

El siguiente espacio, tal como se aprecia en la figura 30, tendrá como cometido permitir la introducción de nuevos servicios en el sistema. Cuando tengamos que crear un servicio, tendremos que suministrarle los siguientes campos:

- "NOMBRE": Será el nombre del servicio, y también su identificador.
- "NIVELES DE OBTRUSIVENESS": Mediante este campo se indicará el número de niveles de molestia de los que constará el espacio de adaptación de molestia del servicio.
- "NIVEL MOLESTIA INICIAL": Indicará el nivel inicial de molestia del servicio (el que se le otorga en tiempo de diseño, y que posteriormente gracias a la adaptación de nuestro sistema, se podrá ajustar en tiempo de ejecución).

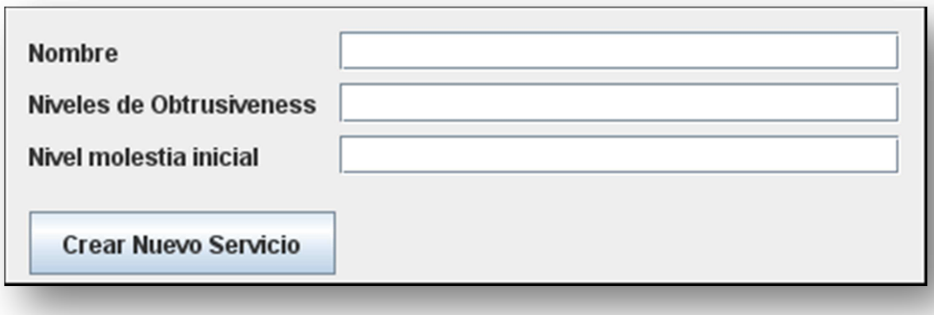

Figura 30 – Área de creación de nuevos servicios.

7.1.1.5 *Enviar notificación*

Posteriormente, existe un área para indicarle a un servicio que mande una notificación. Se dispondrá de un botón para dicha acción, y una vez la notificación sea mandada, se podrá ver por qué nivel se ha mandado la notificación por el recuadro de texto situado en la parte superior del botón, como se muestra en la figura 31.

Figura 31 – Área de notificaciones.

7.1.1.6 *Enviar feedback*

Por último, se encuentra la zona de obtención del *feedback* del usuario. Como puede observarse en la figura 32, es una zona dividida a su vez mediante pestañas en otras tres áreas.

Figura 32 – Área de *feedback explícito*.

Fijándose en la primera pestaña, se puede observar el espacio para devolver *feedback* **explícitamente**, mediante el botón  para devolver *feedback* positivo y el botón  para el negativo. Como muestra la figura 33, para el *feedback* negativo podemos especificar si se desea incrementar o disminuir el nivel de molestia del servicio que ha mandado la notificación de la cual se está realimentando con el *feedback*.

Figura 33 – Detalle de elección de modo de *feedback* (Incrementar/Disminuir).

La segunda pestaña, como se muestra en la figura 34, se corresponde con el espacio para devolver *feedback* **implícitamente.**

Figura 34 – Área de *feedback implícito*.

Mediante la lista de acciones seleccionables de la figura 35, se podrá simular devolver un comportamiento (por parte del usuario frente a la notificación recibida) al sistema, el cual se encargará de traducirlo, mediante una función semántica, en *feedback* positivo o negativo, y cuando sea de este último tipo, lo clasificará en modo disminuir o modo incrementar nivel.

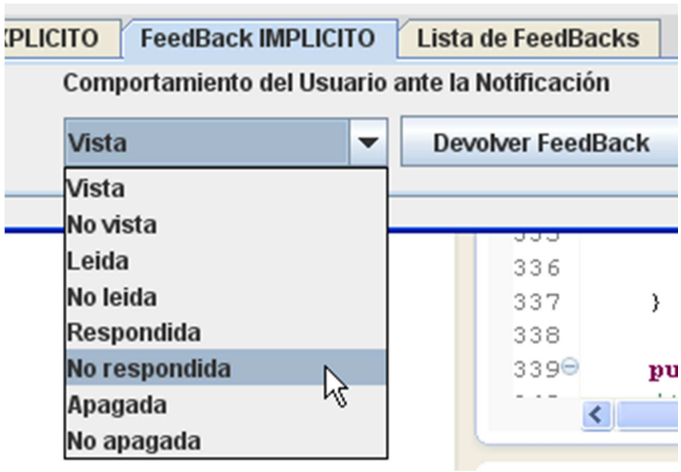

Figura 35 – Detalle de comportamientos de *feedback* implícito.

Por último, la tercera pestaña nos permite observar la lista de *feedback* negativo del servicio activo en el sistema, como se observa en la figura 36.

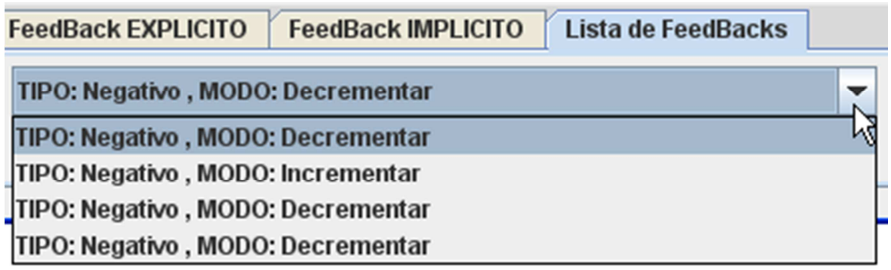

Figura 36 – Área de lista de *feedback*.

7.1.2 Capa de lógica o de negocio

En los sucesivos subapartados se presentarán el diseño de objetos del sistema, las colecciones de datos utilizadas en la implementación y una lista de los métodos a implementar, basándose en el diagrama de secuencia de la etapa de análisis, donde se detallarán los parámetros de entrada y de salida, así como que objeto ejecuta cada método.

7.1.2.1 Estructuras de datos

Para diseñar la colección de *feedbacks* que va recibiendo un servicio, cuyo modelo conceptual vimos en el apartado 6.5, se ha optado por utilizar una lista, en concreto, bajando al nivel de implementación con Java, una *LinkedList* como la representada en la figura 37, una lista enlazada de nodos donde cada nodo contiene elementos (objetos, otras listas,… en nuestro caso objetos *feedback*) y un puntero hacia la posición de memoria del siguiente nodo.

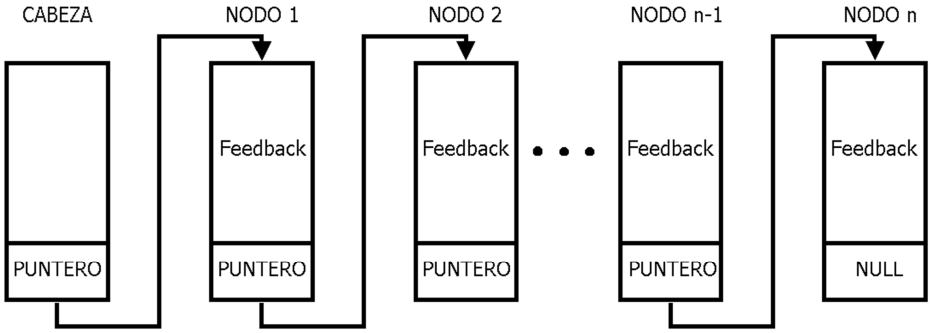

Figura 37 – Lista de *feedback*

Es un tipo de colección adecuada para nuestro sistema, pues ofrece facilidades para recorrerla secuencialmente mediante un iterador y para obtener el último y primer elemento (operación que se hace a menudo para obtener el signo del último *feedback*). Nuestro sistema nunca necesita acceder a posiciones concretas de la lista, por lo que no necesita otro tipo de colección como un vector o un mapa ordenado.

Para implementar la colección de servicios del sistema, se ha hecho uso de un mapa ordenado, que en nuestro proyecto servirá para "mapear" un valor clave (el identificador del servicio, en nuestro caso) a un objeto (el servicio). Específicamente, emplearemos un *TreeMap*, que vamos a utilizar como implementación de la interfaz *SortedMap*, interfaz Java que permitirá que los servicios de dentro del conjunto de la colección estén ordenados totalmente, facilitando por tanto su acceso en búsquedas y haciendo más rápida su consulta.

Por último, para implementar el *espacio de adaptación de molestia* de cada servicio, se ha utilizado un *array* multidimensional, concretamente uno de dos dimensiones, de manera que la primera dimensión se corresponderá con la acción “mantener” para el estado de número correspondiente con el índice del *array* (por ejemplo, el valor de la primera componente del tercer elemento del *array*, será el peso de la acción “mantener” para el estado 3). Lo mismo pasa para la

segunda componente, que se corresponderá con el peso de la acción "cambiar".

7.1.2.2 *Principales métodos*

En este subapartado se presentan a alto nivel, desde un punto de vista más amigable, las cabeceras de los métodos principales que recogen la lógica de negocio del sistema, y que posteriormente habrá que implementar. Se podrá apreciar que tienen una correspondencia casi directa con los pasos vistos en el diagrama de secuencia presentado en el análisis en el punto 6.6, así como con los pasos del algoritmo *Q-Learning*, algoritmo de aprendizaje base de nuestro sistema, analizado en el punto 5.3.3.

La tabla 4 muestra los métodos pertenecientes a la clase Servicio, presentada en el modelo conceptual del sistema en el punto 6.4

MÉTODO	PARÁMETROS		FUNCIONALIDAD
	In	Out	
sendNotification()	-	-	Cambia (o no) de estado por donde mandarse en base a una política de comportamiento. Después, manda una notificación por el nuevo estado (o por el estado antiguo si no ha cambiado).
behaviorPolicy(estado)	estado : Estado de que quiere estudiarse la política de comportamiento.	-	Estudia, para el estado que se le pasa como parámetro, cuál de sus dos acciones (cambiar o mantener) debe ser elegida, en base a una política de comportamiento, que para nuestro sistema será elegir la que tenga un peso mayor.
observeFB(feedBack)	feedBack: Feedback recibido por parte del usuario	-	Observa en *feedback* recibido ante una notificación, y en caso de ser negativo, lo guarda.
getFBmode()	-	FBmode: (i/d)	Devuelve el modo (incrementar/disminuir) del *feedback*

getNivelMolestia()	-	nivelMolestia	Devuelve el nivel de molestia del servicio actualmente seleccionado
semanticsFunction (feedTipoAux)	feedTipoAux	tipoFB,modoFB	Transformará un *feedback* implícito que se le pasa como parámetro de entrada, en un *feedback* explícito, devolviendo su tipo (positivo/negativo) y si modo (incrementar/disminuir)
updateV()	-	-	Actualiza el valor de calidad Q(s,a), correspondiente a la combinación entre el estado s y la acción a.

Tabla 4 – Listado de métodos de la clase Servicio.

La tabla 5 hace lo propio con los métodos de la clase Feedback, también presentada previamente en el modelo conceptual del análisis.

METODO	PARAMETROS		FUNCIONALIDAD
	In	Out	
getModo()	-	modo: (+/-)	Devuelve el modo (+/-) del *feedback*
getTipo()	-	Tipo: (i/d)	Devuelve el tipo (incrementar/disminuir) del *feedback*

Tabla 5 – Listado de métodos de la clase Feedback.

7.1.3 Capa de persistencia

Con el fin de hacer más realista el uso del programa implementado en nuestro desarrollo, nada más iniciarse la aplicación el sistema se cargará automáticamente con los servicios leídos a partir del modelo EMF de la figura 38, que representa un sistema de *espacios de adaptación de molestia* previamente existente. Así, se obtendrán algunos servicios lo que permitirá establecer un estado inicial en nuestra aplicación donde ya se podrá trabajar directamente con los servicios del sistema, sin necesidad de crear ningún servicio nuevo.

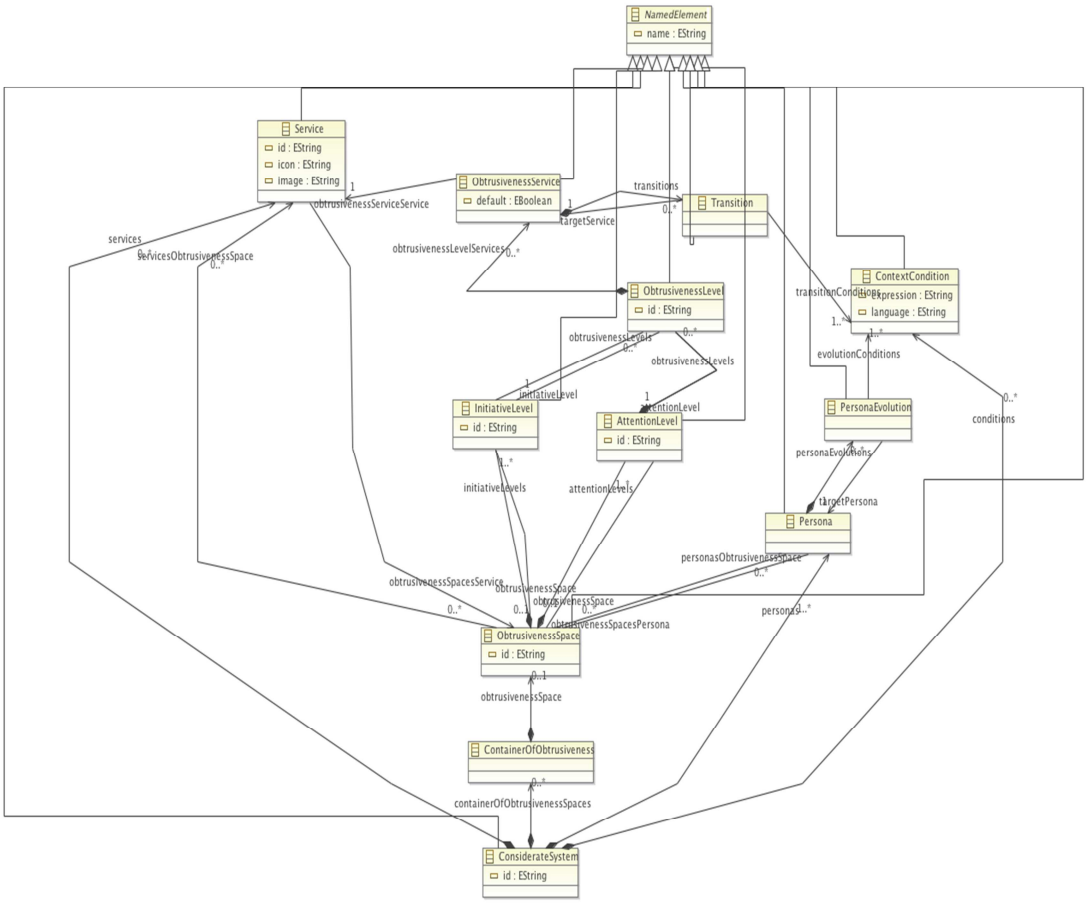

Figura 38 – Modelo EMF del sistema de Obtrusiveness Spaces

Para facilitar el acceso a este modelo EMF, dado en forma de un fichero XMI, se ha utilizado el *framework* OSGi, presentado en el apartado 2.2.2. A través de la clase Activator de este *framework*, presente en nuestro sistema, se encapsula la lectura de este modelo, ocultando la complejidad de dicha lectura y ofreciendo acceso a los elementos del modelo a partir de una variable de tipo *ConsiderateSystem* donde se cargará el sistema.

Concretamente, la clase Activator implementa la interfaz *org.osgi.framework.BundleActivator*, que obliga a implementar al menos dos métodos:

- Start: Método que se invoca al arrancar las aplicaciones OSGi.
- Stop: Método que se invoca al pararlas.

En el listado 1 se muestra la parte de la implementación del método *start*(), gracias a la cual cargamos en la variable *cs* el modelo leído, pudiendo accederse posteriormente a los elementos del modelo a través de ella (por ejemplo mediante órdenes del tipo *cs*.getContainerOfObtrusivenessSpaces())

```
public void start(BundleContext bundleContext) throws Exception
{
    Activator.context = bundleContext;



    XMIResourceFactoryImpl _xmiFac = new XMIResourceFactoryImpl();
    ResourceSet rSet = new ResourceSetImpl();
    rSet.getResourceFactoryRegistry().getExtensionToFactoryMap().put("*", _xmiFac);

    Resource res = rSet.getResource
    (URI.createFileURI("C:\\TFM\\workspace\\models\\demo.obtrusivenessmodel"), true);
    cs=(ConsiderateSystem)res.getContents().get(0);
```

Listado 1 - Método *start*() de la clase Activator

7.2 Implementación

Presentado el análisis y el diseño de nuestro sistema, se van a introducir algunos detalles de cómo ha sido realizada su implementación.

El programa que se ha desarrollado consiste en una implementación en Java del algoritmo analizado en el apartado 5.3.3. En la tabla 2 del mismo apartado, se aprecia que el algoritmo requiere 3 factores para empezar a trabajar:

Requiere:	*tasa de aprendizaje* (α),*parámetro de descuento* (γ), *valor umbral de Q(s,a)* (Δ)

Para implementar estos valores, se pensó en utilizar 3 constantes, llamadas ALFA, GAMMA y DELTA, cuya declaración en nuestro programa se muestra en el listado 2. En el mismo listado se puede

apreciar la declaración de otras variables utilizadas por el programa, que se irán comentando conforme se vayan utilizando.

Siguiendo con el algoritmo, el primer paso una vez inicializadas α, γ y Δ es el siguiente:

Paso 1	Inicializar s con el estado de molestia inicial, dando en dicho estado un peso mayor a la acción "*mantener*", mientras al resto de estados se le dará mayor peso a la acción "*cambiar*".

En este paso, se necesita inicializar el espacio de adaptación de molestia del servicio (en adelante *obtrusivenessSpace*). En nuestro sistema, se ha representado el *obtrusivenessSpace* por medio de la variable de nombre *obtrusivenessSpace*, implementada en forma de un *array* multidimensional (como ya se mostró al ver el diseño de las estructuras de datos en el punto 7.1.2.1). Puede apreciarse su declaración en el listado 2.

```
// Constantes y variables para la formula:
// Q(s,a)<-[Q(s,a) + ALPHA*(R + GAMMA*maxa'€A(s')Q(s',a') - Q(s,a))]
static final double ALPHA=0.5;  //learning parameter
static final double GAMMA=0.99; //discount factor
private int R;
private double Qsa; // Q(s,a)

// Constante para la formula:
/* V(s,a)<- {   -DELTA if Q(s,a)<=-DELTA
                Q(s,a) if -DELTA<Q(s,a)<DELTA
                DELTA if Q(s,a)>=DELTA
            ) */
static final int DELTA=3;       //limite value
// (s) Nivel de molestia en que se encuentra el servicio
int nivelMolestia;
// (s') Nivel de molestia "temporal" (usado por el propio algoritmo)
int nivelMolestiaAux;
// Para saber a que estado hay que saltar en los primeros saltos
// que se producen en cualquier ajuste de nivel de molestia
int nivelMolestiaPrevio;

//Espacio de Adpatación de molestia
double[][] ObtrusivenessSpace;
// Estados maximo y minimo
    int sMin;
    int sMax;
// Acciones (en forma de indices de un array, para facilitar su uso)
    int columna_mantener=0;
    int columna_cambiar=1;

LinkedList<FeedBack> listaFB;
```

Listado 2 - Declaración de las variables del algoritmo en la clase Servicio

La inicialización del *obtrusivenessSpace* de este primer paso se realizará de la siguiente forma:

- En el estado (o nivel) “s” (que se corresponderá con el nivel de molestia inicial del servicio), la acción “mantener” se inicializará con un valor más grande que la acción cambiar (recordemos que cuando pasa esto, significa que si el servicio está en este estado, no desea cambiar su nivel de molestia).
- En el resto de estados del *obtrusivenessSpace*, se hará a la inversa.

Esta inicialización se realiza en el constructor de la clase Servicio, tal como muestra el listado 3. En él, vemos como se inicializa la variable *obtrusivenessSpace* con la lógica que acabamos de explicar.

```
public Servicio(String id, int cantidadNiveles, int nivelMolestia) {
    this.id=id;
    //Inicializar nivel de molestia
    // RESTAMOS -1 porque las listas y vectores contaran desde 0
    this.nivelMolestia          = nivelMolestia-1;  //s
    this.nivelMolestiaAux       = this.nivelMolestia;  //s'
    this.nivelMolestiaPrevio    = this.nivelMolestia;

    // Inicializar el Espacio de adaptacion de molestia
    ObtrusivenessSpace = new double [cantidadNiveles][2];
    // Se inicializa a 0.0 en todos los niveles
    // El [2] es para tener 2 columnas de acciones
    // una para mantener, y otra para cambiar
    sMin=0;
    sMax=cantidadNiveles-1;
    // RESTAMOS -1 porque las listas y vectores contaran desde 0

    Qsa=0.1;    // Valor con el que vamos a inicializar las columnas
                // mantener/cambiar cuando no deban ser 0.0
    for (int i=sMin;i<=sMax;i++)
    {
        if (i==this.nivelMolestia)
        {
            ObtrusivenessSpace[this.nivelMolestia][columna_mantener]=Qsa;
            //MANTENER del nivel de molestia inicial
        }
        else
        {
            ObtrusivenessSpace[i][columna_cambiar] =Qsa;
        }
    }

    //Para guardar los futuros feefbacks
    listaFB = new LinkedList<>();

} // Fin constructor de Servicio
```

Listado 3 - Constructor de la clase Servicio

Comentar que las variables *columna_mantener* y *columna_cambiar* están inicializadas respectivamente a 0 y 1 (como se aprecia en el listado 2), y simplemente tienen la función de recordar y facilitar el acceso a la columna del *array* bidimensional *obtrusivenessSpace* que pertenece a la acción deseada, o bien la acción mantener (la primera columna, de índice 0) y bien la acción cambiar (la segunda columna, de índice 1).

Una vez explicado el primer paso, se continúa con la disección del algoritmo:

Paso 2	**repetir**	
…		
Paso 12	**hasta**	fin de proceso de aprendizaje

Se observa claramente la estructura de un bucle entre los pasos 2 y 12 (el último) del algoritmo. En nuestro sistema, al estar implementado mediante una interfaz gráfica y estar dirigido por eventos, esos pasos no tienen traducción por nada concreto en la implementación. El sistema estará siempre "escuchando" hasta que el usuario determine que ha terminado el proceso de aprendizaje de los servicios y cierre el programa.

Siguiendo con el paso 3 del algoritmo, tenemos:

Paso 3	**cuando** haya un notificación N_t que enviar:

En nuestro sistema, habrá una notificación que enviar cuando así lo decida el usuario. Para ello, en el área de notificaciones del interfaz que vimos en la figura 31 al presentar la interfaz de usuario, pulsará en el botón de enviar notificación. El sistema tiene un *listener* asociado a esa acción de pulsar ese botón, que ejecutará el código mostrado en el listado 4.

```
JButton NOTIFICACION_btnMandar = new JButton("Mandar Notificacion");
NOTIFICACION_btnMandar.setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
NOTIFICACION_btnMandar.addActionListener
(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent arg0)
        {
            NOTIFICACION_textFieldWhere.setText( servicio.sendNotification() );
        }
    }
);
```

Listado 4 - Listener del botón "mandar notificación" en la clase Formulario

El *listener* ejecutará la acción de mandar notificación, invocando al método de la clase Servicio *sendNotification*(), que se va a ir analizando por partes:

Dentro de *sendNotification*() lo primero que se ejecuta es una invocación del método *behaviorPolicy*(), cuyo código podemos observar en el listado 5.

```
public String behaviorPolicy(int estado){

    String accion="mantener";

        if (ObtrusivenessSpace[estado][columna_mantener]
            < ObtrusivenessSpace[estado][columna_cambiar])
        {
            accion="cambiar";
        }
    return accion;

} // Fin behaviorPolicy
```

Listado 5 - Método *behaviorPolicy()* de Servicio

Este método *behaviorPolicy*(), en base al funcionamiento que se ha detallado al explicar el paso 1 del algoritmo (cuando el peso de la acción mantener era mayor que el peso de la acción cambiar, significaba que no había que cambiar de estado), devolverá un *string* con valor de "mantener" o de "cambiar", dependiendo de los valores que tenga el *array* *obtrusivenessSpace* en el estado que se le esté pasando al método por medio del parámetro *estado*.

Siguiendo con *sendNotification*() , en base al resultado de invocar a *behaviorPolicy*(), elegirá que acción va a realizar, si mantener el estado por donde mandará la notificación o cambiarlo, que se corresponde con el siguiente paso del algoritmo:

Una vez elegida la acción, la ejecuta, tal como dicta el siguiente paso del algoritmo:

Paso 5	Llevar a cabo dicha acción "a"

Si la acción elegida ha sido mantener, *sendNotification()* mantendrá el estado actual s, pero si la acción elegida ha sido cambiar, modificará el *nivel de molestia* del servicio, cambiando de estado s (realmente, no cambia de estado s todavía, cambia el nivel del estado s' [inicializado al mismo nivel que s], que es un estado auxiliar que usa el algoritmo).

Este cambio de nivel de molestia, puede realizarse hacia un nivel inferior o hacia un nivel superior (incluso no realizarse, si se ha llegado al límite marcado por las variables *sMax* o *sMin*, inicializadas respectivamente con los niveles máximo y mínimo del *obtrusivenessSpace*, como puede apreciarse en el listado 2).Hacia qué dirección cambiar, vendrá dado por la frecuencia del modo del feedback almacenado desde el último cambio (tal como se detalló en el apartado 6.5). El método *getFBmode()* es el que se encarga de calcular las frecuencias y devolver la dirección adecuada ("i" para incrementar, "d" para disminuir o "n" para no cambiar, para el caso de que se ha llegado al límite máximo o mínimo de niveles posibles para ese *obtrusivenessSpace*).

En el listado 6 mostramos la parte de su código que recorre la *LinkedList* del *feedback* calculando las frecuencias.

```
Iterator<FeedBack> iter = listaFB.iterator();
while(iter.hasNext())
{
    switch (iter.next().getTipo())
    {
        case "i":
            incrementos += 1;
            break;
        case "d":
            decrementos += 1;
            break;
    }
}
```

Listado 6 - Método *getFBmode*() de la clase Servicio, recorriendo lista de FB

Por último, una vez *sendNotification*() ha modificado (o no) el nivel de molestia del servicio, manda la notificación, devolviendo un *string* que indica por qué nivel de molestia se ha mandado la notificación, cumpliendo con el siguiente paso del algoritmo:

Paso 6	Mandar N_t por el nuevo estado s'

Recordemos, que s' es un estado auxiliar del algoritmo (implementado en nuestra solución mediante la variable *nivelMolestiaAux*, cuya inicialización puede verse en el listado 2), que valdrá lo mismo que s (si *sendNotification*() no ha cambiado su valor) o valdrá s+1 o s-1 (si *sendNotification*() ha incrementado o disminuido su valor respectivamente), tal como se ha explicado en los párrafos anteriores.

Una vez mandada la notificación, el método *sendNotification*() ha terminado y el sistema pasa a esperar el *feedback* del usuario ante dicha notificación. Las formas en que los usuarios pueden devolver el *feedback*, que tipos de feedback admite el sistema (explícito/implícito, positivo, negativo,…) y como mandarlo, ya han sido explicadas detalladamente en el apartado 7.1.1.6. Los listados 7 y 8 muestran

respectivamente los *listeners* del botón de envío de *feedback* explícito positivo y explícito negativo.

```
btnFBExplicitoPositivo.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0)
    {

        feed = new FeedBack("+","+");

        servicio.observeFB(feed);
        servicio.updateV();
        servicio.changeLevel();

        actualizarOBTSpace(servicio.id);

    }
});
```

Listado 7 - *Listener* de botón de *feedback* explícito positivo

```
btnFBExplicitoNegativo.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        String tipo;
        tipo = comboBoxFBdecreIncre.getSelectedItem().toString();

        if ("Decrementar".equals(tipo)) {tipo="d";}
        else {tipo="i";}

        feed = new FeedBack("-",tipo);

        servicio.observeFB(feed);
        servicio.updateV();
        servicio.changeLevel();

        actualizarOBTSpace(servicio.id);

    }
});
```

Listado 8 - *Listener* de botón de *feedback* explícito negativo

Se puede observar que la lógica de ambos códigos es exactamente la misma, excepto que la del *feedback* negativo, determina el tipo del nuevo *feedback* que va a crear en base al elemento seleccionado actualmente en el *comboBox* de nombre *comboBocFBdecreIncre*, que

se corresponde con el *comboBox* del formulario grafico que muestra los tipos de feedback negativo disponibles (Incrementar/Disminuir), como ya se mostró en la figura 33.

Una vez mandado el *feedback* (una vez creado), se ejecuta el método *observeFB*(), que en nuestra implementación cumplirá con los pasos 7,10,11 y 12, que han sido implementados juntos por eficiencia.

Paso 7	Observar el *feedback* r ante dicha notificación
Paso 10	**si** r es negativo **entonces**:
Paso 11	Guardar el modo (incrementar/disminuir) del *feedback*
Paso 12	**fin si**

El método *observeFB*(), mostrado en el listado 9, empieza creando una copia del *feedback* que se le ha pasado como parámetro de entrada en la variable global *lastFB*, con el objetivo de, cuando la lista de *feedback* está vacía porque se acaba de realizar un cambio de nivel de molestia (en cada cambio, la lista de *feedback* se vacía para partir de cero con el cálculo de las frecuencias de tipos de *feedback* [recordemos, tipo incrementar/ Disminuir]), poder saber si el último *feedback* fue positivo o negativo, necesario esto para saber si tener que incrementar o disminuir el nivel en el caso de que hubiera que volver a cambiar dicho nivel antes de introducir más *feedback*, ya que no se podría hacer en base al cálculo de las frecuencias (porque recordemos que la lista estaría vacía).

```
public void observeFB(FeedBack feedBack)
{
    lastFB=feedBack;

    if ("-".equals(lastFB.getModo()))
    {
        listaFB.add(feedBack);
    }
} // Fin metodo fBobserveFB
```

Listado 9 - Método observeFB de la clase Servicio

Por último, *observeFB()* observa el *feedback* que se le ha enviado como parámetro de entrada, obtiene su modo (+ o -) mediante el método de la clase Feedback *getModo()*, y en caso de que sea del tipo negativo, lo añade a la lista de *feedback*. Esta lista de *feedback* ya ha sido presentada y analizada al ver el diseño de las estructuras de datos, solo recordar que era una *LinkedList* de java (cuya declaración podemos observar en el listado 2).

Terminado *observeFB()*, seguimos con el siguiente paso del algoritmo:

Paso 8	Actualizar V(s,a)

El método *updateV()* es el método que se encargará de este paso del algoritmo, donde se actualizan los valores del *obtrusivenessSpace* del servicio, según se ha estudiado y analizado en los apartados 5.3.3 y 5.3.3.1 (y cuya actualización también se mostrará con un ejemplo práctico en el apartado 8.1.1.3). Como es un método importante (y largo), se va a ir analizándolo por partes:

El método empieza, tal como se muestra en el listado 10 calculando r, que como se expuso en el capítulo 5.3.3, valdrá +1 en caso de que se haya devuelto *feedback* positivo, o -1 en caso de *feedback* negativo.

```
public void updateV(){
    // Calculo de factor r dependiendo de tipo de feedback
    if ( "+".equals( lastFB.getModo() ) ) {r=1;}
    else {r=-1;}
```

Listado 10 - Método *updateV()* de Servicio – Calculo de r

Después, el método calcula Q(s,a) mediante la siguiente fórmula (analizada en el apartado 5.3.3):

$$Q(s, a) \Leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a))$$

Para ello, descompone su cálculo en dos partes. Primero calcula el término $max_{a' \in A(s')} Q(s', a')$, tal como se muestra en el listado 11.

```
// CON s' : Calculo de factor maxa'EA(s')--------------------------------
double max_aSPrima;

if ( "cambiar".equals(behaviorPolicy( this.nivelMolestiaAux )) )
{
    max_aSPrima=
        ObtrusivenessSpace[this.nivelMolestiaAux][columna_cambiar];
}
else
{
    max_aSPrima=
        ObtrusivenessSpace[this.nivelMolestiaAux][columna_mantener];
}
```

Listado 11 - Método updateV() de Servicio – Calculo de max_aSPrima

Podemos observar que inicializará el valor de la variable *max_aSPrima* al valor del *array* *obtrusivenessSpace* en el índice formado por:

- Primera componente: *nivelMolestiaAux*.
- Segunda componente: Dependiendo del resultado de aplicarle la misma política de comportamiento que veíamos en el listado 5 con el método *behaviorPolicy*(), la segunda componente será la *columna_mantener* (que recordemos equivalía al índice 0 (primera columna)) o la *columna_cambiar* (índice 1 (segunda columna)).

[Por ejemplo, si tuviera que inicializar al termino *max_aSPrima* con el valor de *obtrusivenessSpace* [*nivelMolestiaAux*][*columna_cambiar*] (siendo *nivelMolestiaAux* por ejemplo 4), estaría inicializándolo con el valor de del *array* en la posición [4][1]]

Es importante destacar que tanto la política de comportamiento, como el valor que se estaba extrayendo de *obtrusivenessSpace*, estaban referidos a *nivelMolestiaAux* (s'), que recordemos que inicialmente vale lo mismo que *nivelMolestia* (s), pero que en el método *sendNotification*() puede haberse aumentado o disminuido (tal como se

ha explicado al hablar de *sendNotification()* y ver el paso 6 del algoritmo).

Una vez calculado el término *max_aSPrima*, *updateV()* calcula por fin el valor del término Q(s,a), tal como se aprecia en el listado 12. En él, vemos como calcula el valor actual de Q(s,a) de una manera similar a como se ha explicado que calculaba *max_aSPrima*, pero en esta ocasión no para *nivelMolestiaAux* (s') sino para *nivelMolestia* (s), y con el resultado inicializa a la variable *Qsa*.

Una vez tiene los valores de los 2 términos de la fórmula, la calcula dejando el resultado en *Qsa*.

```
// CON s: CALCULO DE Q(s,a) ----------------------------------------
if ( "cambiar".equals(behaviorPolicy(nivelMolestia)) )
{
    Qsa     = ObtrusivenessSpace[nivelMolestia][columna_cambiar];
}
else
{
    Qsa     = ObtrusivenessSpace[nivelMolestia][columna_mantener];
}
// Q(s,a)<-[Q(s,a)   + ALPHA*(R + GAMMA*maxa'€A(s')Q(s',a') - Q(s,a)) ]
   Qsa     = Qsa      + ALPHA*(R + GAMMA*max_aSPrima          - Qsa    );

   Qsa = Redondear(Qsa);
```

Listado 12 - Método updateV() de Servicio – Calculo de Q(s,a)

Tal como se explicó en el apartado 5.3.3.1, el algoritmo *Q-Learning* implementado por nuestra solución tiene una mejora dada en forma de la función V, que pone un valor tope al peso que puede alcanzar una acción en un estado (con lo que se consigue que necesite menos tiempo en aprender):

$$V(s,a) = \begin{cases} -\Delta & \text{si} \quad Q(s,a) \leq -\Delta \\ Q(s,a) & \text{si} \quad -\Delta < Q(s,a) < \Delta \\ \Delta & \text{si} \quad Q(s,a) \geq \Delta \end{cases}$$

Por lo tanto, una vez ha calculado *Qsa*, el método *updateV()* continua actualizando *Qsa* en función de esa función, tal como se aprecia en el listado 13

```
if ( Qsa >= DELTA || Qsa <= (DELTA*-1) )
{
     Qsa=DELTA;
}
```

Listado 13 - Método *updateV()* de Servicio – Función V

Calculado el valor final de *Qsa*, se procederá por último a adaptar con él al valor de Q(s,a) del estado actual s, en el *ObtrusivenessSpace*, tal como muestra el listado 14.

```
// CON s: ACTUALIZACION DE Q(s,a) -----------------------------
if ( "cambiar".equals(behaviorPolicy(nivelMolestia)) )
    {
        ObtrusivenessSpace[nivelMolestia][columna_cambiar]=Qsa;
    }
else
    {
        ObtrusivenessSpace[nivelMolestia][columna_mantener]=Qsa;
    }
```

Listado 14 - Método *updateV()* de Servicio – Adaptación del ObtrusivenessSpace

Por último, el método *updateV()* implementa el último paso del algoritmo:

Paso 9	s:=s'

Para ello, invoca al método *changeLevel()* , cuyo código se muestra en el listado 15, que simplemente actualiza el valor de la variable *nivelMolestia* al valor que tenga la variable *nivelMolestiaAux*. Es aquí donde, si en el método *sendNotification()* (basándose en la política de comportamiento que hemos explicado al analizar dicho método) ha decidido que hay que cambiar el nivel de molestia del servicio (y por lo tanto habrá aumentando o disminuyendo el valor de *nivelMolestiaAux*), se actualiza el valor del estado actual s (recordemos, representado por la variable *nivelMolestia*) al estado s' por el que *sendNotification()* ha

decidido que hay que cambiar (recordemos, representado por la variable *nivelMolestiaAux*).

```
public void changeLevel()
{
    nivelMolestia=nivelMolestiaAux;
}
```

Listado 15 - Método *changeLevel*() de Servicio

En este punto, se ha presentado la implementación de todo el algoritmo de aprendizaje. Como hemos comentado al explicar los puntos 1 y 12 del algoritmo, a partir de haber procesado el *feedback*, el sistema entra en un bucle representado por la espera de una nueva interacción por parte del usuario, bien para crear un nuevo servicio, bien para seleccionar un servicio distinto o bien para seguir aprendiendo el nivel de molestia del servicio actual.

CAPÍTULO 8

Caso de estudio

Una vez introducidos los conceptos básicos necesarios para la mejor comprensión del trabajo realizado en esta tesis, así como una exposición de la teoría del sistema de auto-adaptación de preferencias desarrollado, desde sus maneras de captar el *feedback* del usuario hasta el algoritmo de aprendizaje en que se basa, se procederá ya a presentar dicho sistema de una manera práctica, mostrando el interfaz real del sistema y realizando un ejemplo de utilización con el que demostrar la funcionalidad del sistema desarrollado, así como su forma de uso, ilustrándose con capturas del propio sistema.

Se mostrarán distintos escenarios en los que se trabajará con servicios de distinta procedencia, cargados desde un modelo de *espacio de adaptación de molestia* importado, o bien con nuevos servicios creados en tiempo de ejecución.

También se mostrarán los resultados de una evaluación experimental basada en el ejemplo práctico presentado.

8.1 Un caso práctico

Para demostrar la funcionalidad del sistema desarrollado, así como su forma de uso, se va a proceder a realizar un caso de práctico de estudio. Se presentará el sistema de auto-adaptación del espacio de molestia y como, para un servicio seleccionado, se podrá simular un envío de una notificación, para que posteriormente el usuario pueda devolver un *feedback*. El sistema mostrará en todo momento el espacio

de molestia del servicio actualmente seleccionado, por lo que podrá verse su adaptación y su evolución a lo largo del tiempo, conforme se vaya recibiendo *feedback* positivo o negativo.

El usuario podrá interactuar con servicios de cada uno de los dos escenarios siguientes:

1) El usuario dispondrá de un sistema precargado con una serie de servicios importados de un sistema previamente existente.
2) El usuario tendrá la capacidad de crear e introducir nuevos servicios en el sistema.

En cualquiera de ambos escenarios, podrá seleccionar un servicio e interaccionar con él, mediante la simulación de envío de notificaciones y mediante el envío de *feedback* (tanto explícito como implícito).

8.1.1 Ejemplo de uso del sistema

Ya presentada la interfaz gráfica en el apartado de diseño, se va a proceder a la demostración de su uso del sistema, que incluirá dos escenarios diferentes, uno cargando el sistema con un *espacio de adaptación de molestia* importado a partir de un modelo EMF previamente existente, y otro donde mostraremos la creación de un servicio nuevo con el que interactuaremos.

8.1.1.1 Carga del sistema

Nada más ejecutarse, nuestro sistema se precargará con los servicios de un sistema preexistente que importamos a través de un modelo EMF. En nuestro ejemplo, tal como podemos observar en la figura 39, vamos a cargar un sencillo modelo de nombre “*demo*”, que contiene un único espacio de adaptación de molestia, que a su vez contiene un único servicio, “*agenda*”, con 6 niveles de molestia y un nivel inicial igual a 1.

Figura 39 – Ejemplo de carga del sistema.

Una vez se ha realizado la carga del sistema, el usuario ya puede interactuar con él, en este caso interactuando con el servicio "*agenda*" cuyo espacio de adaptación de molestia ya observamos anteriormente en la figura 29. Observamos que dicho servicio está inicializado con nivel de molestia inicial de 1, por lo cual el peso de la acción "mantener" en el estado 1 será mayor que el peso de la acción "cambiar", mientras para el resto de estados será al contrario, tal como resaltamos en la figura 40.

ESTADO	Mantener	Cambiar
1	0.1	0.0
2	0.0	0.1
3	0.0	0.1
4	0.0	0.1
5	0.0	0.1
6	0.0	0.1

Figura 40 – Espacio de adaptación de molestia del servicio "*agenda*".

Se va a desarrollar la siguiente secuencia:

- Por cada notificación del servicio agenda por el nivel 1, el usuario va a devolver *feedback* positivo, significando que le satisface la interacción con el servicio por dicho nivel.
- Cuando se alcanzar el valor umbral (Δ) para el peso de la acción mantener del estado 1, el usuario cambiará de opinión y empezará a introducir *feedback* negativo, tanto explícito como

implícito, con la intención de incrementar el nivel de molestia del servicio del estado 1 al estado 3.

El primer paso será simular el envío de una notificación. El usuario pulsará en el botón 

, dando como resultado el mensaje en el área de notificaciones que se aprecia en la figura 41:

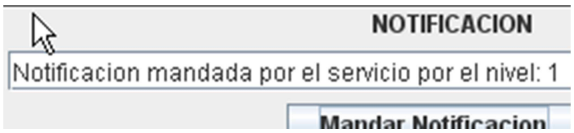

Figura 41 – Ejemplo de mensaje por área de notificaciones

Entonces, el usuario mandará un *feedback* positivo pulsando en el correspondiente botón 

de la pestaña de *feedback* explícito mostrada en la figura 32. El sistema, mediante el algoritmo de aprendizaje, procesará el *feedback* y actualizará los pesos de las acciones para el estado 1:

Se volverá a repetir el mismo proceso de simular notificación – devolver *feedback*, hasta alcanzar el umbral para la acción mantener (recordemos, que en nuestro sistema dicho umbral tiene un valor de 3).

La iteración de este proceso hasta llegar al umbral, provoca las actualizaciones en los pesos de las acciones del estado 1 apreciables en la tabla 6.

Notificación	Mandada por estado	FEEDBACK	ESTADO	Mantener	Cambiar
1	1	+	1	0.5995	0.0
2	1	+	1	1.0965	0.0
3	1	+	1	1.591	0.0
4	1	+	1	2.083	0.0
5	1	+	1	2.5726	0.0
6	1	+	1	3.0	0.0

Tabla 6 – Ejemplo de adaptación ante *feedback* positivo.

Siguiendo la secuencia marcada como objetivo, llegados al umbral de la acción mantener del estado 1, el usuario cambia de idea y ante la siguiente notificación que reciba desde el estado 1 devolverá un *feedback* de tipo negativo pulsando en el botón  de la pestaña de *feedback* explícito de la figura 32, seleccionando el modo "incrementar".

Seguirá este proceso de notificaciones – *feedback* negativo hasta que consiga que el servicio le mande la notificación por el estado 3, dando como resultado la tabla 7:

Notificación	Mandada por estado	FEEDBACK	ESTADO	Mantener	Cambiar
7	1	-	1	2.485	0.0
8	1	-	1	1.9726	0.0
9	1	-	1	1.4627	0.0
10	1	-	1	0.9554	0.0
11	1	-	1	0.4506	0.0
12	1	-	1	-0.0517	0.0

Tabla 7 – Ejemplo de adaptación ante *feedback* negativo.

Llegados a este punto, la siguiente notificación (la notificación 13), al ejercerse la política de comportamiento que compara el peso de las acciones del estado 1 y ver que la acción cambiar tiene mayor peso que la acción mantener, ya será enviada por el nivel 2, como se aprecia en la figura 42.

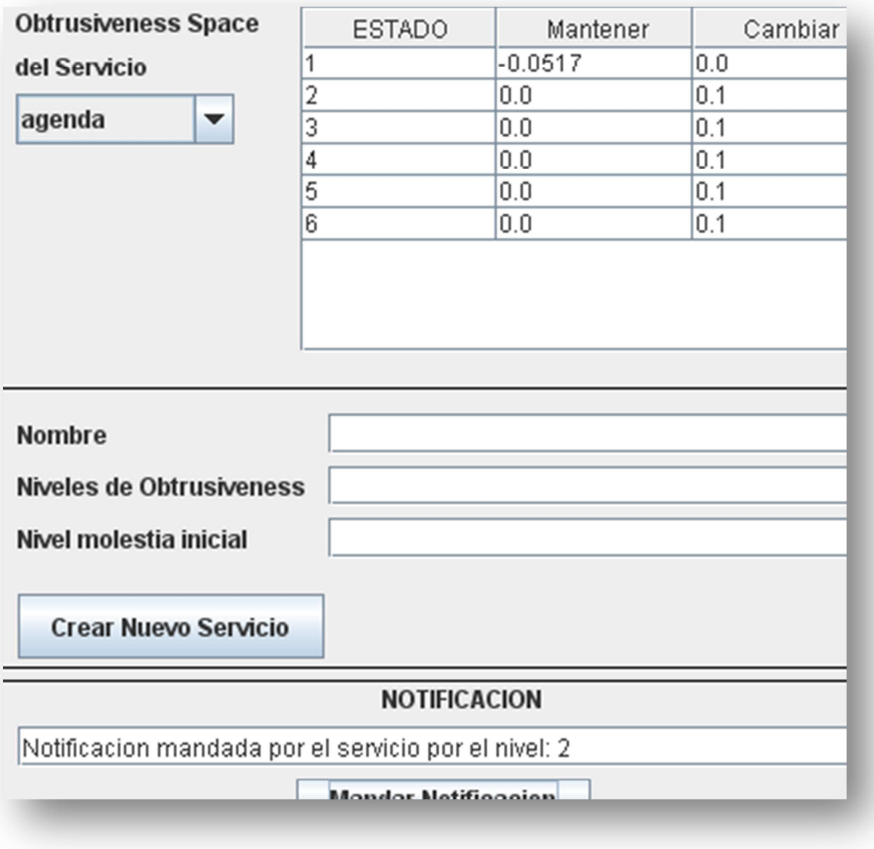

Figura 42 – Detalle de cambio de nivel en el envío de una notificación.

Como nuestra intención es conseguir que el servicio interaccione en el nivel 3, seguiremos enviando *feedback* negativo de modo incrementar, hasta que consigamos que las notificaciones se manden por el nivel 3, lo que dará lugar a la secuencia de datos de la tabla 8:

Notificación	Mandada por estado	FEEDBACK	ESTADO	Mantener	Cambiar
13	2	-	1	-0.0517	-0.4505
14	3	+	1	-0.0517	-0.4505
			2	0.0	0.5995

Tabla 8 – Actualización del peso de las acciones en varios estados.

Obsérvese como, a medida que se cambia de estado, los pesos que se modifican ya no se modifican sobre el estado inicial (el 1), sino que se van modificando los pesos de los estados establecidos como nuevos estados de molestia.

En este punto se ha llegado al objetivo, el nivel de molestia del servicio será ahora de 3. Sin embargo, si volvemos a mandar una notificación, obtenemos el siguiente mensaje:

Lo que significa que el servicio ha vuelto a modificar su nivel de molestia, del 3 que era el nivel deseado por el usuario, al nivel 2. Esto es debido a pequeños ***saltos de ajuste*** que produce el algoritmo de aprendizaje mientras esta "aprendiendo" el nivel correcto deseado. Por lo tanto, se volvería a mandar un *feedback* negativo de modo "incrementar" hasta que el nivel de molestia del servicio se estabilice correctamente en el nivel deseado (recordemos que era el nivel 3). Continuando la tabla anterior a partir de la notificación 15, el resultado de notificaciones hasta alcanzar dicho nivel seria la correspondiente con la siguiente tabla 9:

Notificación	Mandada por estado	FEEDBACK	ESTADO	Mantener	Cambiar
15	2	-	1	-0.0517	-0.4505
			2	0.0	0.5995
			3	0.0	-0.1532
16	3	+	1	-0.0517	-0.4505
			2	0.0	0.7998
			3	0.0	-0.1532
17	3	+	1	-0.0517	-0.4505
			2	0.0	0.7998
			3	0.5	-0.1532
18	3	+	1	-0.0517	-0.4505
			2	0.0	0.7998
			3	0.9975	-0.1532

Tabla 9 – Saltos de ajuste mientras se está aprendiendo el nivel deseado.

Al llegar a la notificación número 16 es cuando se ha alcanzado la estabilidad en el nivel de molestia 3, siendo en las notificaciones posteriores, que vienen ya todas por su nivel correcto, cuando podemos afirmar que se ha logrado el aprendizaje del nivel (el ajuste

del *espacio de adaptación de molestia* del servicio *agenda*, de manera satisfactoria con las necesidades del usuario).

8.1.1.2 Creación de nuevos servicios

Procederemos ahora a presentar la creación de nuevos servicios y a mostrar cómo interaccionar con ellos. Para ello, se va a replicar el ejemplo de caso de estudio visto en [18], creando el servicio "*healthcare*" con 3 niveles de molestia, correspondientes con 3 niveles de atención (*invisible, slightly y aware*) y uno de iniciativa (*proactive*). Vamos a suponer que es un servicio importante para el usuario, y por tanto su nivel de molestia por defecto será el nivel 3 (*aware-proactive*), que por ejemplo pude tener configurada una interacción acústica, en forma de mensaje de voz.

Para crear dicho servicio, nos posicionaremos en el área de creación de nuevos servicios y rellenaremos los datos del servicio "*healthcare*", tal como ilustra la figura 43.

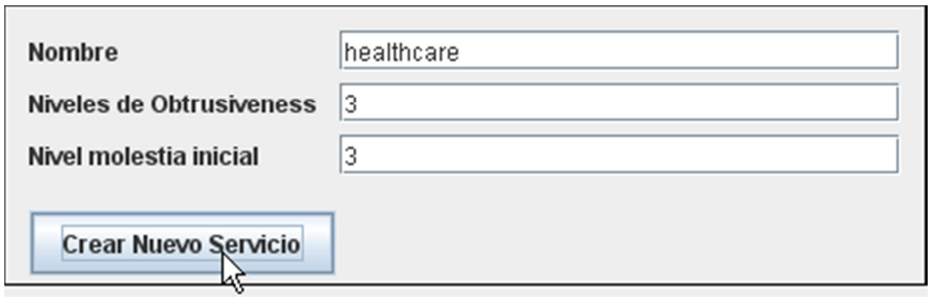

Figura 43 – Ejemplo de creación del servicio "*healthcare*".

El sistema creará el nuevo servicio, y nos mostrará automáticamente su espacio de adaptación de molestia, como muestra la figura 44.

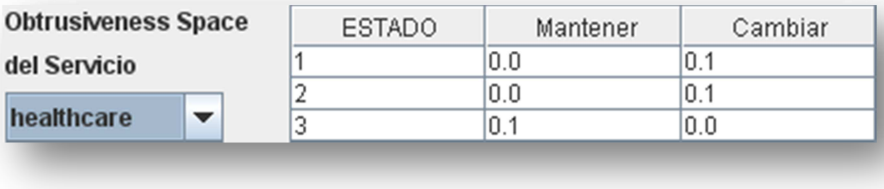

Figura 44 – Espacio de adaptación de molestia del servicio "*healthcare*".

8.1.1.3 *Interactuando con el sistema*

Como ejemplo de aplicación, siguiendo con el ejemplo del sub-apartado anterior, vamos a suponer que el usuario, que inicialmente estaba desempleado, ha conseguido trabajo y ahora pasa la mayor parte del día rodeado de sus compañeros. Aunque el servicio "*healthcare*" sigue siendo muy importante para él, no quiere que dichos compañeros sean conscientes del contenido de las notificaciones de este servicio (por ejemplo cuando le recuerda que no se olvide de tomar algún tipo de pastilla). Por tanto, las notificaciones del servicio deberían adaptarse al nivel de molestia 2 (*slightly-proactive*), adaptación que el usuario conseguirá a base de devolver *feedback* negativo a las notificaciones que se den en otro nivel distinto.

Así pues, vamos a representar el siguiente escenario:

- El usuario estará satisfecho con el diseño inicial del servicio durante 39 interacciones.
- A partir de la 40 notificación, el usuario habrá cambiado de idea, y empezará a devolver *feedback* negativo, intentando reducir el nivel de molestia.

Con este objetivo, el usuario interacciona con el sistema a base de dar *feedback* positivo hasta la interacción 39 y posteriormente *feedback* negativo hasta lograr estabilizar el servicio en el nivel deseado, de manera análoga a la forma presentada en el apartado anterior.

Estas interacciones, dan como resultado la sucesión de espacios de molestia que representamos en la tabla 10, según una inicialización de los parámetros del algoritmo (presentados en apartado 5.3.3) de

$\alpha = 0.5$, $\gamma = 0.99$ y $\Delta = 3$:

Notificación	Mandada por estado	FEEDBACK	ESTADO	Mantener	Cambiar
1	3	+	1	0.0	0.1
			2	0.0	0.1
			3	0.1	0.0
2	3	+	1	0.0	0.1
			2	0.0	0.1
			3	0.5995	0.0
3	3	+	1	0.0	0.1
			2	0.0	0.1
			3	1.0965	0.0
4	3	+	1	0.0	0.1
			2	0.0	0.1
			3	1.591	0.0
5	3	+	1	0.0	0.1
			2	0.0	0.1
			3	2.083	0.0
6	3	+	1	0.0	0.1
			2	0.0	0.1
			3	2.5726	0.0
7	3	+	1	0.0	0.1
			2	0.0	0.1
			3	3.0	0.0
8	3	+	1	0.0	0.1
			2	0.0	0.1
			3	3.0	0.0
…	IDEM	IDEM	IDEM		
39	3	+	1	0.0	0.1
			2	0.0	0.1
			3	3.0	0.0
40	3	-	1	0.0	0.1
			2	0.0	0.1
			3	2.485	0.0
41	3	-	1	0.0	0.1
			2	0.0	0.1
			3	1.9726	0.0
42	3	-	1	0.0	0.1
			2	0.0	0.1
			3	1.4627	0.0
43	3	-	1	0.0	0.1
			2	0.0	0.1
			3	0.9554	0.0
44	3	-	1	0.0	0.1
			2	0.0	0.1
			3	0.4506	0.0
45	3	-	1	0.0	0.1
			2	0.0	0.1
			3	-0.0517	0.0
46	2	+	1	0.0	0.1
			2	0.0	0.1
			3	-0.0517	0.5495

47	3	-	1	0.0	0.1
			2	0.0	-0.178
			3	-0.0517	0.5495
48	2	+	1	0.0	0.1
			2	0.0	-0.178
			3	-0.0517	0.7748
49	2	+	1	0.0	0.1
			2	0.5	-0.178
			3	-0.0517	0.7748
50	2	+	1	0.0	0.1
			2	0.9975	-0.178
			3	-0.0517	0.7748
51	2	+	1	0.0	0.1
			2	1.4925	-0.178
			3	-0.0517	0.7748
52	2	+	1	0.0	0.1
			2	1.985	-0.178
			3	-0.0517	0.7748
53	2	+	1	0.0	0.1
			2	2.4751	-0.178
			3	-0.0517	0.7748
54	2	+	1	0.0	0.1
			2	2.9627	-0.178
			3	-0.0517	0.7748
55	2	+	1	0.0	0.1
			2	3.0	-0.178
			3	-0.0517	0.7748
56	2	+	1	0.0	0.1
			2	3.0	-0.178
			3	-0.0517	0.7748
…	IDEM	IDEM	IDEM		
99	2	+	1	0.0	0.1
			2	3.0	-0.178
			3	-0.0517	0.7748

Tabla 10 – Espacio de adaptación de molestia durante las iteraciones.

En esta sucesión de interacciones, se pueden remarcar los siguientes hitos:

- En la iteración número 7 se alcanza el valor umbral para el peso de la acción mantener del estado 3, y en la iteración número 55 se alcanza el de la acción mantener del estado 2.
- De los estados 8 al 39, se repiten las mismas condiciones y el mismo espacio de adaptación de molestia. Lo mismo sucede del estado 56 al 99.
- En la iteración número 48 se consigue el objetivo, establecer el nivel 2 como nivel de molestia del servicio "*healthcare*". Como se puede apreciar en el estado 46, previamente consigue cambiarlo momentáneamente, pero vuelve a saltar a su antiguo estado, ya

que se necesitan una serie de “saltos de ajuste” para que el algoritmo aprenda el nivel deseado y lo estabilice.

8.2 Evaluación experimental. Calidad del comportamiento

La tabla 10 representa la traza para las primeras 99 interacciones entre el servicio “*healthcare*” y un usuario del sistema que se comporte según los objetivos presentados en el apartado anterior, devolviendo *feedback* positivo o negativo consistentemente para alcanzar dichos objetivos, que simplificando eran:

- Estar satisfecho con el nivel inicial de molestia del servicio las primeras 39 interacciones
- Después, al haber conseguido un trabajo, cambia de opinión y quiere disminuir el nivel de molestia.

Para nuestra evaluación experimental, será el propio usuario con su *feedback*, quien califique el comportamiento aprendido, indicando si la manera de recibir la notificación era la que él tenía en mente o no. La *calidad de un comportamiento* será su correspondencia con lo que espera el usuario.

La figura 45 muestra los valores de calidad que resultan del comportamiento del servicio “*healthcare*”, basados en el *feedback* dado por el usuario para la traza presentada en las tablas 6-10:

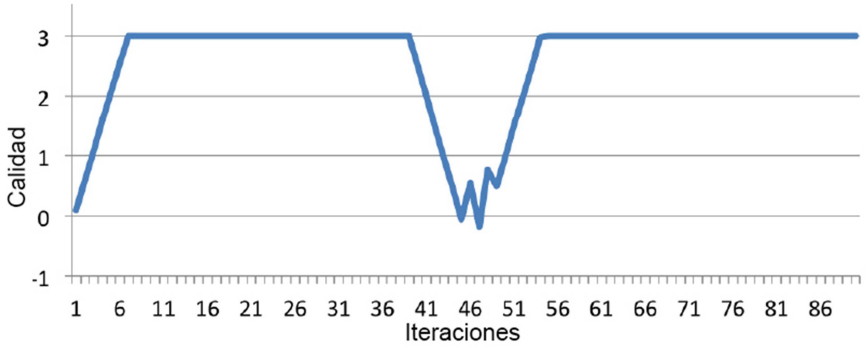

Figura 45 – Valores de calidad del comportamiento del servicio “healthcare”.

Estos valores de calidad revelan la correspondencia entre el comportamiento del sistema con el comportamiento que de verdad quiere el usuario (sus preferencias). En esta gráfica, la curva es ascendente (o plana, debido a que alcanza el umbral Δ y no puede crecer más) mientras el comportamiento aprendido está acorde con lo que quiere el usuario, y este devuelva *feedback* positivo (desde la iteración 1 hasta la 39).

Después, la calidad disminuye cuando el usuario cambia de opinión sobre el comportamiento deseado y empieza a devolver feedback negativo (iteraciones de la 40 a la 45). En ese momento, la calidad empieza a fluctuar mientras el sistema está prendiendo el nuevo nivel deseado por el usuario (iteraciones desde la 46 hasta la 48).

A partir de este momento, el usuario ya está satisfecho con el nuevo comportamiento conseguido, y la curva vuelve a subir (o mantenerse plana al volver a alcanzar el umbral) (iteraciones de la 49 hasta el final).

A raíz de estos resultados, podemos afirmar que nuestro sistema, como ya descubrimos en el capítulo de trabajos relacionados al compararlo con las otras soluciones, no necesita de un periodo previo de aprendizaje, aprendiendo sin ninguna información previa desde el primer momento en que el usuario empieza a enviar su *feedback*, adaptando el nivel de molestia de los servicios con un pequeño coste en cuanto a número de pasos para cada transición de nivel (en nuestro ejemplo, el representado con las iteraciones 40-48).

8.3 Conclusiones

En la presentación del caso de estudio se ha mostrado la ***interfaz gráfica del sistema*** desarrollada en esta tesis, descubriendo sus diferentes áreas (área de notificaciones, área de *feedback*, etc.), así como las maneras en que el usuario puede interactuar con los servicios a través de estas áreas.

Para ilustrar mejor todo el proceso de como el usuario utiliza el sistema, se ha presentado un caso práctico, donde se empieza realizando una ***carga del sistema*** (importando los servicios de un sistema existente previamente), después se ha procedido a la ***creación de un nuevo servicio***, y posteriormente, en base a un escenario de supuestas acciones en la mente de un usuario, se ha mostrado como este usuario interactúa con el sistema consiguiendo una adaptación satisfactoria del *nivel de molestia* de los servicios.

Se han detallado las posibilidades que tiene el usuario de interactuar con el sistema (al detalle de que botones debe pulsar, que indicadores o áreas de información posee, etc.), mostrando la secuencia de acciones que seguirá para realizar el caso práctico presentado.

Como demostración de la información interna de los pesos y los valores de calidad que maneja el algoritmo de aprendizaje, se presenta una traza para las 99 interacciones del caso práctico, donde se detalla el *espacio de adaptación de molestia* del servicio en ejecución para cada una de esas interacciones entre ese servicio y el usuario.

Por último, se ha presentado la noción de ***calidad del comportamiento***, y en base a la traza de interacciones realizada, se ha realizado una evaluación de la calidad del comportamiento para ese ejemplo.

CAPÍTULO 9

Conclusiones y trabajo futuro

Ya presentado en detalle el desarrollo del trabajo realizado en esta tesis, no queda más que exponer unas conclusiones acerca de dicho trabajo, así como aquellos puntos por donde podría seguir ampliándose esta tesis en forma de trabajos futuros.

9.1 Conclusiones

En esta tesis se ha desarrollado un sistema para adaptar automáticamente el *espacio de adaptación de molestia* de los servicios, basado en un algoritmo de aprendizaje por refuerzo, el algoritmo ***Q-Learning***.

Dicho sistema permite adaptar en tiempo de ejecución el mencionado *espacio de adaptación de molestia* de un servicio, predefinido en tiempo de diseño, mediante un aprendizaje del nivel adecuado, a base de ensayo y error.

Este aprendizaje se consigue explotando el ***feedback*** que los usuarios devuelven como respuesta a una notificación de un servicio. Su respuesta será una recompensa (*feedback* positivo) cuando la interacción se realice de una manera satisfactoria para el usuario, y un castigo (*feedback* negativo) cuando no sea así.

De esta forma, y gracias a esta explotación del *feedback* y al refinamiento del *espacio de adaptación de molestia*, se consigue una

minimización de la carga de las notificaciones y un aumento de la satisfacción del usuario.

9.2 Trabajo futuro

El trabajo realizado en esta tesis, como se expuso en el apartado 1.5 sobre el contexto de la tesis, brinda una solución satisfactoria a una necesidad funcional requerida por parte de otro sistema mayor, desarrollado por el *Centro de Investigación en Métodos de Producción de Software (ProS)*[1] de la *Universitat Politècnica de València*[2].

A continuación se presentarán una serie de desarrollos que podrían formularse como una ampliación al presente trabajo, y que quedan introducidos en forma de trabajos futuros.

9.2.1 Integración del sistema

Esta tesis se ha realizado como ampliación de funcionalidad de un sistema preexistente, un sistema "padre". Concretamente, un sistema de auto-adaptación de preferencias en dispositivos ubicuos, que sabiendo adaptar la molestia de los servicios ante cambios de contexto mediante reglas predefinidas, necesitaba una solución para conseguir una adaptación de dicha molestia en tiempo de ejecución que fuera no dependiente de ningún modelo preestablecido, sino que se basase en el *feedback* del usuario, ajustando la molestia a base de "ensayo y error".

Esta solución ha sido desarrollada en nuestra tesis, en forma de un sistema autónomo, que sabe cargar los modelos de espacios de molestia utilizados por el sistema padre, y que desarrolla todo el proceso de adaptación del espacio de molestia en base a ese *feedback*

[1] http://www.pros.upv.es
[2] http://www.upv.es

del usuario, siguiendo el algoritmo de aprendizaje por refuerzo marcado por los requerimientos de dicho sistema padre.

Por lo tanto, el siguiente paso lógico sería integrar este nuevo sistema en su sistema padre, llevando a cabo las adaptaciones necesarias para utilizar el interfaz de usuario del padre, o en caso de preferir que continuase actuando como un sistema autónomo con su propia interfaz, que tuviera la funcionalidad de, una vez acabado el proceso de adaptación del *espacio de adaptación de molestia* de un servicio, devolver dicho espacio adaptado al sistema padre en forma de modelos similares a los que ha cargado.

9.2.2 Mejora del algoritmo de aprendizaje mediante la implantación de métodos estadísticos de probabilidad

Futuras ampliaciones del sistema desarrollado en esta tesis, podrían hacer uso de métodos estadísticos de probabilidades para intentar mejorar los tiempos de adaptación del espacio de molestia.

Al igual que se almacena el *feedback* recibido para posteriormente procesarlo cada vez que haya que hacer un cambio de nivel, y así saber si se debe incrementar o disminuir dicho nivel, podría almacenarse también información referente a que cantidad de veces ha tenido que recibir un servicio *feedback* negativo para indicar un deseo verdadero de cambiar de nivel por parte de un usuario determinado.

Así pues, para un usuario concreto, esta información podría dar una idea de que el usuario desea cambiar el nivel de un servicio de manera más rápida a como la obtiene el algoritmo de aprendizaje.

Por ejemplo, si vamos almacenando este tipo de información, de manera que sabemos que cada vez que el usuario “x” emite 4 *feedbacks* negativos se traduce en un cambio de nivel con una posibilidad de, digamos, un 90%, si en un determinado momento el usuario empieza a mandar *feedback* negativo, aunque los valores de

calidad del algoritmo se estén ajustando y aún falten bastantes iteraciones para provocar un cambio de nivel, podríamos aventurarnos y cambiar automáticamente el nivel simplemente con recibir 4 *feedbacks* negativos. La adaptación ganaría en rapidez y, aunque podría perder en precisión, siempre tendríamos la opción de, en el caso de habernos equivocado con el cambio de nivel, arreglarlo esta vez sí con el propio funcionamiento "normal" del algoritmo de aprendizaje *Q-Learning*.

Ampliando la idea, de forma análoga a como podríamos procesar información para un usuario determinado, podíamos también enfocarlo en guardar y procesar información para cada servicio, de manera que pudiéramos manejar información del tipo "cuando a un servicio "x" se le empieza a dar *feedback* negativo, la probabilidad de que cambie de nivel de molestia es de "y"%".

9.2.3 Aprendizaje de patrones de comportamiento

De la misma forma que podemos aprender probabilidades para un servicio o un usuario determinado, se podría intentar aprender patrones de conductas de usuario. Con cada *feedback* de usuarios concretos, se podrían ir aprendiendo patrones de comportamiento ante tipos similares de interacciones, que posteriormente nuestra solución podría aplicar globalmente a cualquier usuario, uniendo la información contenida en estos patrones con la información que nos proporciona el algoritmo de aprendizaje por refuerzo.

Referencias

[1] Weiser, Mark. "The computer for the 21st century." *Scientific american* 265.3 (1991): 94-104.

[2] http://mashable.com/2013/07/18/google-play-50-billion-apps/

[3] http://www.thetechgets.com/2013/05/apples-app-store-surpasses-50billion.html

[4] Chen, Hao, and James P. Black. "A quantitative approach to non-intrusive computing." *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2008): 44.

[5] Gibbs, W. Wayt. "Considerate computing." *Scientific American* 292.1 (2005): 54-61.

[6] Ju, Wendy, and Larry Leifer. "The design of implicit interactions: Making interactive systems less obnoxious." *Design Issues* 24.3 (2008): 72-84.

[7] http://www.javahispano.org/storage/contenidos/OSGI_Roberto_Montero.pdf

[8] http://www.ibm.com/developerworks/opensource/tutorials/os-rcp1/section3.html

[9] http://www.dsic.upv.es/docs/bib-dig/informes/etd-01192006-130355/DSIC-II-02-06_Tech_Report.pdf

[10] Abowd, Gregory D., et al. "Towards a better understanding of context and context-awareness." *Handheld and ubiquitous computing*. Springer Berlin Heidelberg (1999).

[11] Rosenthal, Stephanie, Anind K. Dey, and Manuela Veloso. "Using decision-theoretic experience sampling to build personalized mobile phone interruption models." *Pervasive Computing*. Springer Berlin Heidelberg (2011): 170-187.

[12] Byun, Hee Eon, and Keith Cheverst. "Exploiting user models and context-awareness to support personal daily activities." *Workshop in UM2001 on User Modeling for Context-Aware Applications* (2001).

[13] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. Vol. 1. No. 1. Cambridge: MIT press (1998).

[14] Gil, Miriam, and Vicente Pelechano. "Exploiting user feedback for adapting mobile interaction obtrusiveness." *Ubiquitous Computing and Ambient Intelligence*. Springer Berlin Heidelberg (2012):274-281.

[15] Gil, Miriam, Pau Giner, and Vicente Pelechano. "Personalization for unobtrusive service interaction." *Personal and Ubiquitous Computing* 16.5 (2012): 543-561.

[16] Gil, Miriam. "Adapting interaction obtrusiveness : Making ubiquitous interactions less obnoxious." *PhD thesis (2003)*.

[17] Watkins, Christopher JCH, and Peter Dayan. "Q-learning." *Machine learning* 8.3-4 (1992): 279-292.

[18] Learning from User Feedback for Adapting Interaction Obtrusiveness: A Case Study.
http://www.pros.upv.es/index.php/en/60-spanish-root-es-es/1209-learning

[19] Chittaro, Luca. "Distinctive aspects of mobile interaction and their implications for the design of multimodal interfaces." *Journal on Multimodal User Interfaces* 3.3 (2010): 157-165.

[20] Schmidt, Albrecht, et al. "interacting with 21st-Century Computers." *Pervasive Computing, IEEE* 11.1 (2012): 22-31.

[21] Tedre, Matti. "What Should Be Automated? The fundamental question underlying human-centered computing". *In Proceedings of the 1st ACM international workshop on Human-centered multimedia, HCM '06* (2006): 19-24.

[22] Ferscha, Alois. "20 Years Past Weiser: What's Next?". *Pervasive Computing, IEEE 11.1* (2012): 52-61

[23] Dix et al. "Human-Computer Interaction" (3rd Edition), Prentice Hall, New York, NY,USA (2003).

[24] Schmidt, Albrecht. "Implicit human computer interaction through context." *Personal Technologies* 4.2-3 (2000): 191-199.

[25] Schmidt, Albrecht. "Context-aware computing: context-awareness, context-aware user interfaces, and implicit interaction." *The Encyclopedia of Human-Computer Interaction, 2nd Ed.* (2013).

[26] Horvitz, Eric, et al. "Models of attention in computing and communication: from principles to applications." *Communications of the ACM* 46.3 (2003): 52-59.

[27] Godoy, Daniela, and Analia Amandi. "User profiling for web page filtering." *Internet Computing, IEEE* 9.4 (2005): 56-64.

[28] Schiaffino, Silvia, and Analia Amandi. "Polite personal agent." *Intelligent Systems, IEEE* 21.1 (2006): 12-19.

[29] http://javacuriosities.blogspot.com.es/2010/05/osgi-first-step.html

[30] Doctor, Faiyaz, Hani Hagras, and Victor Callaghan. "A fuzzy embedded agent-based approach for realizing ambient intelligence in intelligent inhabited environments." *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 35.1 (2005): 55-65.

[32] Wang, L-X., and Jerry M. Mendel. "Generating fuzzy rules by learning from examples." Systems, Man and Cybernetics, IEEE Transactions on 22.6 (1992): 1414-1427.

[37] Pérez, Tomás. Apuntes del curso de desarrollo de aplicaciones en java. UPV (2012).

[38] Apuntes de Ingeniería del Software de Gestión. DSIC, UPV (2009)