El Comportamiento Subjetivo en Aplicaciones Context-Aware

Tesis de Grado

Facultad de Informática
Universidad Nacional de La Plata

Betiana Darderes
betiana.darderes@lifia.info.unlp.edu.ar

Director: Dra. Silvia Gordillo

La Plata - Buenos Aires
Argentina
Noviembre de 2006

A todos los que hicieron cosas por mí mientras escribía.

Índice

Capítulo 1

INTRODUCCIÓN

1.1. MOTIVACIÓN

En los lenguajes Orientados a Objetos tradicionales, el mecanismo de adaptación del comportamiento más conocido es el uso de sentencias *if-case*. Dichas sentencias evalúan ciertas condiciones del mensaje y resuelven qué pedazo de código ejecutar. El principal problema de este mecanismo es que las condiciones están embebidas en la implementación del mensaje, junto con la implementación de los diferentes comportamientos. Esto provoca que las respuestas de un mensaje no sean independientes y que las condiciones estén definidas a nivel de implementación. Por otra parte, el excesivo uso de dichas sentencias hace que el código resultante sea poco claro y difícil de mantener.

Existen algunos lenguajes orientados a objetos, como CLOS [BOB/86] y CommonLisp [BOB/88], que usan multimétodos para adaptar el comportamiento de los mensajes. Este mecanismo elige el método a ejecutar dependiendo del tipo o de la clase del receptor y de los argumentos del mensaje. La principal ventaja que posee es que las implementaciones del mensaje son independientes. Otro mecanismo de adaptación propuesto, *predicate dispatching* [ERN/98], aumenta la adaptación de multimétodos ya que además elige la implementación del mensaje según la evaluación de predicados que se construyen a partir de componentes de los parámetros del mensaje o del receptor. De todas formas, la adaptación que provee éste último mecanismo sigue siendo limitada, ya que por ejemplo no es posible adaptar la respuesta del mensaje según propiedades del emisor o de

cualquier otro objeto del ambiente que influencie de alguna manera el comportamiento del receptor. Esta incapacidad complica el diseño y la implementación de aplicaciones en las cuales es necesario que el contexto influencie el comportamiento de los objetos. Tomemos como ejemplo de una aplicación inspirada en una tesis realizada sobre *context-awareness* [DEY/00]. Supongamos que tenemos una pequeña aplicación que corre en un dispositivo móvil. Dicha aplicación tiene como requerimiento funcional principal mandar mensajes a través de una red. Supongamos que dentro del modelo tenemos que objetos que representan al usuario registrado al dispositivo y a los usuarios presentes en la misma habitación. Además dicho usuario registrado interactúa con un objeto *messenger* encargado de mandar mensajes a otros dispositivos conectados a la red de comunicaciones. Supongamos que un usuario manda un mensaje de texto invocando el mensaje *#deliver:aMensaje* del *messenger*, y que luego este último reenvía el mensaje a la red de comunicaciones. La red de comunicaciones esta representada por el objeto *network*, que registra el estado de la red y conoce el protocolo para mandar mensajes a través del medio de comunicación. En la figura 1.1 vemos un diagrama con la estructura de la aplicación descripta. Todos los diagramas realizados en esta tesis utilizan la sintaxis UML [BOO/98].

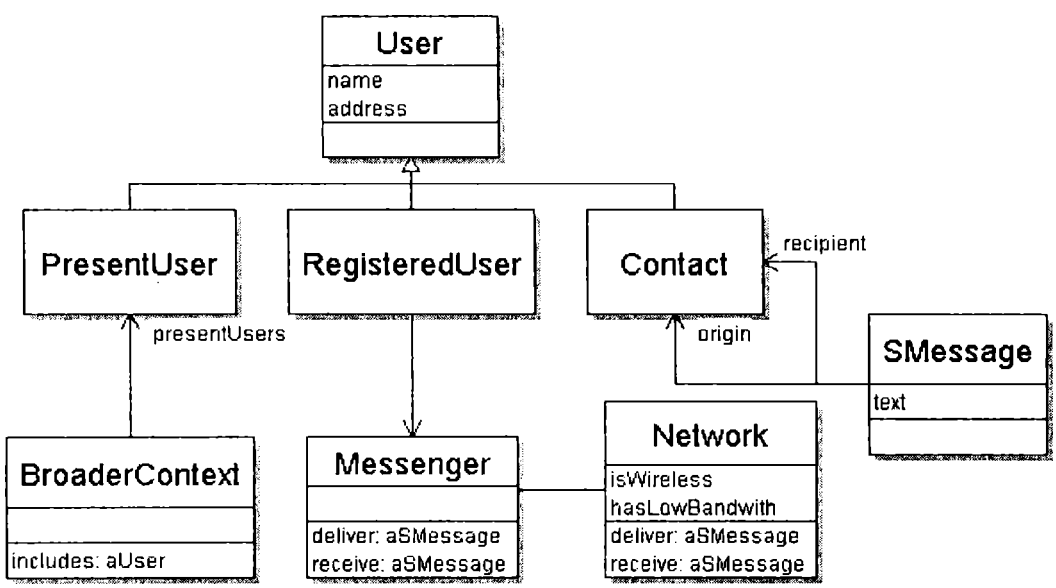

Figura 1.1: Estructura de una aplicación que envía mensajes a través de una red.

Supongamos que además del requerimiento funcional de mandar mensajes, es necesario que la aplicación cumpla con los siguientes requerimientos:

(1) Que los mensajes sean encriptados, si la red de comunicaciones es inalámbrica;
(2) que sean compactados, si la red tiene poco ancho de banda;
(3) que sean enviados con mayor prioridad a los usuarios presentes en la habitación;
(4) o que sean enviados sólo si el emisor del mensaje está autorizado.

Estos nuevos requerimientos están relacionados con el requerimiento funcional de mandar mensajes, ya que varían la forma de realizar el envío. Observemos, por ejemplo, el requerimiento (3). En este caso, el *messenger* no cuenta con toda la información necesaria para saber si el destinatario del mensaje está presente en la habitación, ya que precisa conocer a los usuarios presentes. Si miramos nuevamente la figura 1, notamos que no existe ninguna relación que lo vincule con esta información. Esto sucede porque el modelo fue hecho teniendo en cuenta únicamente los requerimientos funcionales de la aplicación. Ahora, con los nuevos requerimientos, el *messenger* deberá analizar el contexto del objeto, es decir, los objetos que lo circundan, además de sus colaboradores habituales. Por lo tanto, para resolver el problema es necesario agregar nuevas referencias. Analicemos brevemente los problemas que esto puede llegar a ocasionar. Está claro que para implementar los requerimientos (3) y (4), es necesario que el receptor tenga dos nuevas referencias para construir las condiciones: una referencia al emisor del mensaje y otra al objeto *boarderContext* que registra los usuarios presentes (Ver Figura 1.2). Al sumar estas dos referencias dentro del modelado orientado a objetos tradicional, el *boaderContext* y el emisor se convierten en colaboradores del *messenger*. Sin embargo, estas nuevas referencias sólo se utilizarán para implementar condiciones del mensaje, mientras que el comportamiento de envío seguirá siendo realizado con la colaboración exclusiva del objeto *network*. Por lo tanto, estas nuevas relaciones de colaboración advenidas por el requerimiento (3) y (4) se suman a los elementos de diseño de los requerimientos funcionales, sin otorgar valor funcional real, ya que no intervienen en el desarrollo de la repuesta del mensaje, sino que implementan la lógica que decide la respuesta. No distinguir entre estos dos tipos de relaciones entre objetos puede aumentar la confusión al momento de mantener la aplicación, sobre todo en aplicaciones donde el cambio de requerimientos vinculados al contexto es frecuente [KOR/00].

Por otra parte, para obtener la referencia al emisor en el momento de la recepción del mensaje, es necesario modificar la signatura del mismo (e.g. *messenger deliver: aSMessage forSender: self*). Esto es inconveniente si más adelante cambia el requerimiento de adaptación y no es necesario analizar más la identidad del emisor. En ese caso el parámetro pierde sentido y, al borrarlo, las invocaciones al mensaje se vuelven inválidas. Otro problema es que el emisor pierde su rol en el contexto de recepción para convertirse en colaborador externo (i.e. parámetro del mensaje). De esta forma, un emisor no autorizado podría lograr el envío invocando el mensaje con un parámetro autorizado en el lugar del emisor.

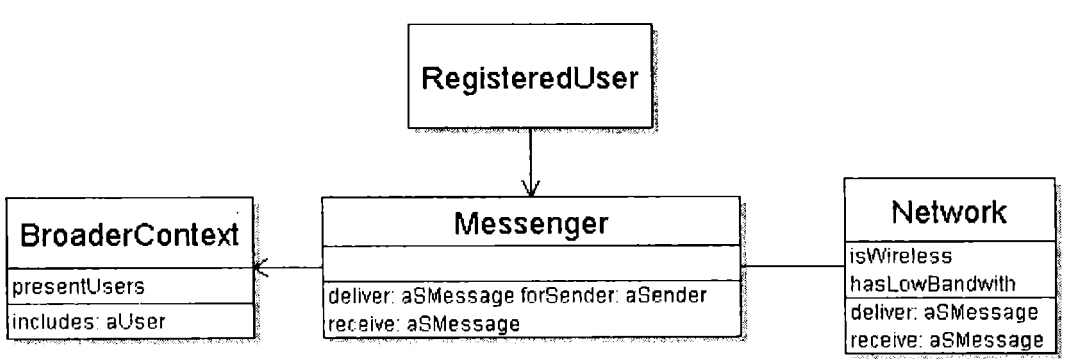

Figura 1.2: Estructura con la referencia al BroaderContext y al emisor del mensaje.

En contrapartida, el comportamiento subjetivo separara los dos tipos de relaciones entre objetos: *influyente* y *colaborador*. Aquellos objetos que implementen condiciones serán influyentes del receptor, dejando como colaboradores a los que realmente realicen alguna acción que forme parte de la respuesta al mensaje. A su vez, el comportamiento subjetivo libera el alcance de los objetos influyentes, permitiendo que cualquier objeto de la aplicación implemente las condiciones del mensaje. Además, la lógica de respuesta del comportamiento subjetivo propuesto por esta tesis conserva las ventajas de otros mecanismos de adaptación. No sólo fuerza a definir cada respuesta del mensaje en un método distinto, permitiendo la total independencia de las posibles respuestas, sino que además conserva sencillez y permite que una misma respuesta pueda ejecutarse para distintas condiciones de recepción. Esto trae como ventaja la reutilización de código a nivel de mensaje. La intención de estos nuevos recursos es facilitar el diseño de aplicaciones cuyo comportamiento es altamente influenciado por el contexto.

En lo que resta del capítulo repasaremos los diferentes mecanismos de adaptación de comportamiento propuestos por la Orientación a Objetos. Además se dará un breve resumen de los trabajos hechos en subjetividad y finalmente se enunciará el objetivo de la tesis y su organización.

1.2. DESPACHO SIMPLE DE MENSAJES Y MULTIMÉTODOS

El mecanismo de despacho de mensajes simple generalmente esta basado en clases. Si el *binding* es dinámico, cuando un mensaje es invocado, su método asociado se busca en la clase de un argumento del mensaje. Si no se encuentra ningún método asociado, se busca en la superclase de dicho argumento y así sucesivamente. Este mecanismo de adaptación de comportamiento tiene en cuenta la clase de un argumento para decidir la respuesta del mensaje. Además, es externo a la implementación del mensaje y no puede modificarse por el diseñador, ya que forma parte de la implementación del lenguaje subyacente.

Smalltalk [GOL/98] es un lenguaje orientado a objeto con despacho dinámico simple y basado en clases. Cuando un mensaje es invocado su método asociado se busca en la clase de un argumento especial del mensaje, llamado receptor. En la sintaxis Smalltalk este argumento especial se indica antes de hacer la llamada (e.g. *specialArgument messageWith: anArgument and: otherArgument*).

Si tenemos un conjunto de objeto polimórficos, la respuesta de los mensajes depende de la clase del receptor y es única. Las diferentes respuestas de un mismo mensaje se encapsulan en diferentes clases y son independientes una de otras. La figura 1.3 muestra un conjunto de clases de instancias polimórficas. Cada clase de estrategia sabe responder al mismo mensaje, pero lo implementa de diferente manera. Por lo tanto, el comportamiento del mensaje *#doIt* depende de la clase del receptor.

En los lenguajes con despacho de mensajes múltiple (i.e. multi-methods) todos los argumentos son tratados simétricamente en la elección del método a aplicar. En el momento de la invocación se analizan todos los argumentos para determinar el método que debe ejecutarse. Por ejemplo, en *CommonLisp* los métodos o funciones son despachados dinámicamente dependiendo del tipo de los argumentos.

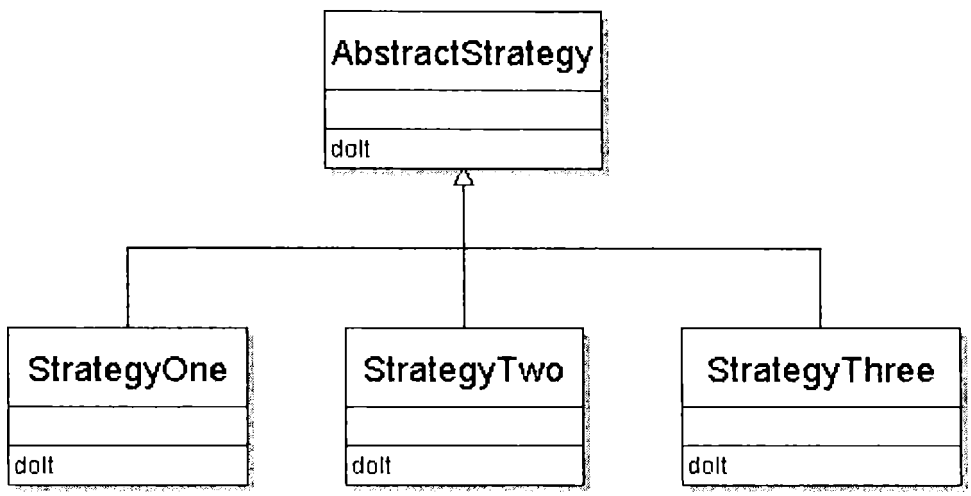

Figura 1.3: Jerarquía de clases de instancias polimórficas.

Cuando se trabaja con lenguajes que discriminan tipo de datos en tiempo de compilación la selección de alternativas puede ocurrir en tiempo de compilación. El acto de crear esas funciones alternativas en tiempo de compilación se llama usualmente sobrecargar una función.

En los lenguajes que difieren la identificación del tipo de dato hasta el tiempo de ejecución, la selección de las funciones alternativas puede ocurrir en tiempo de ejecución basándose en la determinación dinámica de los tipos de los argumentos. Las funciones cuya alternativa de implementación se selecciona de esta manera se llaman generalmente multimétodos.

En algunos lenguajes la distinción entre sobrecarga de mensajes y multimétodos es difusa, dado que el compilador determina cuando se puede aplicar la selección en tiempo de compilación o cuando es necesario un despacho de mensajes en tiempo de ejecución. Esto se debe a que existe cierto costo de ejecución asociado el despacho de mensajes dinámico, incluso algunos lenguajes ofrecen la opción de suspender el despacho dinámico si es necesario.

1.3. DESPACHO POR PREDICADO

En el mes de Julio de 1998 se publica en ECOOP el trabajo "Predicate Dispatching: A Unified Theory of Dispatch"[ERN/98]. En este paper describe una forma de mecanismo de despacho de mensajes basado en predicados. El despacho

por predicado generaliza los mecanismos de despacho de mensajes vistos al momento, permitiendo que predicados arbitrarios controlen la aplicación de métodos y también usa las implicaciones lógicas cómo relaciones de sobre escritura. En este caso, el método ejecutado, cuando se invoca el mensaje, no sólo depende de las clases de los argumentos, si no que también de las subcomponentes de las clases, del estado de los argumentos y de las relaciones entre los objetos. Una declaración de un método especifica su aplicabilidad mediante una expresión de predicados. Esta expresión es una fórmula lógica compuesta de un testeo de clase y de una expresión booleana arbitraria. Un método es aplicable cuando su expresión de predicado evalúa verdadera. A su vez, un método m1 sobrescribe un método m2, cuando el predicado lógico de m1 implica lógicamente él de m2.

1.4. PROGRAMACIÓN MODAL

En el mes de Junio de 1993 Antero Taivalsaari escribe su artículo denominado "Object-oriented programming with modes" [TAI/93]. En el mismo, plantea el concepto de "modos"(o "estados lógicos") en los que se puede encontrar un objeto y deja plasmada teóricamente una posible extensión del modelo convencional orientado a objetos basado en clases que soporte la definición explícita de modos.
Taivalsaari expresa que durante el diseño y la implementación de programas, uno a menudo se encuentra con objetos que pueden alcanzar diferentes estados o condiciones, no refiriéndonos a los valores de las variables de dicho objetos, sino a una clase de estado más conceptual. Por ejemplo, en una interfaz gráfica las ventanas pueden estar abiertas, cerradas, representadas como íconos, etc. A estos estados lógicos son a los que Taivalsaari denomina "modos". Veamos más detenidamente su planteo.

Él pudo observar que la presencia de "modos" en los programas es muy frecuente y rara vez se le presta atención, aunque es obvio que tales estados lógicos son importantes en el diseño y la implementación de los programas, ya que los métodos suelen variar considerablemente dependiendo del modo en el que se encuentren. Y además de ser esenciales en la implementación de abstracciones, los "modos" son claramente esenciales en el diseño, especificación y análisis de las componentes del programa. Sin embargo, aunque la programación orientada a objetos enfatiza la importancia de la abstracción, la potencialidad de los modos había sido ignorada hasta ese momento, ya que casi todos los lenguajes orientados a objetos hasta ese entonces forzaban a que la información de los estados lógicos de los objetos esté oculta en las operaciones de dichos objetos (por ejemplo, incluyendo sentencias *if-case* para testear algún estado interno del objeto y decidir en base a ello qué rama de operaciones ejecutar). Esto provoca que las operaciones incluyeran considerables construcciones condicionales con el solo propósito de asegurar que se elija la porción adecuada de comportamiento en una situación particular. Esto no era muy satisfactorio, ya que el uso excesivo de sentencias condicionales hacía que las definiciones de las operaciones tendieran a ser más largas y complicadas de leer y mantener, haciendo decrecer la claridad

conceptual de los programas y forzando a que la información esencial de diseño se ocultara en la implementación de las operaciones.

1.5. TRABAJOS REALIZADOS EN SUBJETIVIDAD

En Septiembre de 1993 durante la conferencia "Object-Oriented Programming Systems, Languages and Applications" (OOPSLA) en Washington, DC; William Harrison y Harold Ossher presentan su paper denominado "Subject-Oriented Programming (A Critique of Pure Objects)" [HAR/93]. Este trabajo marca el comienzo explícito del tema de "Subjetividad". En este paper los autores proponen la denominada "Programación Orientada a Sujetos", cuyo objetivo es facilitar el desarrollo y la evolución de los conjuntos de aplicaciones cooperantes. En este contexto son importantes los siguientes requerimientos:

- Debe ser posible desarrollar aplicaciones separadas y luego componerlas, siendo estas aplicaciones independientes una de otras no importando su grado de cooperación.
- Debe ser posible la introducción de una nueva aplicación a una composición sin requerir modificaciones a las otras aplicaciones integrantes de la composición, y sin invalidar los objetos persistentes ya creados por ellas
- Debe soportarse todas las paliaciones no previstas, inclusive las extensiones de las ya existentes
- Dentro de una aplicación deben conservarse las ventajas de encapsulamiento, herencia y polimorfismo.

En el paradigma orientado a sujetos descripto por Harrison y Ossher, cada aplicación es un sujeto o una composición de sujetos, un término que define una colección de especificaciones de estado y comportamiento asociados a una aplicación en sí. Estos sujetos reflejan una pequeña y delimitada percepción del mundo real, porque "son programas (o fragmentos de programas) orientados a objetos que modelan su dominio a su propio modo: de manera subjetiva", ya que modelan las diferentes "vistas subjetivas" que tiene sobre los objetos.

Un sujeto es una colección de especificaciones de estado y comportamiento que reflejan una estructura particular, una percepción del mundo, tal como las que son vistas por una aplicación o por una herramienta en particular. Este modelo subjetivo está definido por la jerarquía de clases del sujeto, que contiene solamente los detalles implementados y/o utilizados por el sujeto.

Bajo este modelo, un sujeto es un paquete que contiene la información de varias clases de objetos que trabajan juntos para alcanzar un objetivo en particular. Cada sujeto provee sus propias definiciones de clases, reflejando el punto de vista sobre el cual se basa el sujeto.

Los sujetos pueden componerse y producir nuevos sujetos y el comportamiento funcionales del nuevo sujeto esta controlado por la especificación de una serie de reglas que describen las correspondencias y la forma de composición de los

elementos que componen al sujeto. Al momento de componer sujetos deben examinarse cuidadosamente, junto con sus interrelaciones, para determinar cómo serán combinados. Esto incluye: correspondencia de operaciones, clases, de variables de instancias y de métodos. Existen justamente dos tipos básicos de reglas:

- reglas de correspondencia: especifica el significado de los nombre en el sujeto compuesto
- reglas de composición: establece cómo se combinan los correspondientes elementos de los sujetos que forman parte de la composición del nuevo sujeto.

Bajo el modelo propuesto por Harrison y Ossher , la Programación Orientada a Sujetos permite acomodar la característica del mundo real en la cual diferentes sujetos clasifican a los objetos en diferentes jerarquías. Por ejemplo, el pájaro podría clasificar los objetos en plantas proveedoras de néctar, proveedoras de insectos). Mientras tanto, el leñador podría clasificar a los sujetos en árboles de madera dura y de madera blanda y no árboles. Estas clasificaciones representan formas diferentes de mirar las cosas.

La Programación Orientada a Sujetos propuesta por Harrison y Ossher intenta proveer una solución al manejo de estas situaciones instanciando sujetos y ruteando los mensajes en forma adecuada.

En el mes de Octubre de 1994 se lleva a cabo el primer workshop relacionado con el área de Subjetividad [HAR/94]. Su objetivo fue facilitar la discusión exploratoria de la Subjetividad e identificar algunos de los conceptos y conclusiones claves. David Ungar y Randall Smith resumieron el soporte para la subjetividad brindado por el lenguaje Us que permite a un objeto comportarse en forma diferente dependiendo de la "perspectiva" utilizada por el cliente.

En el mes de Octubre de 1995 se llevó a cabo el segundo workshop de subjetividad [HAR/95], que fue estructurado para cubrir una serie de tópicos: requerimientos de aplicaciones, principios, soporte de subjetividad en el mundo real, conexiones entre vistas subjetivas de un objeto e interacción de usuarios con Objetos Subjetivos.

En ese workshop Pablo Victory expuso el position paper "Real World Object Behavior" [PRI/95], sobre el cual se basa el comportamiento subjetivo explicado en esta tesis. Hubo un acuerdo generalizado en cuanto a que un despacho del estilo multi-método era una buena cosa para hacer, pero también hubo inquietudes acerca de la comprensibilidad del sistema resultante.

En 1996 Máximo Prieto y Pablo Victory publican su position paper "Subjectivity: Towards True Polymorphism" [PRI/96] continuando con su trabajo del año anterior y presentando una manera de extender el concepto de polimorfismo a nivel de objeto. Esta extensión se da en la subjetividad ya que los objetos subjetivos deben elegir una de varias implementaciones para contestar un mensaje.

En 1997 los mismos autores publican su artículo "Subjective Object Behavior" [PRI/97] donde amplían los conceptos de fuerzas y de desarrollo en un ambiente subjetivo, marcando las ventajas de la utilización de tales ambientes.

En 1999 Claudia Callegari, Laura Eliaschev y Carla Tanner, realizan su tesis de Licenciatura: "Subjetividad en un Ambiente de Objetos" [CAL/99]. Dicha tesis se basa en los trabajos realizados por Máximo Prieto y Pablo Victory. En este trabajo se analizó con más profundidad las diferentes fuerzas que influencian la respuesta de un mensaje, pero no se descartó que puedan existir otras fuerzas aún no detectadas. Además se analizó de qué manera influye la Subjetividad en los conceptos básicos de la Orientación a Objetos y se realizó una extensión que soporte la definición de objetos subjetivos.

En el año 2001 Kristensen presenta su paper "Subjective Behavior" [KRI/01] en donde se realiza una definición de comportamiento subjetivo. Kristensen dice que en el modelado orientado a objetos la descripción elegida por una invocación es determinada sólo por el tipo del objeto receptor. El comportamiento subjetivo significa que la elección depende de otros factores, como el objeto que realiza la invocación, los objetos del contexto, y el estado de los objetos. El también dice que la motivación del comportamiento subjetivo en este sentido es soportada por la idea de tener objetos más autónomos y con capacidad de evolución.

En 2002 Betiana Darderes, Andrea Grassano y Maximo Prieto presentan su position paper en OOPSLA "Subjective Behavior of Objects" [DAR/02]. En este trabajo se realiza un análisis más extenso y completo de las fuerzas que pueden influenciar las respuestas de un mensaje y se propone un nuevo mecanismo de despacho de mensajes basado en un árbol de decisión.

En 2003, Andrea Grassano realizan su tesis de Licenciatura:"Subjetividad en un Ambiente de Objetos" [GRA/03]. Dicha tesis se basa en el position paper presentado en 2002 y además define fuerzas de comportamiento en ambientes virtuales a partir de las fuerzas definidas para el comportamiento subjetivo.

En 2004, Betiana Darderes y Maximo Prieto presentan su position paper en OOPSLA [DAR/04]. En este trabajo se exploran todos los elementos que pueden influenciar la respuesta de un mensaje y compara el Comportamiento Subjetivo con otros mecanismos de adaptación de comportamiento. Concluye que el comportamiento permite que los objetos sean totalmente adaptables al cambio del contexto de recepción de un objeto. Además analiza ciertas características dinámicas obtenidas a partir de su utilización.

En 2005, Betiana Darderes y Maximo Prieto presentan su paper "Comportamiento Subjetivo: Objetos Adaptables a Cambios de Contexto" en la JAIIO [DAR/05]. Continuando con el trabajo del 2004, se aplica el comportamiento subjetivo al dominio de aplicaciones *context-aware* y se analizan los beneficios obtenidos al aplicar comportamiento subjetivo en el diseño de este tipo de aplicaciones.

1.6. OBJETIVO DE LA TESIS

Esta tesis se basa en los trabajos hechos en el 2004 y en el 2005. El principal aporte es analizar las consecuencias en el diseño orientado a objetos de la utilización comportamiento subjetivo.

Justificaremos el uso de subjetividad, si demostramos que favorece la construcción de diseños más compactos e igual de mantenibles, que los diseños logrados con los mecanismos de adaptación tradicionales.

Dado que el comportamiento subjetivo sólo es aplicable cuando es necesario variar el comportamiento de una aplicación según el contexto de ejecución, se utilizó el domino de aplicaciones *context-aware* para elegir una especificación que justificara la utilización de este tipo de comportamiento.

Otro aporte de esta tesis, es mejorar el mecanismo de respuesta. El árbol de decisiones expuesto en los trabajos previos sobre comportamiento subjetivo, separa las condiciones de un mensaje de sus implementaciones, pero no evita el uso de construcciones condicionales anidadas que dificultan la lectura del mensaje. El mecanismo expuesto por este trabajo reemplaza el árbol de decisión permitiendo organizar las implementaciones de manera secuencial y facilitando aún más la comprensión del mensaje y por lo tanto su mantenimiento.

En el siguiente capítulo definiremos qué es el comportamiento subjetivo y cuáles son los objetos que pueden influenciar la respuesta de un mensaje. Luego, en el capítulo 3, describiremos aspectos de implementación del prototipo de comportamiento subjetivo hecho por esta tesis. Este prototipo se realizo para verificar el funcionamiento del mecanismo de despacho de mensajes propuesto por esta tesis. En el capitulo 4 presentaremos un ejemplo de una aplicación *context-aware* y se realizará un diseño de sus requerimientos funcionales. En el capítulo 5 extenderemos el diseño agregando los requerimientos de adaptación al contexto utilizando un paradigma orientado a objetos tradicional. Luego en el capítulo 6 volveremos a agregar dichos requerimientos de adaptación pero aplicando comportamiento subjetivo. Finalmente, en el último capítulo, realizaremos conclusiones sobre el uso del comportamiento subjetivo en el diseño de requerimientos de adaptación al contexto.

Capítulo 2

COMPORTAMIENTO SUBJETIVO

2.1. INTRODUCCIÓN

Como vimos en la introducción, los mecanismos existentes para adaptar el comportamiento de un objeto tienen limitaciones. Por ejemplo, no permiten adaptar la respuesta del mensaje a cambios del emisor o a cambios de cualquier otro objeto del ambiente que influencie de alguna manera el comportamiento de la aplicación. También vimos que esta incapacidad puede complicar el diseño y el mantenimiento de aplicaciones en las cuales el contexto y las necesidades del usuario cambian constantemente [KOR/00].

En el caso de estudio de la introducción, vimos una aplicación que corre en un dispositivo móvil. Dicha aplicación, además de tener como requerimiento funcional mandar mensajes a través de la red, tenía que cumplir ciertos requerimientos de adaptación al contexto, por ejemplo, mandar los mensajes con mayor prioridad, si los destinatarios se encuentran presentes en la habitación. Dicho requerimiento afectaba el comportamiento del objeto *messenger,* quien debía analizar su contexto, es decir, los objetos que lo circundan, además de sus colaboradores habituales. Esta nueva capacidad del *messenger* era cubierta asociándole nuevos colaboradores capaces de implementar las condiciones del mensaje, por ejemplo el *broaderContext* que registra los usuarios presentes y el emisor del mensaje que determina la urgencia. Sin embargo, el *boaderContext* y el emisor sólo implementaban las condiciones del mensaje y el verdadero comportamiento siempre era realizado con colaboración exclusiva del objeto *network.* Por lo tanto, estas nuevas relaciones de colaboración advenidas por el requerimiento de adaptación, se sumaban a los elementos de diseño de los requerimientos funcionales sin otorgar verdadero valor funcional.

Con comportamiento subjetivo es más fácil definir e interpretar el diseño ya que podemos separar los dos tipos de relaciones. Aquellos objetos que implementen condiciones son *influyentes* del comportamiento del receptor, dejando como *colaboradores* a los que realmente realicen alguna acción que forme parte de la respuesta al mensaje. Además, el comportamiento subjetivo permite definir condiciones utilizando cualquier objeto del contexto de recepción del mensaje, librando la necesidad de nuevas relaciones.

También habíamos dicho que el comportamiento subjetivo conserva las ventajas de otros mecanismos de adaptación de comportamiento. Fuerza a definir cada respuesta del mensaje en un método distinto, permitiendo la independencia de las posibles respuestas y permite que varias respuestas apliquen en diferentes circunstancias, favoreciendo la reutilización de código a nivel de mensaje. La intención de estos nuevos recursos es facilitar el diseño de aplicaciones en donde el comportamiento es altamente influenciado por el contexto.

Este capítulo tiene como objetivo definir qué es el comportamiento subjetivo. Se dará una definición de lo que llamamos requerimiento de adaptación y se analizarán las diferencias que tienen con respecto a los requerimientos funcionales. Luego se dará una definición de comportamiento subjetivo analizando los elementos que pertenecen al contexto de recepción de un mensaje. Por último, se describirá un mecanismo de respuesta para el comportamiento subjetivo afín a las necesidades de los requerimientos de adaptación y se describirá un ejemplo de una clase subjetiva

2.2. REQUERIMIENTOS DE ADAPTACIÓN

Los requerimientos funcionales son los requerimientos fundamentales de una aplicación. El motivo por el cual la aplicación fue construida. Por ejemplo, siguiendo con nuestro caso de estudio, el objetivo funcional de la aplicación descripta es el envío y la recepción de mensajes. Estos requerimientos son estables, un cambio en un requerimiento funcional significa un replanteo del propósito de la aplicación. Pero muchas veces existen otros requerimientos adosado a los requerimientos funcionales. Este tipo de requerimiento describe diferentes maneras de realizar las operaciones establecidas por lo requerimientos funcionales dependiendo de las circunstancias de la aplicación. A estos requerimientos los llamaremos *requerimientos de adaptación*. Por ejemplo, si tomamos el requerimiento funcional "mandar mensajes a través de una red de comunicaciones", puede tener adosados los siguientes requerimientos de adaptación: "los mensajes deben ser encriptados, si la comunicación es inalámbrica", ó "deben ser compactados, si la red posee poco ancho de banda".

Notemos que un cambio en un requerimiento de adaptación no debería implicar un cambio fundamental de la aplicación. Este tipo de requerimiento surge por cuestiones coyunturales, como el ancho de banda de la red y el tipo de red, en el caso del ejemplo. Por ese motivo, son más inestables que los requerimientos funcionales.

Otra característica de los requerimientos de adaptación es que complican la lógica de respuesta de los mensajes, introduciendo nuevas condiciones y diversos comportamientos a realizar frente a dichas condiciones. Los mensajes con lógica compleja son difíciles de mantener, por lo tanto el modelado de estos requerimientos implican esfuerzos de diseño para mantener la simpleza y claridad del código.

El hecho de que los requerimientos de adaptación complican la lógica de la aplicación y que dichos requerimientos a su vez son altamente modificables, hace peligrar el buen mantenimiento de la aplicación. El propósito del comportamiento subjetivo y de su mecanismo de respuesta es brindar soporte a la construcción de requerimientos de adaptación obteniendo un diseño simple y de fácil construcción, que lleve a su vez a un mantenimiento sencillo.

En la siguiente sección analizaremos el contexto de recepción del mensaje y daremos una definición de comportamiento subjetivo. Luego explicaremos el mecanismo de respuesta propuesto por esta tesis y daremos un ejemplo de cómo modelar requerimientos de adaptación utilizando comportamiento subjetivo.

2.3. DEFINICION DE COMPORTAMIENTO SUBJETIVO

Un objeto tradicional reacciona de manera objetiva cada vez que recibe un mensaje, en el sentido de que tiene para cada mensaje dentro de una interfase, un único método que implementa su respuesta. Por el contrario, un mensaje con *comportamiento subjetivo* puede tener muchas respuestas, donde cada una está representada por un método diferente y disponible en el momento de recepción. Además, cualquier característica del contexto de recepción puede influir la decisión del método a ejecutar. De ahora en más, llamaremos *mensaje subjetivo* a los mensajes cuyo comportamiento sea subjetivo, *método subjetivo* a una implementación del mensaje subjetivo y *sujetos* o los objetos que tengan definido por lo menos un mensaje subjetivo. Este último término fue también adoptado por Kristensen [KRI/01] con propósitos similares. [PRI/97]

2.4. CONTEXTO DE RECEPCIÓN DE UN MENSAJE

La decisión de la respuesta de un mensaje subjetivo puede ser influenciada por cualquier objeto del ambiente. Esto quiere decir que las condiciones de los mensajes subjetivos pueden implementarse utilizando cualquier objeto de la aplicación. Esta nueva característica de los mensajes pretende amplificar la sensibilidad de los objetos tradicionales, de manera de hacerlos totalmente adaptables a cualquier cambio de la aplicación sin que esto requiera esfuerzos de diseño. Dado que la decisión de la respuesta ocurre en el momento de recepción del mensaje, vamos a analizar cómo se conforma el contexto de recepción de un mensaje. Para ello, se desplegarán todos los objetos que lo conforman, dejándolos disponibles para la construcción de condiciones.

Cuando se produce la recepción de un mensaje subjetivo, los objetos de la aplicación pueden ocupar los siguientes roles: *emisor*, *receptor*, *colaborador* (i.e. parámetros del mensajes y variables de instancia). Además, puede existir otro objeto dentro de la aplicación, que por alguna razón deba influenciar la respuesta, y que no esté incluido en los roles anteriores. A este nuevo rol lo llamaremos *acquaintance* [1]. Un *acquaintance* es un objeto que no interacciona en la invocación y que no es conocido por el receptor, pero, sin embargo, este último debe cambiar su comportamiento dependiendo de sus propiedades. Por ejemplo, continuado con el caso de estudio de los capítulos anteriores, el *braderContext* no es colaborador del objeto que realiza el envío, pero puede convertirse en un *acquaintance* para influenciar el comportamiento de envío. Ver figura 2.1. [GRA/03, DAR/04]

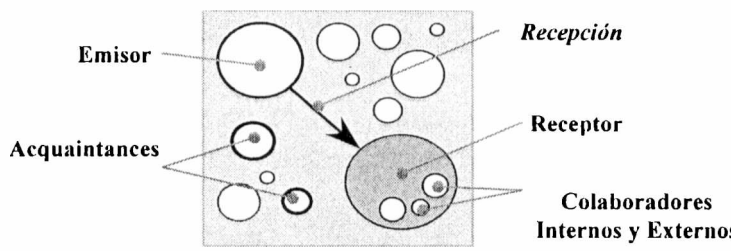

Figura 2.1: Roles en el momento de recepción de un mensaje.

Los objetos de la aplicación que adquieran uno de los roles antes mencionados conforman el contexto de ejecución de un mensaje subjetivo y pueden implementar las condiciones que este posea, influenciando la decisión del comportamiento que el mensaje deba realizar. Estos roles determinan todos los tipos de influencia que puede tener un mensaje y no excluye a ningún objeto del ambiente que pueda influenciar su respuesta.

Un objeto que ocupa el rol de *colaborador*, puede ser referenciado tanto en las implementaciones del mensaje como en sus condiciones. En tanto que los *acquaintances* y el *emisor* sólo pueden utilizarse en la implementación de condiciones y nunca se podrán referenciar desde los métodos del mensaje.

Dado que el patrimonio de un objeto es su identidad, su estado interno y su tipo[2]. Las condiciones de los mensajes subjetivos pueden armarse realizando cualquiera los siguientes tipos de averiguaciones sobre los objetos influyentes:

> *identidad* (i.e. si el objeto influyente tiene cierta identidad),
> *tipo* (i.e. si el objeto influyente define cierto conjunto de mensajes)
> *y propiedad* (i.e. si el objeto influyente responde de manera determinada a cierto mensaje) [CAL/99, GRA/03, DAR/04].

[1] No encontré una palabra castellana que denote el mismo significado.

[2] Un *tipo* es utilizado para denotar una interface particular. [GAM/95]

2.5. MECANISMO DE RESPUESTA DEL COMPORTAMIENTO SUBJETIVO

En los lenguajes Orientados a Objetos basados en clases con despacho dinámico simple, cuando un mensaje se invoca, su método asociado se busca en la clase del receptor. Si este no se encuentra, se busca en la superclase del receptor y así sucesivamente. En algunos lenguajes orientados a objetos como *CLOS* y *CommonLisp*, también es posible asignar varias respuestas a un mensaje dependiendo del tipo de los operadores, este mecanismo se llama sobrecarga de operadores. También dijimos que otro mecanismo de adaptación muy usado es el uso de sentencias if-case. Estas sentencias están embebidas en la implementación del mensaje y elijen que pedazo de código ejecutar dependiendo de ciertas condiciones.

El mecanismo de adaptación de un mensaje subjetivo además de conservar los mecanismos descriptos, si es que el lenguaje subyacente los provee, adiciona otro mecanismo de adaptación. Este mecanismo es interno al mensaje, y se construye ligando las condiciones del mensaje a sus implementaciones. Es igual de flexible que las sentencias if-case, con el agregado de que las condiciones del mensaje son independientes de las implementaciones y a su vez cada implementación es independiente una de otra. Además, es definido por el diseñador, de la misma forma que se definen las sentencias if-case.

En los trabajos anteriores de comportamiento subjetivo, dicho mecanismo se construía formando un árbol de condiciones [DAR/02]. Dicho árbol tenía en cada nodo una condición y un comportamiento. Este árbol de condiciones se evaluaba por niveles. Si alguna condición del primer nivel era verdadera, se ejecutaba el comportamiento que tenía asociado. El comportamiento de ciertas condiciones era una implementación del mensaje y el comportamiento de otras condiciones podía determinar la evaluación del segundo nivel nodos. Este mecanismo de respuesta permitía definir lógica condicional compleja, pero promovía el uso de condiciones anidadas que dificultan la lectura del mensaje. Por ese motivo, esta tesis propone otro mecanismo que permite definir también lógica compleja, pero conservando la sencillez y la claridad en la definición del mensaje.

Como el mecanismo anterior, las diferentes respuestas del mensaje serán independientes, pudiendo adicionar nuevas, borrarlas e intercambiarlas, sin perjudicar el resto de la definición del mensaje. Además de estas características, el mecanismo de respuesta propuesto permite sumar comportamientos, es decir, permitir que varias respuestas de un mensaje se ejecuten de manera secuencial en una misma invocación. Esta nueva capacidad, aumenta la granularidad de las respuestas del mensaje, haciéndolas más sencillas y disminuye la repetición de código. A continuación explicaremos dicho mecanismo de respuesta, denominado *secuencia de métodos agregables.*

2.6. SECUENCIA DE MÉTODOS AGREGABLES

Los mensajes subjetivos se definen a través de un conjunto de métodos que pueden sumarse o agregarse entre sí para formar la respuesta final del mensaje. Cada método, además de tener una secuencia de expresiones, tiene asociada una condición. Si en el momento de recepción del mensaje la condición del método se hace verdadera, el método puede *aplicar* su comportamiento y agregarlo a la respuesta final. Hay dos tipos de métodos: los métodos agregables y los no agregables. Ambos métodos evalúan su condición en el momento de invocación. Primero se evalúan los métodos agregables y todos los que apliquen ejecutarán su comportamiento de manera secuencial según se encuentren definidos. En segundo término, si ningún método agregable puedo aplicar, se evalúan los métodos no agregables. Cuando un método no agregable es ejecutado, corta la evaluación, evitando que otro método no agregable aplique su comportamiento. Por lo tanto, si un método no agregable logra evaluarse y su condición es verdadera, su comportamiento se ejecuta de manera exclusiva con respecto a todos los métodos del mensaje. A estos métodos los llamaremos *otherwise*. Finalmente, si ningún método *otherwise* resulta verdadero, el mensaje tiene comportamiento nulo y no ejecuta ninguna expresión. A continuación revisaremos diferentes construcciones condicionales y veremos cómo pueden definirse utilizando secuencias de métodos agregables. Para definirlas utilizaremos la sintáctica de *Smalltalk* [GOL/83].

Estructura If-then-else

La primera estructura que vamos a analizar es if-then-else. Utilizando la sintáctica de Smalltalk tiene la forma:

```
condition1
        ifTrue:[S1]
        ifFalse:[S2]
```

Dónde S1 y S2 es una secuencia de expresiones.

Primero debe definirse un método agregable con el comportamiento S1, asociado a la condicion1 y en segundo término un método no agregable con el comportamiento S2. La estructura queda definida de la siguiente manera:

```
condition: condition1 method: S1.
otherwiseMethod: S2.
```

En la definición del *otherwiseMethod* la condición está omitida y se asume verdadera.
Como muestra el ejemplo, los métodos agregables se especifican con la palabra clave *method y* los no agregables con la palabra *otherwiseMethod*, ambos precedidos de la condición a evaluar, que se indica con la palabra clave *condition*. En la estructura ejemplo, el último método se ejecuta cuando la condición anterior es falsa, por lo tanto es necesario que tenga una condición siempre verdadera para que la aplicación de S1 y S2 sea alternante.

Estructura Case

Una estructura condicional de tipo case, puede definirse con varios métodos *otherwise* consecutivos:

```
condition: condition1 otherwiseMethod: S1
condition: condition2 otherwiseMethod: S2
condition: condition3 otherwiseMethod: S3
otherwiseMethod: S4
```

Notemos que el segundo método se evalúa siempre que el primero no aplique, y el tercero siempre que no aplique ninguno de los dos anteriores. Finalmente, el cuarto aplica con seguridad si ninguno de los métodos anteriores pudo aplicar. Notemos que, si alguno de los métodos de la estructura aplica, por la definición de los método no agregables, su comportamientos es el único ejecutado.

Estructura If-then-else anidada

Analicemos que sucede cuando la lógica del mensaje nos lleva a definir estructuras if-then-else anidadas para decidir el comportamiento a ejecutar. Veamos este ejemplo:

```
condition1
        ifTrue: [S1]
        ifFalse:[condition2
                        ifTrue: [S3]
                        ifFalse:[S4].
                ]
```

Donde S1, S3 y S4 son secuencias de expresiones.

Notemos que S1 se ejecuta de manera única, sólo si condition1 es verdadera. Además S3 o S4 se ejecutan, si *condition1* es falsa y de manera alternante con respecto a *condition2*. Analicemos una solución de esta estructura con comportamiento subjetivo. Esta podría tener la siguiente forma:

```
condition: condition1 method: S1
condition: condition1 not and condition2 method: S3
condition: condition1 not and condition2 not method: S4
```

Incluso cualquiera de las palabras clave *method* puede intercambiarse por *otherwiseMethod*, ya que las condiciones formadas son disjuntas ente si y nunca se podría dar el caso de ejecutar dos comportamientos.

El problema de esta solución es que la secuencia obtenida es poco clara y no parece tener los beneficios de mantenimiento antes expresados. Incluso puede ser más confusa si el anidamiento de otras estructuras if-then-else es necesario. Una buena práctica para resolver los casos de sentencias if-then anidadas del estilo es separar la lógica en dos secuencias:

```
condition: condition1 method: S1
otherwiseMethod: S6
```

donde S6 se forma de la siguiente manera:

condition: condition2 method: S3
otherwiseMethod: S4

Utilizando dos secuencias para representar la lógica condicional volvemos a tener una lógica de respuesta simple y fácil de mantener. Otra manera de facilitar la lectura es que los métodos *otherwise* se ubiquen últimos en la secuencia. Veamos este caso de estructuras if-then-else anidadas:

condition1
ifTrue: [S1]
*ifFalse:[**S2.***
condition2
ifTrue: [S3]
ifFalse:[S4].
S5]

Donde S2, debe ejecutarse antes de S5.

Notemos S2 y S5 se ejecutan si *condition1* es falsa y que además S2, S3, S4 y S5 no son lo suficientemente independientes, ya que S2 debe preceder a S3 o S4 y que, a su vez, S3 o S4 debe preceder a S5. Una solución sin factorizar el código de S2 y S5, puede ser:

condition: condition1 method: S1
condition: condition2 otherwiseMethod:.S2.S3.S5
condition: condition2 not otherwiseMethod:.S2.S4.S5

Otra solución que evita la repetición de código puede ser:

condition: condition1 method: S1
otherwiseMethod: S2.S6.S5

donde S6 es también una secuencia de métodos agregables:

condition: condition2 method:.S3
otherwiseMethod:.S4

El problema de la última solución, es que no esta completamente desagregada. Si más adelante queremos que cambie la condición de aplicación de S2 y no la de S5, es necesario modificar la implementación del método del *otherwiseMethod.* Una solución bien granulada y que evita la repetición de código es la siguiente:

condition: condition1 method: S1
otherwiseMethod: S6

donde S6 se forma de la siguiente manera:

method:.S2
condition: condition2 method:S3
condition: condition2 not method:S4
method:.S5

Notemos que S2 y S5 siempre aplican porque tienen condición omitida, que es siempre verdadera. Además, S3 y S4 son alternantes dependiendo de *condition2*. De esta manera, el mismo orden de evaluación de la secuencia especifica el pre y post procesamiento de S3 y S4. Además, las condiciones de aplicación de estos procesamientos pueden manejarse de manera independiente.

En conclusión, el mecanismo de respuesta de los mensajes subjetivos se representa utilizando un conjunto de métodos agregables por condición. Por ejemplo, una clase subjetiva que tenga definida una estructura condicional como la última explicada, estará definida como lo muestra la figura 2.2. Donde el mensaje *#publicSubjectiveMessage* se define con una secuencia agregable en la cual uno de sus métodos llama al mensaje *#privateSubjectiveMessage*.

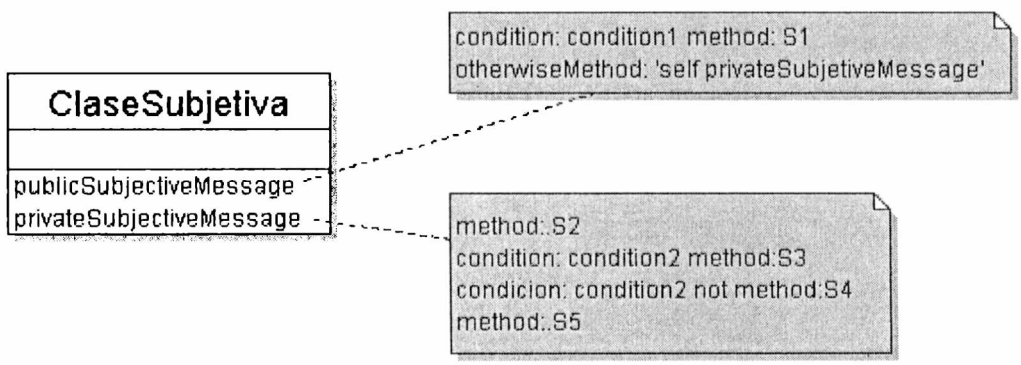

Figura 2.2: Secuencia de métodos agregables privada.

2.7. EJEMPLO DE UNA CLASE SUBJETIVA

A continuación mostraremos un ejemplo de una clase subjetiva. En nuestro caso de estudio, habíamos visto que además del envío y la recepción de mensajes, la aplicación debía cumplir con ciertos requerimientos de adaptación. Estos requerimientos eran, encriptar los mensajes, si el medio de comunicaciones es inalámbrico; compactarlos si la red tiene poco ancho de banda y mandar los mensajes con prioridad urgente, si el usuario esta presente en la habitación.
Suponemos que dentro de nuestra aplicación existe una objeto *network* que se encarga de mantener el estado de la red y de procesar los mensajes antes de enviarlos a un *socket*. Supongamos, además, que el objeto *broaderContext* se suscribió a la máquina subjetiva como *acquaintance* de *network* con la siguiente sentencia:

SubjectiveBehavior addAcquaintance: broaderContext named: #broaderContext toSubject: network

La clase de *network* es una clase subjetiva y se define como los muestra la figura 2.3.

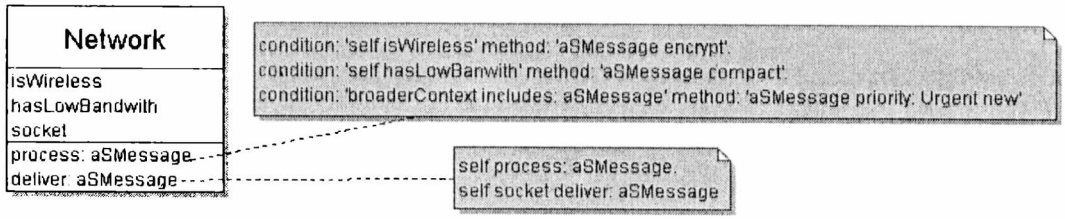

Figura 2.3: Ejemplo de clase subjetiva

El mensaje *#process: aSMessage* es el único mensaje subjetivo. Notemos que la última condición de la secuencia hay una referencia al *broaderContext* y que este no es colaborador de *network*. Notemos además que los comportamientos son agregables por lo tanto pueden aplicarse todos o algunos en la invocación. Si no se aplica ninguno de los tres procesamientos, entonces el mensaje a transmitir por la red no recibirá ninguna modificación y será enviado tal cual fue recibido por el *network*.

En el capítulo 5, analizaremos en profundidad el caso de estudio y definiremos varios comportamientos subjetivos con sus respectivos conjuntos de métodos agregables. Nuestra intención es demostrar que con comportamiento subjetivo se obtiene diseños simples, aunque se representen evaluaciones condicionales complejas.

2.8. CONCLUSIÓN

Los requerimientos funcionales pueden tener adosados varios requerimientos de adaptación. Dichos requerimientos de adaptación hacen que la aplicación varíe su comportamiento dependiendo del contexto. Para incorporar conocimiento al contexto los objetos deben incluir nuevas relaciones de colaboración de manera de implementar las condiciones del mensaje. Estas nuevas relaciones de colaboración se suman a los elementos de diseño de los requerimientos funcionales, sin otorgar valor funcional, ya que no colaboran de la implementación de ningún comportamiento de respuesta. En cambio, el comportamiento subjetivo nos permite establecer dos tipos de relaciones entre objetos: influyente y colaborador. Aquellos objetos que implementen condiciones serán *influyentes* del receptor, dejando como *colaboradores* a los que realmente realicen alguna acción que forme parte de la respuesta al mensaje.

Por otra parte, los requerimientos de adaptación hacen que un mismo mensaje tenga diferentes respuestas dependiendo del contexto de recepción. Como los requerimientos de adaptación son inestables, es necesario que cada definición de comportamiento esté encapsulada, de manera de que sea fácil de intercambiar y modificar. Esta característica de los requerimientos de adaptación también conduce a esfuerzos de diseño, provocando nuevos elementos de modelado que también se entremezclan con los elementos de diseño originados por los requerimientos funcionales. Como consecuencia el diseño resultante no refleja con claridad los requerimientos funcionales de la aplicación, complicando el entendimiento de la aplicación y por lo tanto su mantenimiento.

Capítulo 3

MÁQUINA SUBJETIVA

3.1. INTRODUCCIÓN

En 2003 Andrea Grassano realizó en su tesis de grado [GRA/03] un prototipo para definir comportamiento subjetivo en los mensajes. Dado que en esta tesis se desarrolla un nuevo mecanismo de respuesta, se construyó un nuevo prototipo para verificar su funcionamiento. En este capítulo explicaremos algunos puntos de la implementación y además explicaremos ciertas características reflexivas para modificar el comportamiento subjetivo dinámicamente. Al final del capítulo, discutiremos sobre la eficiencia del comportamiento subjetivo y mencionaremos algunas capacidades dinámicas obtenidas.

3.2. IMPLEMENTACIÓN

Para implementar el conjunto de métodos agregables introducido por esta tesis, se hizo un nuevo prototipo de un ambiente orientado a objetos que permite la definición de mensajes subjetivos. Este prototipo, como el anterior, es realizó sobre la base del lenguaje Smalltalk-80. Este lenguaje es Orientado a Objetos y basado en clases. Cada instancia tiene una única clase cuya definición contiene los métodos. Cuando una instancia recibe un mensaje, el mecanismo de despacho busca el método correspondiente en el diccionario de métodos de su clase. Si no lo encuentra en el diccionario de dicha clase, busca en el diccionario de la superclase y así sucesivamente hasta encontrarlo. Si ninguna de estas clases

implementa un método para el mensaje, la máquina virtual invoca el mensaje *#doesNotUnderstand:*.

En el prototipo implementado para esta tesis, una vez que el diseñador define una secuencia de métodos agregables para un mensaje, estos elementos se guardan en instancias de *SubjectiveMessage*. Dichos objetos se almacenan en un diccionario de clase particular. A la vez que se guarda el mensaje subjetivo, se almacena en el diccionario regular un método que detona la evaluación del mensaje subjetivo. Este método inicializa el contexto de recepción subjetivo del mensaje, busca el mensaje subjetivo en el diccionario de mensajes subjetivos y solicita que se ejecuten las respuestas válidas. En la figura 3.1 vemos un diagrama de las clases principales del prototipo. Notemos que se subclasificó *Class* con la intención de ubicar el diccionario de mensajes subjetivos.

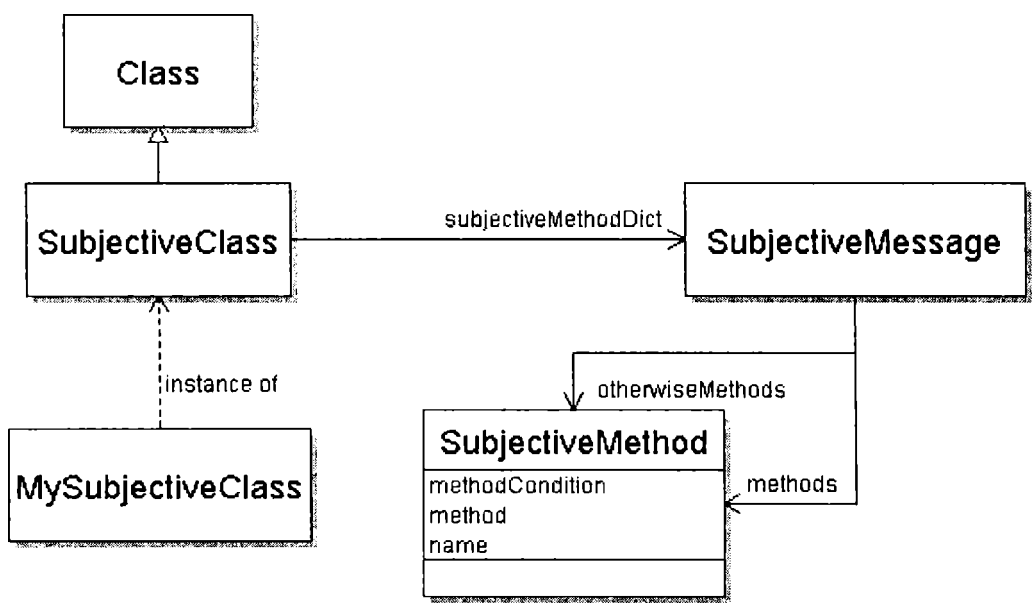

Figura 3.1: Estructura de clases de la máquina subjetiva.

La evaluación de la secuencia de métodos agregables tiene el funcionamiento que se describe en el capítulo anterior. Para realizar dicha evaluación la máquina subjetiva debe armar el contexto de recepción obteniendo las referencias necesarias para resolver las condiciones. Los colaboradores están disponibles naturalmente a través de las variables de instancia y los parámetros del mensaje; la referencia al emisor es obtenida a través del objeto que representa el contexto de ejecución del mensaje; y por último, el acceso a los objetos *acquaintance* es administrado por la maquina subjetiva, que mantiene las referencias de los *acquaintance* vinculados a cada sujeto (i.e. objetos con mensajes subjetivos). Cuando un sujeto tiene que implementar una condición a través de un *acquaintance* es necesario que el objeto *acquaintance* se subscriba. Esta subscripción puede hacerse en el momento de inicialización del *acquaintance* o del sujeto. En el prototipo de esta tesis, se debe invocar el siguiente mensaje: *SubjectiveBehavior addAcquaintance: anAcquaintance named: acquaintanceName toSubject: aSubject.* De esta manera, la maquina subjetiva tiene conocimiento de que el objeto *anAcquaintance* será referenciado por el

objeto *aSubject* utilizando la palabra *acquaintanceName*. Esta referenciación sólo puede darse dentro de las condiciones.

Otra diferencia con respecto al primer prototipo es que se crean dos contextos de ejecución en el momento de invocación de un mensaje subjetivo. Uno para evaluar las condiciones de los métodos y otro, con la misma información que el contexto normal, para evaluar el comportamiento de los métodos subjetivos que apliquen. De esta manera, se diferencian el alcance para construir condiciones y el alcance para la implementación de los métodos. Por lo tanto, el *sender* y los *acquaintances* sólo podrán accederse en la definición de condiciones, y no en la definición de métodos.

3.3. REFLEXIÓN EN UN LENGUAJE ORIENTADO A OBJETOS

Un programa reflexivo es aquel que razona sobre si mismo. Una arquitectura completamente reflexiva es aquella en la que un proceso puede acceder y manipular la representación de su estado completa, explicita y causalmente [FOO/89]. "Causalmente" significa que cualquier cambio hecho a una representación de un proceso se refleja inmediatamente en su estado actual y comportamiento.

La implementación del prototipo puede darse en cualquier ambiente Orientado a Objetos, siempre y cuando cumpla con las siguientes características reflexivas:

- Debe ser posible reestructurar lo métodos de las clases para agregar la lógica de los mensajes subjetivos y
- también debe ser posible acceder al contexto de ejecución de un método. Éste debe referenciar, al menos, el emisor, el receptor, los colaboradores externos y los colaboradores internos. Los colaboradores además deben poder accederse sin especificarlos explícitamente por nombre.

3.4. REFLEXIÓN DEL COMPORTAMIENTO SUBJETIVO

Hasta este momento, el diseñador era el encargado de definir la secuencia de métodos agregables de un mensaje subjetivo. Sin embargo dichos métodos tienen una clara representación en la estructura de la máquina subjetiva, y por lo tanto también pueden generarse dinámicamente a través de solicitudes. Por ejemplo, para agrega un nuevo método subjetivo, alcanza con indicar como se forma el nuevo método y donde se lo debe ubicar. La información necesaria es: el nombre de la clase subjetiva, el nombre del mensaje subjetivo, la expresión de la condición a evaluar y el método subjetivo (i.e. el método que se debe ejecutar cuando la condición es verdadera). Además debe indicarse si se trata de un método agregable o no. Continuando con el caso de estudio, una expresión que agregue un método subjetivo en un mensaje podría ser:

```
Network addSubjetiveMethodTo: #process:aSMessage
```

```
condition: 'self hasLowBandwith'
method: 'aSMessage compact'.
```

Esta expresión crea un método agregable para el mensaje *#process: aSMessage* de la clase *Network*. Este método evalúa la condición '*self hasLowBandwith*' y ejecuta la expresión *'aSMessage compact'*. Si el método a agregar es *otherwise*, la expresión en cambio sería:

```
Network addSubjetiveOtherwiseMethodTo:#process aSMessage
  condition: 'self hasLowBandwith'
  method: 'aSMessage compact'
```

Si queremos agregar más de un método a un mismo mensaje, es necesario establecer el orden en el que deben ejecutarse. Por lo tanto necesitaremos una nueva palabra clave que lo especifique. Por ejemplo:

```
Network addSubjetiveOtherwiseMethodTo:#process aSMessage
  condition: 'self hasLowBandwith'
  method: 'aSMessage compact'
  order: 1

Network addSubjetiveOtherwiseMethodTo:#process aSMessage
  condition: 'self isWireless'
  method: 'aSMessage encrypt'
  order: 2
```

Otra operación reflexiva puede ser borrar un método de un mensaje subjetivo. Para ello, se requiere identificar el método dentro de la secuencia del mensaje. Es posible utilizar el mismo nombre de secuencia, pero es más claro si nombramos los métodos. Sumando la palabra clave *named*: seguida del símbolo que represente el nombre, la expresión de creación quedaría de la siguiente manera:

```
Network addSubjetiveMethod: #CompactWhenLowBandwith
  message: #process:aSMessage
  condition: 'self hasLowBandwith'
  method: 'aSMessage compact'
  order: 1
```

En este caso, la expresión de borrado del método sería la siguiente:

```
Network removeSubjetiveMethod: #CompactWhenLowBandwith
    fromMessage: #process aSMessage
```

También podemos querer modificar un mensaje subjetivo para que tenga siempre comportamiento nulo. Por lo tanto, tendríamos que borrar todo sus métodos asociados. La expresión sería la siguiente:

```
Network removeAllSubjectiveMethodsFromMessage: #process:aSMessage
```

Con las expresiones reflexivas vistas alcanza para modificar el mecanismo de respuesta de un mensaje subjetivo dinámicamente. Al final de este capítulo vamos a ver un ejemplo de cómo se puede aprovechar esta capacidad. En el capítulo 6, utilizaremos expresiones reflexivas para configurar una aplicación con preferencias del usuario.

3.5. CAPACIDADES DINÁMICAS

En esta sección vamos a analizar ciertas capacidades dinámicas que requieren poco esfuerzo de diseño y de implementación utilizando comportamiento subjetivo [DAR/04].

Adaptación dinámica de interfaces

Es muy común tener problemas de incompatibilidad de interfaces cuando se usan clases existentes para armar aplicaciones. Supongamos que una clase de objetos A interactúa con objetos b, enviándoles el mensaje *#message1* y *#message2*. Aquí estamos suponiendo que el mensaje b posee cierta clase que tiene definido el siguiente tipo: (*#message1*, *#message2*). Supongamos, que por algún requerimiento de adaptabilidad, pueden existir otros objetos b de otras clases, con los cuales A debe interactuar. Además supongamos que dichas clases de b, en vez de tener definido el tipo (*#message1*, *#message2*), tienen definido el tipo (*#messageOne*, *#messageTwo*), donde el uso de *#message1* y *#message2* se corresponde con el uso de *#messageOne* y *#messageTwo*, respectivamente.

Dado que el comportamiento subjetivo provee el análisis de tipo de una instancia, tenemos suficiente capacidades dinámicas para resolver el problema de adaptación enunciado. Por ejemplo, suponiendo que la clase *A* interacciona con objetos *b*, mandando primero *#message1* y después *#message2*, podemos definir un mensaje subjetivo en la clase A, sabiendo de antemano los dos tipos conocidos para b y como se corresponden entre ellos. En la figura 3.2 vemos como queda definido el mensaje subjetivo.

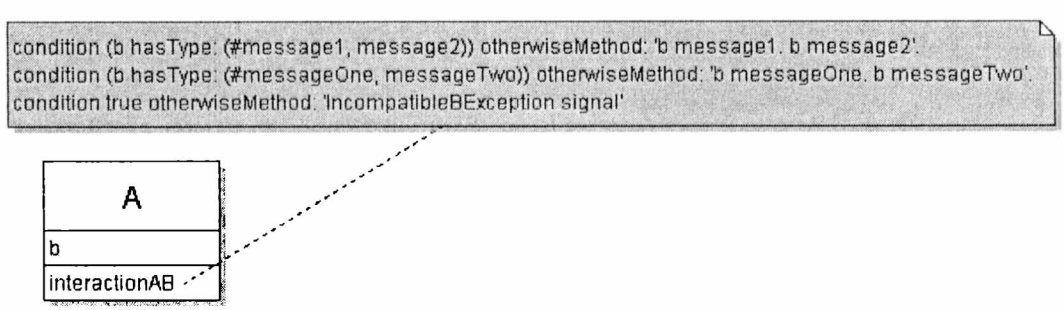

Figura 3.2: El mensaje *#interactionAB* adapta dinámicamente el comportamiento de la clase A, según el tipo del colaborador b.

Esta motivación es similar a la del patrón *Adapter* [GAM/95], salvo que la solución con el mensaje *#hasType*: es dinámica y no configurable como la que propone el patrón. En el libro de patrones el adaptador no pretende modificar su comportamiento automáticamente dependiendo del tipo de la entidad que es adaptada, si no que es configurado externamente. En la solución del *Adapter* la correspondencia entre mensajes se establece definiendo una clase abstracta con el subtipo de *b* que es utilizado por la clase A. Esta clase tiene el rol *Target* y tendrá definido el tipo (*#message1*, *#message2*). Con cada objeto b nuevo, que no defina dicho tipo, se deberá subclasificar el *Target* para que la nueva clase delegue cada

mensaje al correspondiente mensaje del b nuevo. En la figura 3.3 se muestra el diagrama de clases.

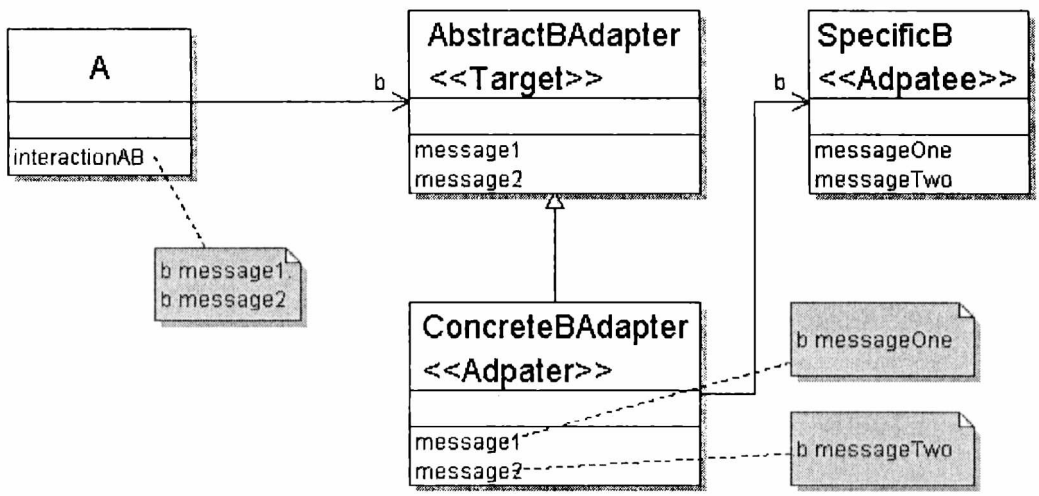

Figura 3.3: Estructura sin comportamiento subjetivo para adaptar el comportamiento de la clase A, según el tipo del colaborador b.

La solución del patrón adapter es menos flexible pero parece más eficiente ya que no requiere el checkeo dinámico del tipo del colaborador b.

Visibilidad Dinámica de Mensajes

En los lenguajes orientados a objetos conocidos no es posible modificar la visibilidad de los mensajes a nivel de instancia. Algunos lenguajes, permiten definir propiedades de visibilidad pero sólo a nivel de clase. Con comportamiento subjetivo, podemos configurar la visibilidad de los mensajes de manera dinámica. Podemos simular que una instancia no sabe responder a un mensaje particular en ciertas condiciones. Siguiendo con el ejemplo, supongamos que el usuario registrado al dispositivo quiere acceder a la dirección de correo de un usuario presente en la habitación y que el mensaje para acceder a la dirección de los usuarios es *#address*. Supongamos también que la dirección de un usuario no debe estar disponible a cualquier solicitante, por lo tanto, vamos a definir en *User* un mecanismo de protección de visibilidad. Este mecanismo puede consistir en tener una lista de usuarios confiables y sólo retornar la dirección, si el emisor del mensaje *#address* es uno de ellos. Por ejemplo, un usuario confiable puede ser aquel que trabaja en el mismo edificio que el solicitado. Si el usuario solicitante no pertenece al mismo edificio que el receptor, la respuesta al mensaje *#address* será elevar una excepción *DNU* (i.e. Does Not Understand Message). En la figura 3.4 vemos la clase *User* con el mecanismo de protección de visibilidad descrito.

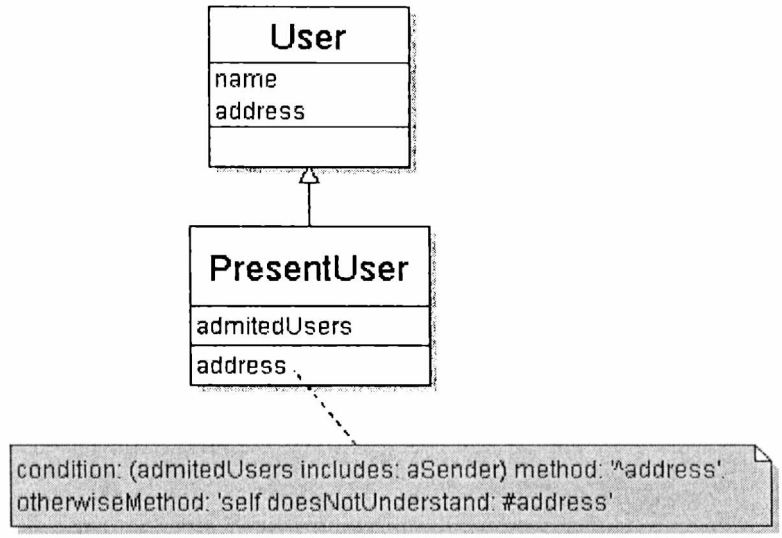

Figura 3.4: Si el emisor está en la lista de usuarios admitidos, el mensaje subjetivo retorna la dirección de correo.

Aprendizaje Dinámico

El diseñador del mensaje subjetivo define las condiciones para decidir la respuesta del mensaje. Este mecanismo de decisión tiene una representación en el sistema y se almacena en una clase subjetiva. Como vimos anteriormente, existen técnicas reflexivas subjetivas que permiten la modificación en ejecución de dicho mecanismo. Por lo tanto, una clase de objetos podría aprender y cambiar su mecanismo de decisión dinámicamente. Por ejemplo, siguiendo con el ejemplo utilizado en la capacidad anterior, supongamos que los usuarios confiables se almacenan en una colección administrada por un objeto *Singleton* [GAM/95] de la aplicación y que esta lista puede crecer y decrecer según solicitudes. Además, supongamos que la aplicación debe operar con cierta eficiencia y que por lo tanto el mensaje *#address* no debería demorarse más de la cuenta realizando la búsqueda entre los usuarios confiables. Para prevenir que el mecanismo de protección disminuya la eficiencia de la aplicación, podría suspenderse automáticamente, cuando la cantidad de usuarios confiables alcance cierto número. En este caso, la manera de modificar el comportamiento condicional del mensaje subjetivo *#address* sería invocar la siguiente sentencia:

```
User removeAllSubjectiveMethodsFromMessage: #address

User addSubjectiveMethod: #freeAccesssToAddress
  message: #address
  method:'^address '
```

Luego las instancias *user* retornaran la dirección a todos los usuarios que la soliciten. Esta política podrá continuar hasta que la colección se hayan reducido o hasta que otra necesidad de la aplicación precise nuevamente del mecanismo de protección. En tal caso, la manera de reincorporar dicho mecanismo sería con la siguiente secuencia:

```
User addSubjectiveMethod: #visibleToAdmitedUsers
```

```
message: #address
condition: 'self admitedUser includes: aSender'
method:'^address '
order: 1


User addSubjectiveOtherwiseMethod:#doNotGiveAdrees
 message: #address
 method: 'self doesNotUnderstand: #address'
 order:2
```

3.6. EFICIENCIA DE EJECUCIÓN DEL COMPORTAMIENTO SUBJETIVO

Las capacidades dinámicas obtenidas a través del comportamiento subjetivo, tienen como desventaja la ineficiencia inherente del comportamiento subjetivo. La ejecución de un mensaje subjetivo siempre va ser más lenta que la ejecución de un mensaje con despacho dinámico simple. El prototipo construido no tiene como fin bajar tiempos de ejecución, pero se sabe de antemano que existe una ineficiencia inevitable debido al cálculo del contexto de ejecución de las condiciones, al cálculo del contexto de las implementaciones y a la elección de los métodos a aplicar. De todas maneras, es posible justificar esta ineficiencia, ya que, presumiblemente, una implementación con despacho dinámico simple, necesite de más clases y relaciones para equiparar una implementación con comportamiento subjetivo. A más clases y relaciones, tendremos más invocaciones entre objetos de la aplicación y por lo tanto, más ineficiencia. En este sentido, el tiempo de ejecución de los mensajes subjetivos puede ser compensado.

En los capítulos siguientes, compararemos dos diseños de la misma aplicación, uno modelado con un lenguaje orientado a objetos tradicional y otro con comportamiento subjetivo. La intensión de esta comparación de diseños es demostrar, mediante un ejemplo de aplicación particular, que el comportamiento subjetivo puede ayudar a la definición de diseños más compactos.

3.7. CONCLUSIÓN

Como habíamos mencionado, la principal desventaja de las sentencias *if-case* es que las condiciones están embebidas en la implementación del mensaje, junto con la implementación de los diferentes comportamientos. Esto, además de no promover la independencia de las respuestas de los mensajes, hace que las condiciones estén definidas a nivel de implementación, dentro del método del mensaje. Como consecuencia, la lógica de decisión de respuesta no tiene una representación separada y resulta difícil modificarla dinámicamente.

Con comportamiento subjetivo esto no sucede ya que el mecanismo de decisión tiene una representación en el sistema y se almacena en un diccionario dentro de la clase subjetiva. Por lo tanto, es posible utilizar técnicas reflexivas subjetivas que permiten la modificación en ejecución de dicho mecanismo.

Además de esta capacidad reflexiva, el comportamiento subjetivo permite definir con simpleza otras capacidades dinámicas, como por ejemplo, la visibilidad de un mensaje dependiendo de características del emisor, o la adaptación del comportamiento dependiendo del tipo de los colaboradores. Todas estas capacidades permiten que la aplicación modifique con facilidad su comportamiento, en función del contexto en el cual se ejecuta.

Estas capacidades dinámicas tienen como contrapartida la ineficiencia de ejecución inherente al comportamiento subjetivo. Uno de los puntos de esta tesis es justificar dicha eficiencia, demostrando que los diseños con comportamiento subjetivo son más compactos. Con ese fin, vamos a analizar en los siguientes capítulos, la aplicación del comportamiento subjetivo en el diseño. Para ello, utilizaremos definiciones de aplicaciones *context-aware*, dado que están definidas en su mayor medida por requerimientos de adaptación.

Capítulo 4

UNA APLICACIÓN CONTEXT-AWARE

4.1. INTRODUCCIÓN

Como se dijo anteriormente, los requerimientos de adaptación describen diferentes formas de realizar las operaciones establecidas por los requerimientos funcionales de la aplicación. Además estas variaciones dependían de las distintas circunstancias de ejecución de la aplicación. A partir de esta definición, hemos encontrado en trabajos sobre *context-awareness*, muchos requerimientos de este tipo. Por lo tanto, se consideró que este dominio de aplicaciones era ideal para la aplicación del comportamiento subjetivo.

En este capítulo elaboraremos una especificación de una aplicación, cuyos requerimientos de adaptación fueron tomados fundamentalmente de la tesis realizada por Anind Dey [DEY/00] y también inspirada en varios trabajos sobre context awareness [CHE/99, HEN/04, HEN/02, WAN/04, FIN/02, FIT/02, BAN/02]. Esta especificación profundiza el caso de estudio utilizado en los capítulos anteriores.

4.2. ESPECIFICACIÓN DE UNA APLICACIÓN CONTEXT-AWARE

Para que la especificación sea ordenada, primero describiremos los requerimientos funcionales y luego adosaremos a dichos requerimientos un conjunto de requerimientos de adaptación.

Requerimientos Funcionales

Tomemos como caso de estudio una aplicación que corre en un dispositivo móvil. Dicha aplicación tiene como requerimiento funcional principal mandar y recibir mensajes a través de una red de comunicaciones. El usuario registrado construye el mensaje de texto, le asigna un destinatario que saca de sus contactos, una prioridad y solicita su envío. Si la conexión esta habilitada, el mensaje es enviado inmediatamente. En cambio, si el dispositivo móvil perdió conectividad, la aplicación debe almacenar los pedidos de envío hasta que la conexión este restablecida. Cuando la conexión esté habilitada nuevamente, los mensajes encolados son enviados según las prioridades asignadas por el usuario. Por otra parte, cada cierto tiempo la aplicación obtiene de la red los mensajes recibidos. Cuando recupera los mensajes notifica al usuario del dispositivo con un aviso en la pantalla y los almacena.

Requerimientos de Adaptación

Además del requerimiento funcional de mandar y recibir mensajes, es necesario que la aplicación sea capaz de adaptarse a cambios de contexto. En particular, debe cumplir con los siguientes requerimientos de adaptación.

i. El encriptado es más importante para las redes de comunicación inalámbrica, que para las redes en donde se tiene control del transporte físico. La consecuencia es que mientras nos movemos por diferentes medios de comunicación, se precisan diferentes niveles de seguridad. Proveer siempre el mayor nivel no es siempre la mejor opción, en general, a más seguridad, más es el costo de procesamiento. Dado que nuestra aplicación puede llegar a operar con distintos tipos de redes y que se trata de una aplicación sencilla, vamos a definir dos niveles de seguridad para el envío de mensaje. El nivel de seguridad más alto se aplica cuando la red es inalámbrica, encriptando los mensajes antes de ser enviados. Y el nivel más bajo cuando las redes poseen medio físico, mandando los mensajes sin ningún procesamiento.

ii. El estado de la red de comunicaciones puede variar y es de esperarse que la aplicación se adapte para operar de la forma más eficiente. Si la red tiene poco ancho de banda, es conveniente enviar información de bajos volúmenes para aumentar la velocidad de envío, por lo que nuestra aplicación compactará los mensajes antes de ser enviados.

iii. La configuración de la aplicación no solo tiene en cuenta las características de la red de comunicaciones y las prioridades de envío asignadas por el usuario, sino también el contexto social del usuario del dispositivo. Por ejemplo, el tipo de reunión de la que participa y los usuarios presentes en la misma habitación, pueden alterar las prioridades de envío del mensaje. Es de esperarse que la los mensajes enviados entre personas que se encuentran reunidas tenga más urgencia que los de personas que no están presentes. Por ese motivo, si al momento de enviar

un mensaje nuestra aplicación detecta que el destinatario está presente, cambiará la prioridad del mensaje a *urgente* antes de solicitar su envío.

iv. Como dijimos anteriormente, el usuario registrado es el único capaz de solicitar el envío de mensajes. Para reforzar esta restricción, y prevenir que cualquier aplicación intrusa ejecute envíos ilegítimos, nuestra aplicación analizará el origen de la solicitud antes de encaminar el mensaje por el medio de comunicación.

v. Las aplicaciones del dispositivo móvil tienen varios modos de operar dependiendo de la batería que le reste al dispositivo. Cuando nuestra aplicación detecte poca batería, cambiará su forma de trabajar de manera de aprovechar los recursos restantes de manera eficiente. Se suspenderá la recepción de mensajes, y el envío sólo se permitirá a direcciones de emergencia.

vi. Cada sala de reuniones tendrá un dispositivo que indique el tipo de reunión y si la reunión ha comenzado. Cuando los usuarios se encuentren en una sala de reuniones, el dispositivo de sala notificara a los dispositivos de los usuarios el momento de comienzo de la reunión, el carácter que esta posee (ordinaria o importante) y el momento en que finaliza. Las aplicaciones de los dispositivos receptores deberán utilizar esta información para adaptar su comportamiento. Dado que el arribo de mensajes suele distraer la atención del usuario, si el usuario se encuentra en una reunión importante, la aplicación almacenará los mensajes recibidos, pero de manera silenciosa, sin realizar el aviso en pantalla. Cuando la reunión culmine, se notificará al usuario los mensajes recibidos.

vii. Queremos que nuestra aplicación corra en cualquier tipo de dispositivo. Por lo tanto, debe estar preparada para conectarse a diferentes tipos de redes e interactuar con diferentes protocolos, que probablemente ya estén implementados. Para permitir esta portabilidad, nuestra aplicación debe ser capaz de reconfigurarse de manera automática para interactuar con las aplicaciones de red existentes y conocidas de antemano por el diseñador. Además debe proveer un mecanismo simple de modificación para incluir a las aplicaciones no conocidas.

viii. Los puntos que hemos desarrollado describen el comportamiento por defecto de la aplicación al momento de enviar y recibir mensajes. Sin embargo, cada usuario que se registra debe poder cambiar dicho comportamiento dependiendo de sus preferencias. Cuando un usuario se registra en el dispositivo, el dispositivo recupera la información del usuario enviando una solicitud a la red. Además de los datos personales, dicha información debe incluir sus preferencias de envío y recepción de mensajes. Las preferencias son directivas que el usuario registrado transmite a la aplicación antes de que esta comience a ejecutarse, estas directivas pueden ser:

   a. mandar mensajes con prioridad urgente a los usuarios presentes.

b. no cambiar la prioridad de los mensajes,
c. encriptar siempre los mensajes enviados,
d. encriptar los mensajes, si la red es inalámbrica,
e. no encriptar nunca los menajes,
f. no compactar mensajes,
g. compactar mensajes cuando la conexión es lenta,
h. siempre notificar la recepción de mensajes,
i. y no notificar la recepción de mensajes en reuniones importantes.

4.3. DISEÑO DE LOS REQUERIMIENTOS FUNCIONALES

Como dijimos en el capítulo 2, el comportamiento subjetivo puede ser una herramienta útil para modelar requerimientos de adaptación, por lo tanto, esperamos que también sea útil para el modelado de aplicaciones *context-aware*. Ya concluida la especificación de la aplicación, en los próximos capítulos presentaremos dos diseños alternativos. Uno basado en un paradigma orientado a objetos tradicional y otro utilizando comportamiento subjetivo para modelar los requerimientos de adaptación. Luego realizaremos comparaciones de los modelos obtenidos y analizaremos la utilidad esperada.

En esta sección vamos a construir un modelo básico teniendo en cuenta sólo los requerimientos funcionales. Esta parte del diseño servirá como punto de partida para adosar los requerimientos de adaptación. Luego, extenderemos el modelo dos veces. Una incluyendo los requerimientos de adaptación siguiendo con el paradigma tradicional y otra utilizando comportamiento subjetivo.

Comenzando con el modelo de requerimientos funcionales, la clase principal de nuestro modelo se llamará *Messenger* y como lo indica la especificación de la aplicación, la responsabilidad principal de esta clase será gestionar el envío y la recepción de mensajes. Por otra parte, el *messenger* tendrá un colaborador que se encargará de la transmisión y recepción de mensajes a través del medio. Su clase se denominará *Network* y su responsabilidad principal será mantener la comunicación con un servidor de mensajes. La comunicación con el servidor se llevará a cabo a través de un *socket*. Cuando un usuario solicite el envío de un mensaje, el *messenger* delegará al *network* la realización de dicho envío. Cuando la conexión con el servidor esté disponible, el *network* solicitará la transmisión de datos a través del *socket*. Mientras el objeto *network* espere que la conexión se habilite, los mensajes serán almacenados en una cola, y ordenados por prioridad. Cuando el *network* detecte que la conexión esta restablecida enviará al *socket* los primeros mensajes de la cola. El *messenger* además solicitara al *network* los mensajes recibidos. El *network* revisará al *socket* y retornará los mensajes recibidos para que el *messenger* gestione el almacenamiento. Si el *messenger* recibe algún mensaje notificará al usuario con un aviso en pantalla y guardará los mensajes en la bandeja de entrada.

En el diagrama de la figura 4.1 se muestra una estructura orientada a objetos que hace posible este tipo de colaboración. El lenguaje elegido para el modelado es

UML [BOO/98] y además se utilizará la sintaxis de *Smalltalk* [GOL/83] para detallar el comportamiento de algunos mensajes.

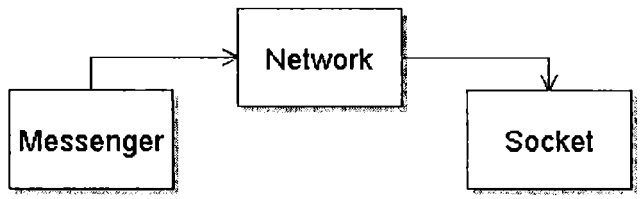

Figura 4.1: Estructura básica para los requerimientos funcionales

Los mensajes más importantes de las clases *Messenger* y *Network* son *#deliver: aSMessage* y *#receive*. Cada clase resuelve un aspecto del envío y de la recepción. El *messenger* dialoga con el usuario, controlando el envío y la recepción; y el *network* concreta la transmisión en colaboración con el *socket* que implementa el protocolo de acuerdo al tipo de la red de comunicaciones. El *network* manda mensajes al *socket* como una cadena de caracteres. Ver figura 4.2, 4.3 y 4.4. El usuario registrado o el *messenger* invocan el mensaje del *messenger* *#receiveMessages* para obtener los mensajes recibidos. El *messenger* obtiene los mensajes del *network* y los almacena en la bandeja de entrada. El *network* recupera los mensajes del *socket* como una colección de *Strings*, los convierte a objetos *SMessage* y los retorna al *messenger*.

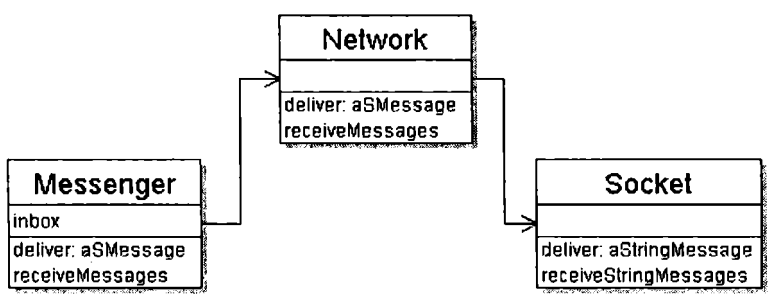

Figura 4.2: Estructura para la transmisión de mensajes.

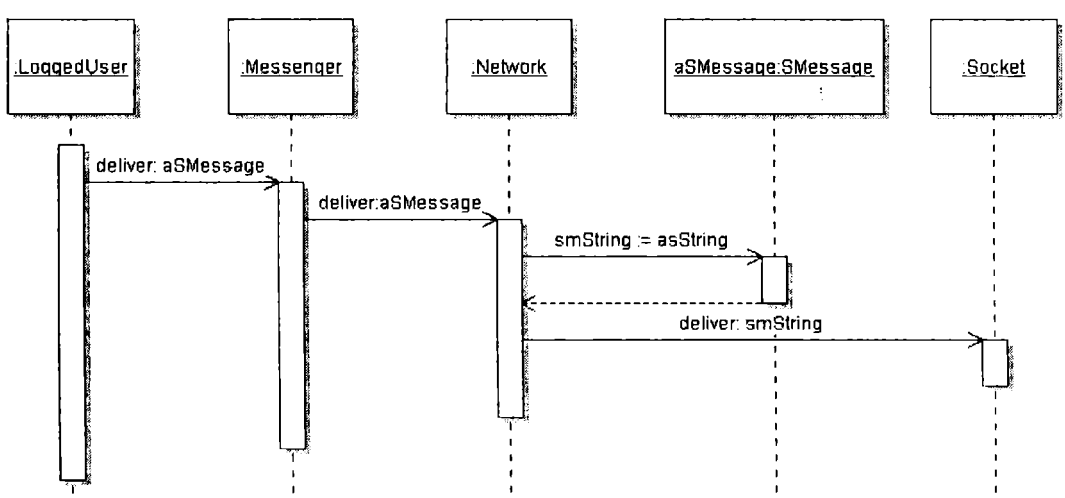

Figura 4.3: Interacción para mandar un mensaje a través de la red.

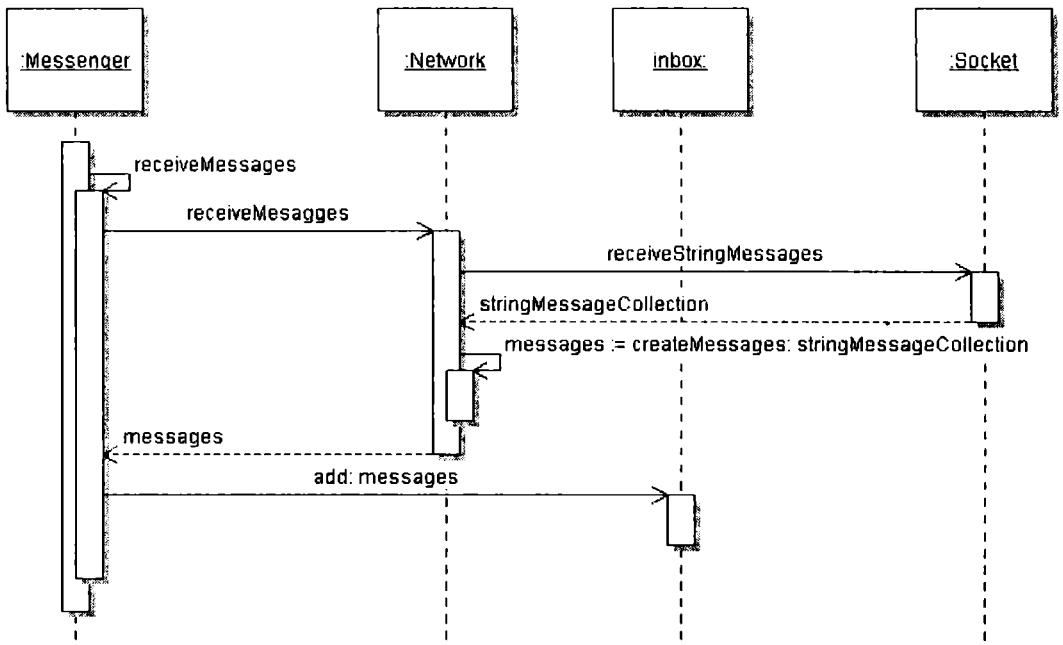

Figura 4.4: Interacción para obtener los mensajes recibidos.

Los mensajes se modelan con objetos de clase *SMessage*. Estos objetos almacenan el texto del mensaje, el destinatario, el emisor y la prioridad de envío. Tanto el destinatario del mensaje como el emisor, están representados por objetos *Contact*. Estos objetos almacenan el nombre de un usuario junto con su dirección de mensajes. Además existe otro tipo de usuario, el registrado al dispositivo. Este objeto se llama *loggedUser* y no sólo almacena el nombre y la dirección del usuario, sino que posee una lista de contactos remotos a los cuales puede enviarles mensajes (Ver figura 4.5).

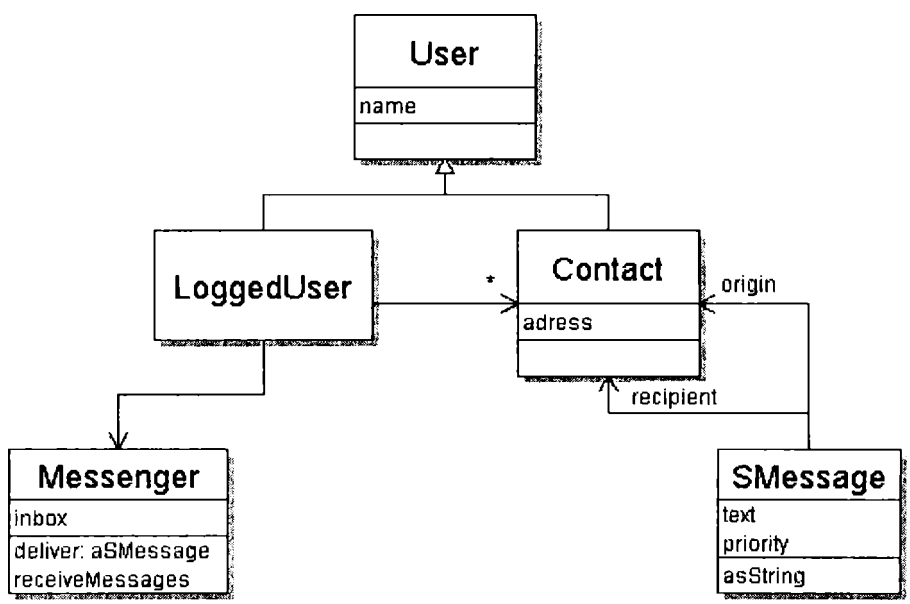

Figura 4.5: Estructura de las clases para modelar el usuario registrado, sus contactos y los mensajes.

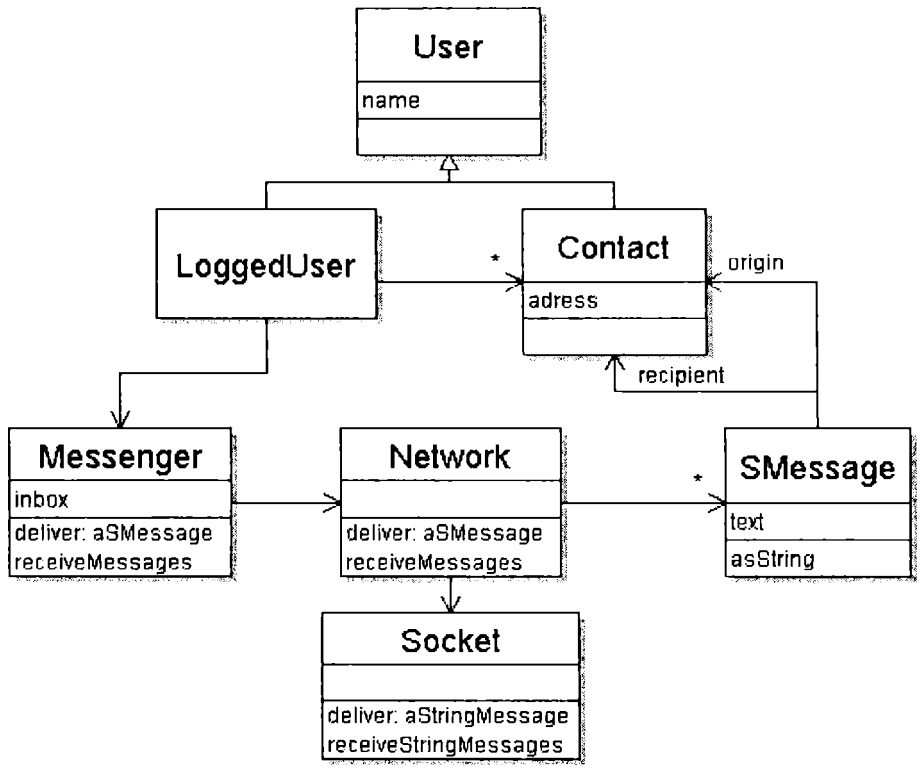

Figura 4.6: Diseño de los requerimientos funcionales.

4.4. CONCLUSIÓN

Hemos especificado una aplicación basada en ejemplos de aplicaciones *context-aware*. Esta aplicación posee pocos requerimientos funcionales con varios requerimientos de adaptación adosados. Hasta este punto describimos el diseño sin incluir los requerimientos de adaptación. Los requerimientos analizados son simples, lo que nos permite llegar sin dificultades a un diseño sencillo. La figura 4.6 muestra el diagrama completo con todos los elementos de diseño que comprende el modelo. En el capítulo siguiente, extenderemos el diseño obtenido incorporando los requerimientos de adaptación.

Capítulo 5

DISEÑO EN UN PARADIGMA TRADICIONAL

5.1. INTRODUCCIÓN

Como mencionamos en capítulos anteriores, la utilidad del comportamiento subjetivo es dar soporte de diseño a los requerimientos de adaptación. Hemos observado que los requerimientos de adaptación aparecen vinculados a los requerimientos funcionales. Los requerimientos de adaptación hacen que una operación indicada por un requerimiento funcional cambie su comportamiento según ciertas condiciones de la aplicación. Generalmente surgen por cuestiones coyunturales a la aplicación, por lo tanto, son más inestables que los requerimientos funcionales [KOR/00]. Para que la aplicación sea modificada acorde a los requerimientos de adaptación debe estar diseñada para el cambio. Por lo tanto, es conveniente que las diferencias de comportamiento introducidas estén encapsuladas y sean independientes unas de otras. De esta manera, pueden suprimirse intercambiarse y modificarse sin que esto traiga grandes consecuencias en el diseño de la aplicación.

5.2. DISEÑO DE REQUERIMIENTOS DE ADAPTACIÓN

En esta sección incorporaremos elementos de diseño para modelar los requerimientos de adaptación. Además del despacho de mensajes simple, en algunas ocasiones utilizaremos sentencias *if-case* como mecanismo de adaptación. También tendremos la precaución de independizar lo más posible las diferentes implementaciones devenidas de los requerimientos de adaptación. Para eso nos valdremos no sólo de mensajes privados, sino de otros recursos más interesantes de la orientación a objetos, como la herencia, la sobreescritura y el polimorfismo. También basaremos nuestras soluciones en patrones de comportamiento propuestos por el libro de patrones de diseño de Gamma [GAM/95].

Encriptado y compactado de mensajes

Los dos primeros requerimientos de adaptación vistos en el capítulo anterior proponen diferentes comportamientos para el envío de mensajes. Si la red de comunicaciones es inalámbrica, los mensajes deben ser encriptados y si la red posee poco ancho de banda, deben ser compactados. Por lo tanto, nuestra aplicación tendrá diferentes estados de comportamiento dependiendo de las condiciones de la red. Supongamos que al momento de abrirse la conexión con el servidor, el *network* detecta el estado y decide que procesamiento realizar sobre los mensaje que serán transmitidos.
Para encapsular los diferentes procesamientos que puede realizar el *network*, podemos aplicar el patrón *Strategy*. Definiremos una jerarquía que tenga como fin implementar las diferentes formas de procesamiento de mensajes antes del envío. Esta forma de procesado es configurada de antemano cada vez que se abre una conexión, por lo tanto, al momento del envío, no tendremos la necesidad de utilizar construcciones if-case.
Dado que un medio de comunicaciones puede ser inalámbrico y tener poco ancho de banda a la vez. El mensaje puede requerir de dos procesamientos, encriptado y compactado. Como resultado, el *network* deberá componerse con dos objetos, uno que defina la forma de procesamiento según el medio físico (*physical media processing strategy*) y otra que dependa del estado del ancho de banda (*bandwith processing strategy*), dichos objetos se llaman *pmPS* y *bmPS*, respectivamente.
Cada vez que el objeto *network* reciba una solicitud de envío, ambas clases de procesamiento sabrán cómo alterar el contenido del mensaje. En la figura 5.1, vemos un diseño con la nueva definición del mensaje *#deliver:aMessage* de la clase *Network*. La figura 5.2 muestra un diagrama de secuencia que muestra el envío de un mensaje a través de una red inalámbrica con poco ancho de banda. En ese caso, los dos tipos de procesamiento (i.e. el encriptado y el compactado) son aplicados.

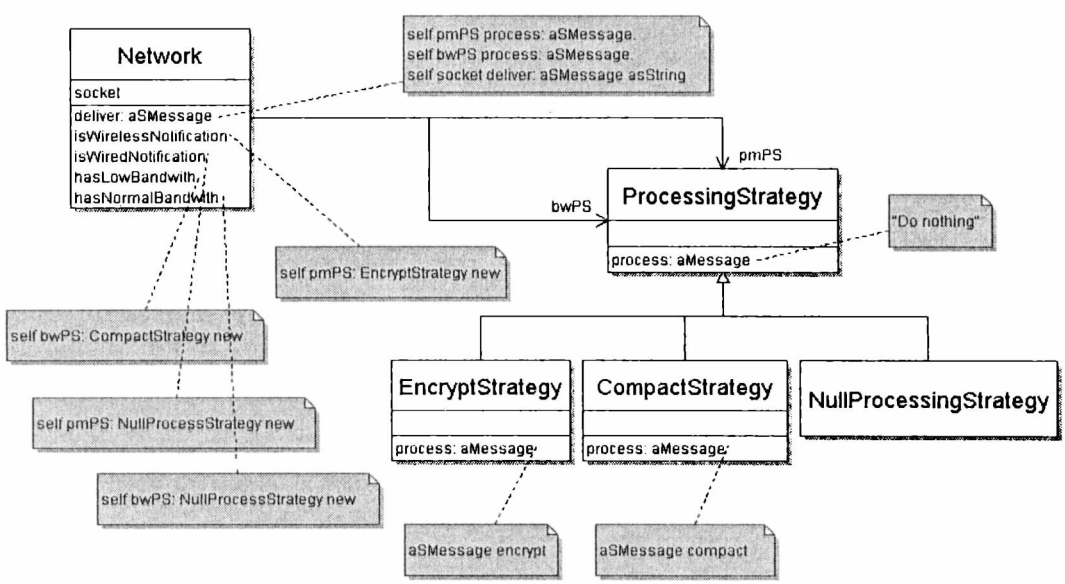

Figura 5.1: Jerarquía de estrategias de procesamiento de mensajes utilizada por la clase *Network*

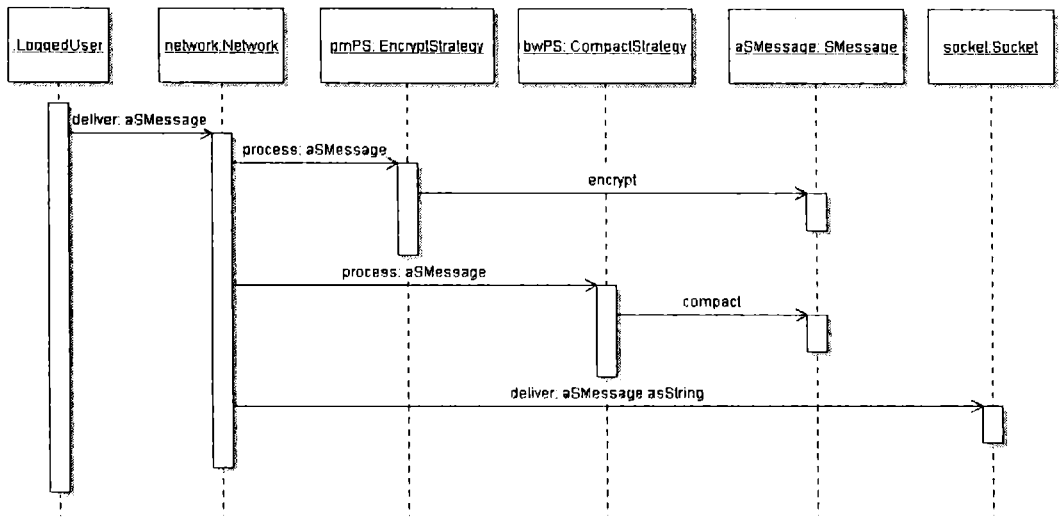

Figura 5.2: Procesamiento al momento de enviar un mensaje.

Prioridad del mensaje a usuarios presentes

Continuando con los requerimientos de adaptación, el número (iii) indica que la prioridad debe cambiarse a *urgente*, si la persona destinataria está presente en la habitación. Dado que los mensajes almacenados en el *network* esperando que la conexión se reestablezca tienen asignadas prioridades, podemos hacer que los mensajes a ciertos usuarios sean considerados primeros para el envío. En particular, debemos mandar los mensajes con alta prioridad, si el destinatario del mensaje se encuentra en la misma habitación. Para ello, vamos a suponer que convive en el mismo dispositivo un objeto de otra aplicación que registra el contexto social del usuario. La clase de dicho objeto se llama *BroaderContext.* El *messenger* deberá acceder al *broaderContext* y reconocer al destinatario entre los usuarios presentes, para luego cambiar la prioridad del mensaje, si es necesario.

Esta acción también puede ser considerada como un procesamiento sobre el mensaje a transmitir. En este caso, el parámetro del mensaje *#deliver: aMessage* y el contexto social del usuario son los influyentes del comportamiento. Como el parámetro del mensaje interviene en la elección de la respuesta del mensaje, no podemos configurar el *messenger* con una estrategia de procesamiento como hicimos en al caso del encriptado. Por lo tanto, utilizaremos una sentencia *if-case* para evaluar el destinatario del mensaje y elegir el comportamiento correspondiente. Otra diferencia con el requerimiento anterior es que, para implementar la condición del mensaje, es necesario referenciar al objeto *broaderContext*, que no es colaborador del *messenger.* Una de las opciones es que sea una variable de instancia del objeto *messenger,* situando al *broaderContext* al mismo nivel que los colaboradores internos del *messenger*. Esto puede resultar confuso, ya que el *broaderContext* no colabora en el desarrollo del comportamiento del mensaje, como sugiere la naturaleza de un colaborador, si no que implementa una condición que determina el comportamiento a realizar. Otra opción, es que el *broaderContext* sea una variable global, permitiendo que todas

las aplicaciones del dispositivo, que también posean requerimientos de adaptación, puedan consultar los usuarios presentes, si es necesario. Esta última alternativa tiene como desventaja que el *broaderContext* puede ser accedido desde cualquier punto, aumentando el riesgo de producir modificaciones indebidas. Por ese motivo, vamos a aplicar la primera alternativa. En la figura 5.3 se muestra un diagrama con los nuevos elementos de diseño incorporados para el modelado del requerimiento. Observemos el detalle del mensaje *Messenger>>#deliver: aMessage*. Primero verifica la presencia del destinatario dentro del contexto. Si esta presente, la prioridad del mensaje es asignada como *urgente* antes de realizar el envío.

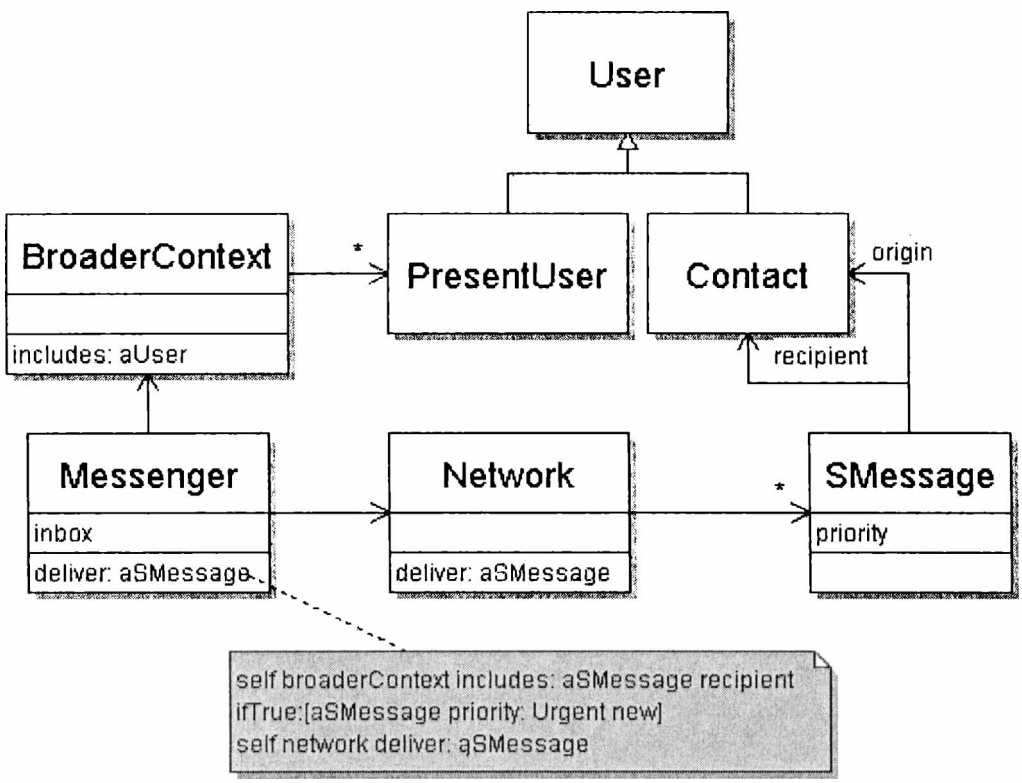

Figura 5.3: Nueva asociación entre el *messenger* y el *broaderContext*. El *messenger* evalúa el *broderContext* para modificar o no la prioridad del mensaje.

Usuario autorizado

Continuando con la lista de requerimientos, el número (iv) indica que el usuario registrado al dispositivo es el único autorizado a enviar mensajes. Supongamos que el usuario registrado esta representado por el objeto *loggedUser* y que además es un *Singleton* [GAM/95], ya que no puede haber dos usuarios registrados a la vez. Dicha instancia será la única autorizada para ejecutar el mensaje *messenger>>deliver: aSMessage*. Para que el *messenger* sepa si el emisor del mensaje está autorizado, debe tener una referencia al emisor y otra a la instancia autorizada. La referencia al *loggedUser* puede ser asignada en la creación del *messenger*, y puede ser conservada en una variable de instancia ya que debe utilizarse cada vez que se invoque el mensaje de envío. En cambio, la referencia

al emisor debe obtenerse en el momento de recepción del mensaje, por lo tanto, es necesario incluir un nuevo parámetro.

Como en el requerimiento anterior, la condición del mensaje depende de un parámetro, con lo cual, debemos utilizar una construcción *if-case* en el cuerpo del método para elegir el comportamiento del mensaje. Además, la modificación del selector *#deliver: aSMessage* para incluir el emisor del mensaje implica modificar el tipo del *messenger*. Por lo tanto, al incorporar este requerimiento al diseño funcional, se generarán cambios que repercutirán en los clientes de la aplicación. Si más adelante el requerimiento de adaptación es eliminado y el parámetro pierde sentido, volver al nombre de selector original del mensaje generará la misma cadena de modificaciones. Sumado a los problemas de modificación del diseño que conlleva la implementación de este requerimiento, tenemos un segundo problema. Ahora al recibir el mensaje *#deliver: aSMessage for. aSender*, el *messenger* asume que un parámetro es el emisor del mensaje. Pero dicha precondición puede no ser cierta, si un emisor no autorizado invoca el mensaje pasando como parámetro la instancia *loggedUser*. En este caso, el emisor no autorizado lograría ejecutar el envío del mensaje, violando el requerimiento de seguridad. A pesar de ello, dado que no es el objetivo de la tesis hacer hincapié en cuestiones de seguridad, damos por concluido el diseño del requerimiento (iv). La figura 5.4 muestra un diagrama con los elementos de diseño agregados y modificados. El mensaje *Messenger>>deliver: aSMessage for: aSender* tiene un parámetro más con respecto al diseño del capítulo anterior. En el comentario del mensaje se detalla el nuevo comportamiento del método. Si el *loggedUser* es el emisor del mensaje *Messenger>>deliver: aSMessage for: aSender,* puede concreta el envío. Si una instancia cualquiera ocupa el rol del parámetro emisor, el *messenger* no realizaría ningún comportamiento.

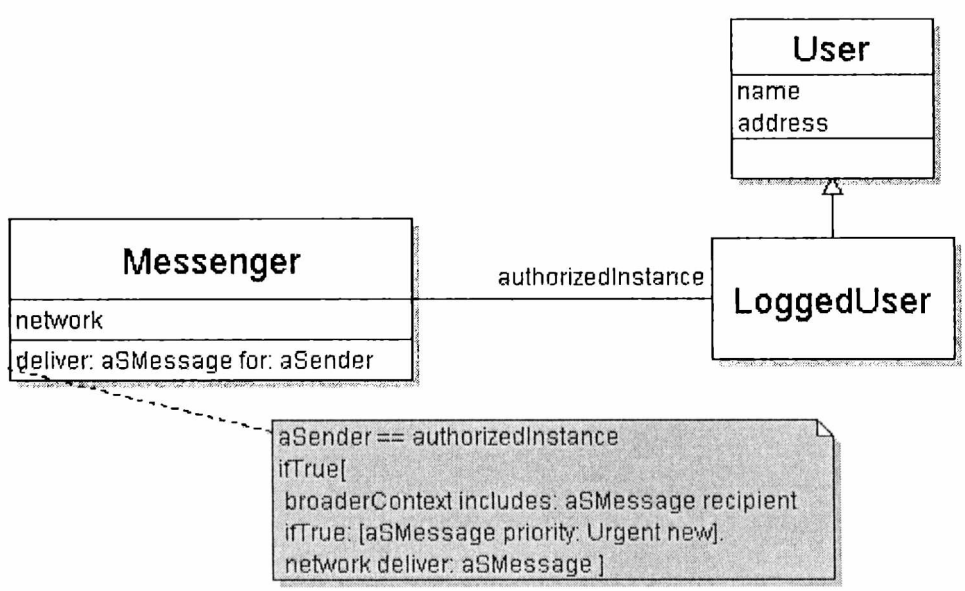

Figura 5.4: Nuevo colaborador externo del mensaje *#deliver: aMessage* y nueva asociación entre el *messenger* y la instancia emisora autorizada. El *messenger* realiza el envío si el emisor del mensaje está autorizado.

Recursos de Batería

Continuemos con el requerimiento número (v). Ahora debemos adaptar el comportamiento de la aplicación a los recursos de batería. Supongamos que el dispositivo posee un objeto *battery* que permite la suscripción de cualquier objeto del ambiente que necesite recibir notificaciones sobre el estado de la batería. Para que el *messenger* pueda ajustar su comportamiento deberá subscribirse en orden de recibir dichas notificaciones. Dado que el cambio de estado de la batería podría influenciar otros mensajes del *messenger* definiremos una jerarquía para definir separadamente los dos modos de comportamiento. Esta decisión de diseño se basa en la solución propuesta por el patrón *State*. De ahora en más, cada operación del *messenger* será delegada a los modos de la batería. La clase que implemente el estado normal se llamará *NormalBatteryMode* y la que implemente el modo para la batería baja se llamará *LowBatteryMode*. Cuando la batería detecta que tiene poca energía, notifica al *messenger* y este cambia su estado de comportamiento por un objeto *lowBatteryMode*. Dicho estado sólo encaminará los mensajes que posean direcciones de emergencia y además suspenderá la recepción de mensajes. Cuando la batería se restablezca, mandará una notificación al *messenger* para que vuelva a cambiar su estado a *normalBattery*. La figura 5.5 muestra la nueva jerarquía de clases que representa los estados del *messenger*. Notemos que, como los cambios de comportamiento se basan en notificaciones externas, no fue necesario incluir construcciones *if-case* en el cuerpo del mensaje.

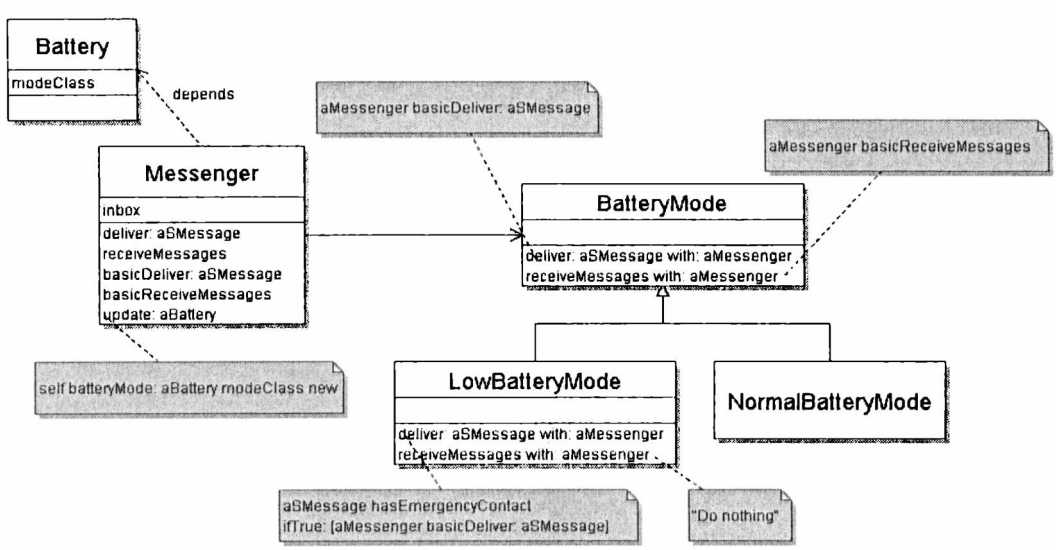

Figura 5.5: Jerarquía de comportamiento del messenger para cada estado de batería.

El *messenger* delega el envío y la recepción de mensajes al estado de batería del dispositivo. Si el estado de la batería es normal o si la dirección del mensaje es de emergencia, el estado de batería vuelve a delegar en el *messenger* el envío del mensaje. Lo mismo sucede con la recepción de mensajes, cuando el estado de batería es normal.

Salas de reunión

El siguiente requerimiento, el número (vi), es similar al requerimiento anterior, pero ahora la aplicación deberá adaptarse a notificaciones de dispositivos de salas de reunión. Cuando un dispositivo detecte que se ha ingresado a reunión importante, el *messenger* deberá deshabilitar el timbre de arribo de mensajes. Este nuevo requerimiento también precisa que la aplicación cambie de comportamiento debido a una notificación externa, por lo tanto, el diseño será similar al caso anterior. Crearemos otra jerarquía de estados con dos clases concretas: *RingtoneMessageMode* y *SilenceMessageMode* (Ver figura 5.6). Dado que los mensajes sólo pueden recibirse, si la batería del dispositivo tiene recursos, haremos depender los nuevos modos de comportamiento del *normalBatteryMode*. Cuando comience una reunión importante, el *messenger* delega el mensaje *#startMeeting: aMeeting* al *normalBatteryMode,* que cambiará su estado a *silenceMessageMode.* A partir de ese momento, el estado de batería normal enviará los mensajes al *messenger* de manera silenciosa. El *lowBatteryMode* ignora la notificación delegada por el *messenger*.

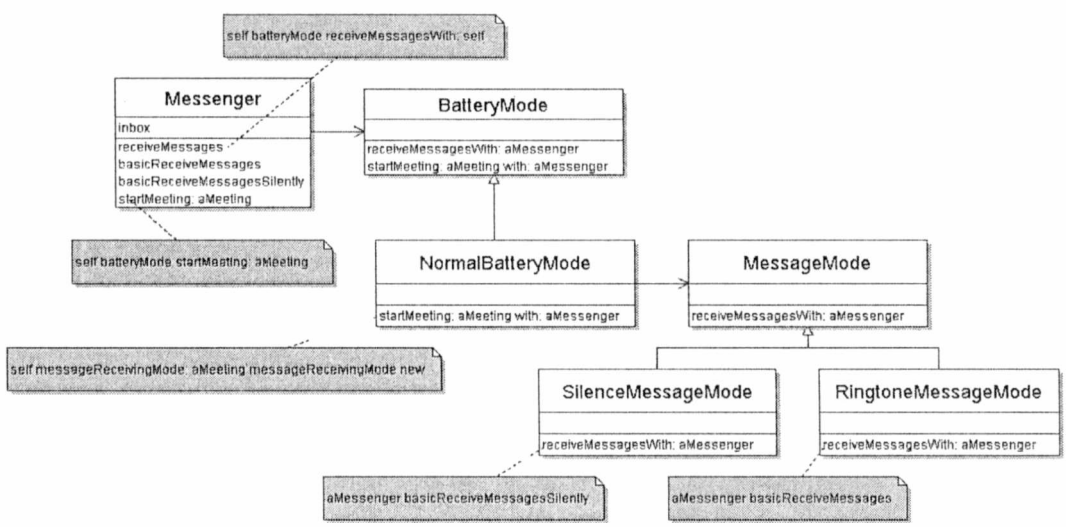

Figura 5.6: El estado de batería normal tiene dos modos de comportamiento que cambian con las notificaciones de los dispositivos de sala.

Aplicaciones de Red

El requerimiento de adaptación número (vii) pretende que nuestra aplicación pueda combinarse con otros módulos existentes que implementen diversos protocolos para encaminar mensajes a través de un medio de comunicaciones. Además dicha capacidad de adaptación debe ser dinámica, si se conocen los módulos de antemano. En nuestro diseño la funcionalidad para encaminar información a la red está encapsulada en el objeto *socket.* Una forma de permitir la flexibilidad propuesta por el requerimiento, es permitir que el *network* pueda interactuar con *sockets* diseñados por otros desarrolladores, y que por lo tanto puedan tener distinto tipo. Para ello, vamos a aplicar una solución similar a la capacidad dinámica vista en el capítulo 3. Dependiendo del tipo del *socket*,

realizaremos la invocación correspondiente, utilizando sentencias if-case. Otra solución mucho más eficiente podremos conseguir utilizando el patrón *Adapter*, y haciendo la reconfiguración del adaptador de manera automática cuando se asigna un nuevo *socket*. De esta forma, nos ahorramos verificar el tipo del colaborador cada vez que tengamos que invocar un mensaje del socket. La figura 5.7 muestra como queda nuestro diseño incorporando el mecanismo de adaptación explicado

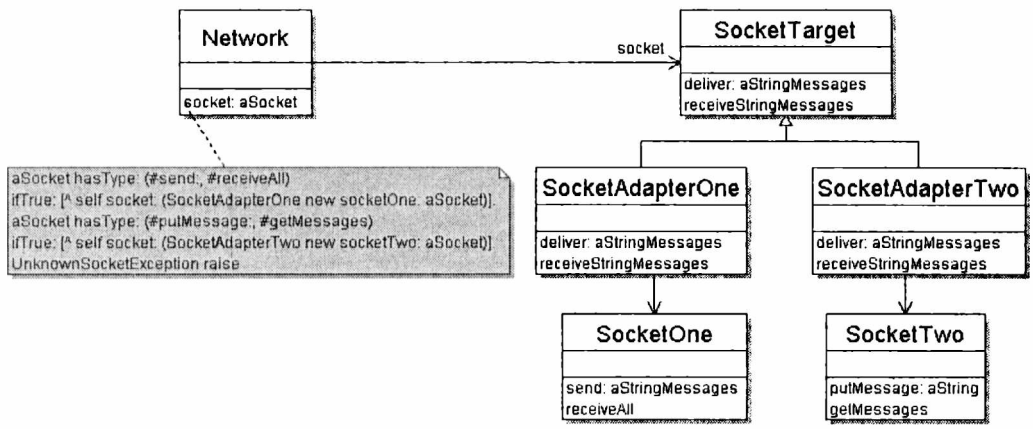

Figura 5.7: Mensaje de inicialización del *socket*. Cuando el *socket* es asignado, el *network* revisa su tipo y crea el adaptador correspondiente.

En el caso del ejemplo, el *network* esta preparado para colaborar con dos tipos de *sockets SocketOne* y *SocketTwo*. Dichos *sockets* realizan funciones similares, pero fueron construidos en distinto momento y por distintas personas, por ese motivo los nombres de los mensajes no son los mismos. Los adaptadores respectivos, implementan la traducción de la interfase esperada por el *network* a los mensajes que entiende el *socket* utilizado. Analizando el tipo del *socket*, se distingue que adaptador es necesario utilizar.

Notemos que el mensaje *#hasType* es una operación reflexiva que examina la clase del receptor y verifica si este implementa el conjunto de selectores recibido como parámetro. Esta función esta incluida dentro de la máquina subjetiva, ya que conceptualmente, el hecho de verificar el tipo de los objetos que forman el contexto de recepción, es parte de la definición de subjetividad. En el capítulo siguiente, notaremos que el modelado subjetivo de este requerimiento es similar al recién obtenido.

Preferencias del usuario

El último requerimiento señala que la aplicación debe poder configurarse según las preferencias del usuario. Cuando el usuario se registra en el dispositivo, además de recuperar sus datos personales, obtiene la lista de las preferencias de configuración. Esta lista de preferencias está modelada con una colección de selectores[3]. Una vez que la lista es recibida, los mensajes que representan los selectores son enviados al objeto *messengerConfigurator* encargado de configurar el *messenger*. Dicho objeto implementa todos los mensajes de configuración que el usuario puede enviar. La figura 5.8 muestra la colaboración entre el

[3] Un *selector* en *Smalltalk* es un *Symbol* que identifica un mensajes dentro de una clase.

registeredUser y el *messengerConfigurator*. El *registeredUser* instancia un *MessengerConfigurator* y solicita la ejecución de la lista de preferencias que tiene almacenadas. El *messengerConfigurator* ejecuta las acciones necesarias sobre el *messenger* para realizar la configuración.

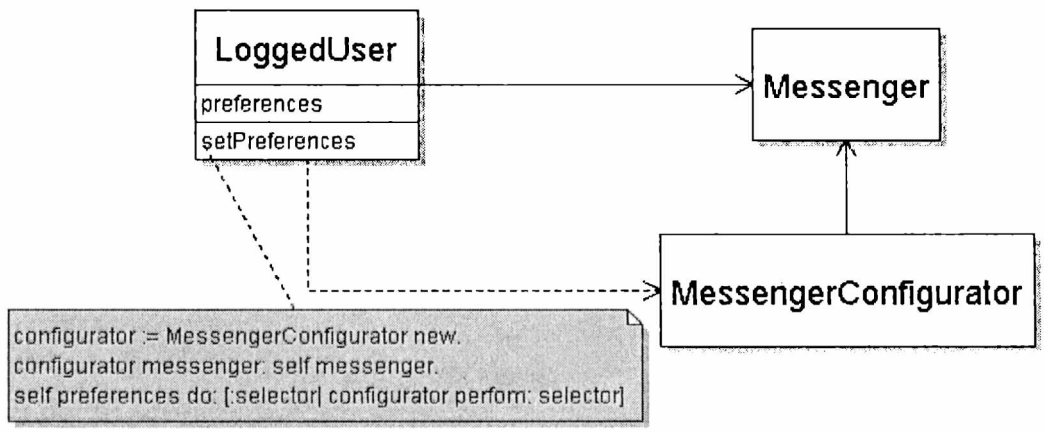

Figura 5.8: Diagrama de clases que muestra la relación entre el usuario registrado y el configurador de la transmisión de mensajes.

Para permitir que la actividad de reconfiguración pueda realizarse, debemos reorganizar el diseño según la lista de preferencias enunciadas en el capítulo anterior:

a. mandar mensajes con prioridad urgente a los usuarios presentes.
b. no cambiar la prioridad de los mensajes,
c. encriptar siempre los mensajes enviados,
d. encriptar los mensajes, si la red es inalámbrica,
e. no encriptar nunca los menajes,
f. no compactar mensajes,
g. compactar mensajes cuando la conexión es lenta,
h. siempre notificar la recepción de mensajes,
i. y no notificar la recepción de mensajes en reuniones importantes.

Preferencias de prioridad de envío

Para implementar las preferencias a y b, tendremos que permitir dos formas de envío. Una debe evaluar el *broaderContext* y cambiar la prioridad a *urgente* antes de enviar el mensaje y la otra debe enviarlo sin modificar la prioridad. En la figura 5.9 y 5.10 se muestra la jerarquía de clases resultante y los mensajes del *messengerConfigurator* que implementan ambas directivas realizando la configuración. El mensaje *Messenger>>basicDeliver: aSMessage* delega a la política de prioridades de envío el cambio de prioridad del mensaje. La política de envío verifica el *broaderContext* y cambia la prioridad a urgente, si el destinatario

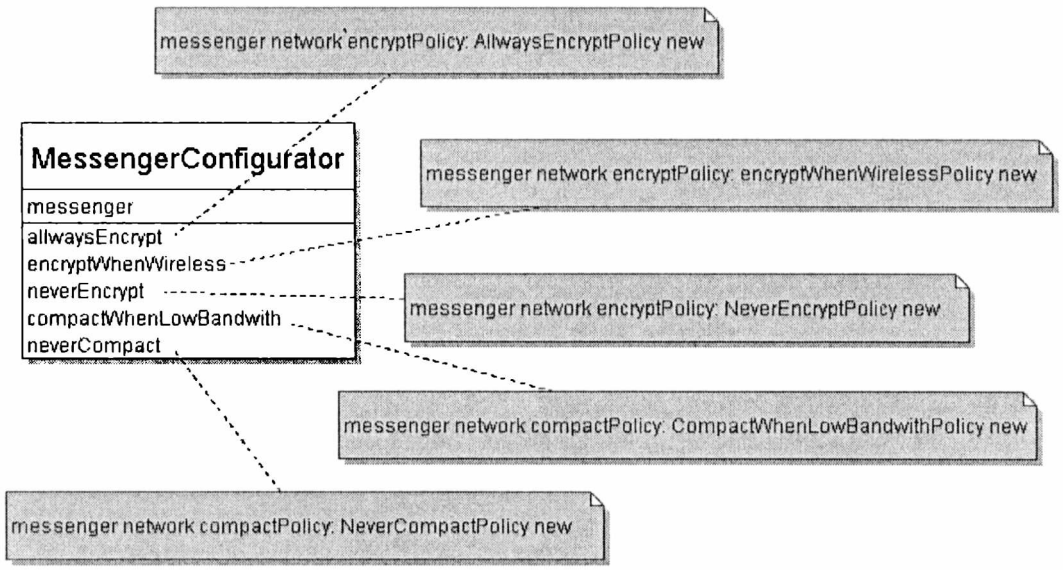

Figura 5.13: Directivas que configuran las políticas de encriptado y compactado.

Preferencias de recepción de mensajes

Por último, para poder aplicar las preferencias h e i, debemos poder configurar el *messenger* con dos políticas alternativas de recepción de mensajes: deshabilitar el timbre de aviso de mensajes, si el usuario se encuentra en una reunión importante o nunca deshabilitar el timbre de arribo de mensajes. Para ello necesitamos modificar el diseño del *normalBateryMode* de manera de que actúe según la política de recepción de mensajes elegida por el usuario. El diseño resultante se muestra en la figura 5.14. El estado de batería normal puede tener dos Políticas de recepción de mensajes. La política *SilenceWhenImportantMeetingPolicy* asigna el modo de recepción silencioso cuando la reunión es importante. La figura 5.15 se muestra la implementación de las directivas.

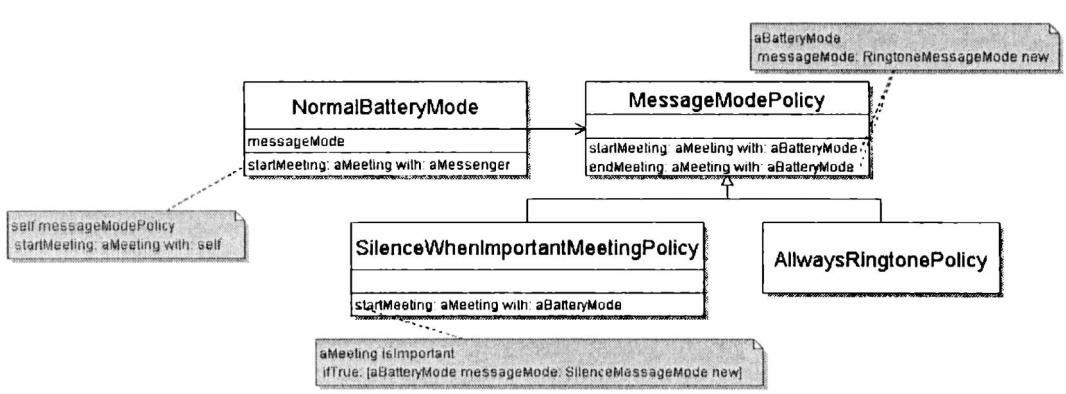

Figura 5.14: Políticas de recepción de mensajes al comienzo de una reunión.

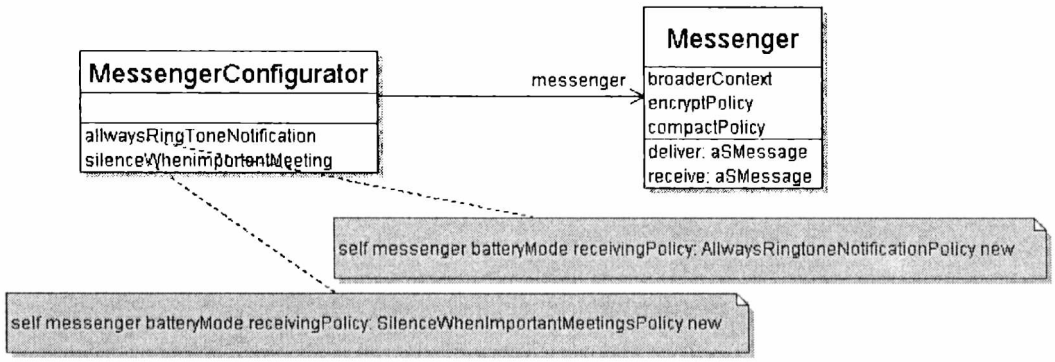

Figura 5.15: Directivas para configurar las políticas de recepción de mensajes.

5.3. CONCLUSIÓN

El diseño orientado a objetos obtenido desacopla, en líneas generales, los diferentes comportamientos del *messenger*, que surgen de los distintos requerimientos de adaptación. Esto hace que los diferentes comportamientos sean intercambiables e independientes y que, en general, la variedad del comportamiento del messenger esté encapsulada dando transparencia el cliente. Con este fin, se introdujeron un gran número de clases, relaciones y mensajes al diseño de requerimientos funcionales del capítulo anterior. Tuvimos dificultades para separar los distintos comportamientos en jerarquías cuando la decisión de la respuesta dependía de un parámetro del mensaje. En estos casos no fue posible configurar el comportamiento adecuado antes de la ejecución del mensajes, por lo tanto hemos aplicado construcciones *if-case* en el cuerpo del método. Otra alternativa hubiera sido utilizar diccionarios con cada caso de comportamiento. Como vimos en el capítulo 3, estas estructuras se utilizaron para implementar el comportamiento subjetivo en los mensajes.

La figura 5.16 es un diagrama que suma todos los elementos de diseño discutidos. Notemos que los elementos de diseño para los requerimientos funcionales y de adaptación se entrelazan al mismo nivel, opacando la lectura de los requerimientos funcionales lograda en el diagrama de requerimientos funcionales Ver la figura 4.5. Esto sucede porque los requerimientos de adaptación son modelados con los mismos recursos utilizados para los requerimientos funcionales. En la siguiente sección presentaremos otro diseño de los mismos requerimientos de adaptación, pero utilizando comportamiento subjetivo. Se espera obtener un diseño más simple, al no tener la necesidad de introducir nuevos elementos de diseño para desacoplar los distintos comportamientos de un mensaje. Además no necesitaremos tantas relaciones de colaboración, ya que tendremos a nuestra disposición cualquier objeto de la aplicación para implementar las condiciones de adaptación. Como resultado se pretende obtener un diseño similar al diseño derivado de los requerimientos funcionales.

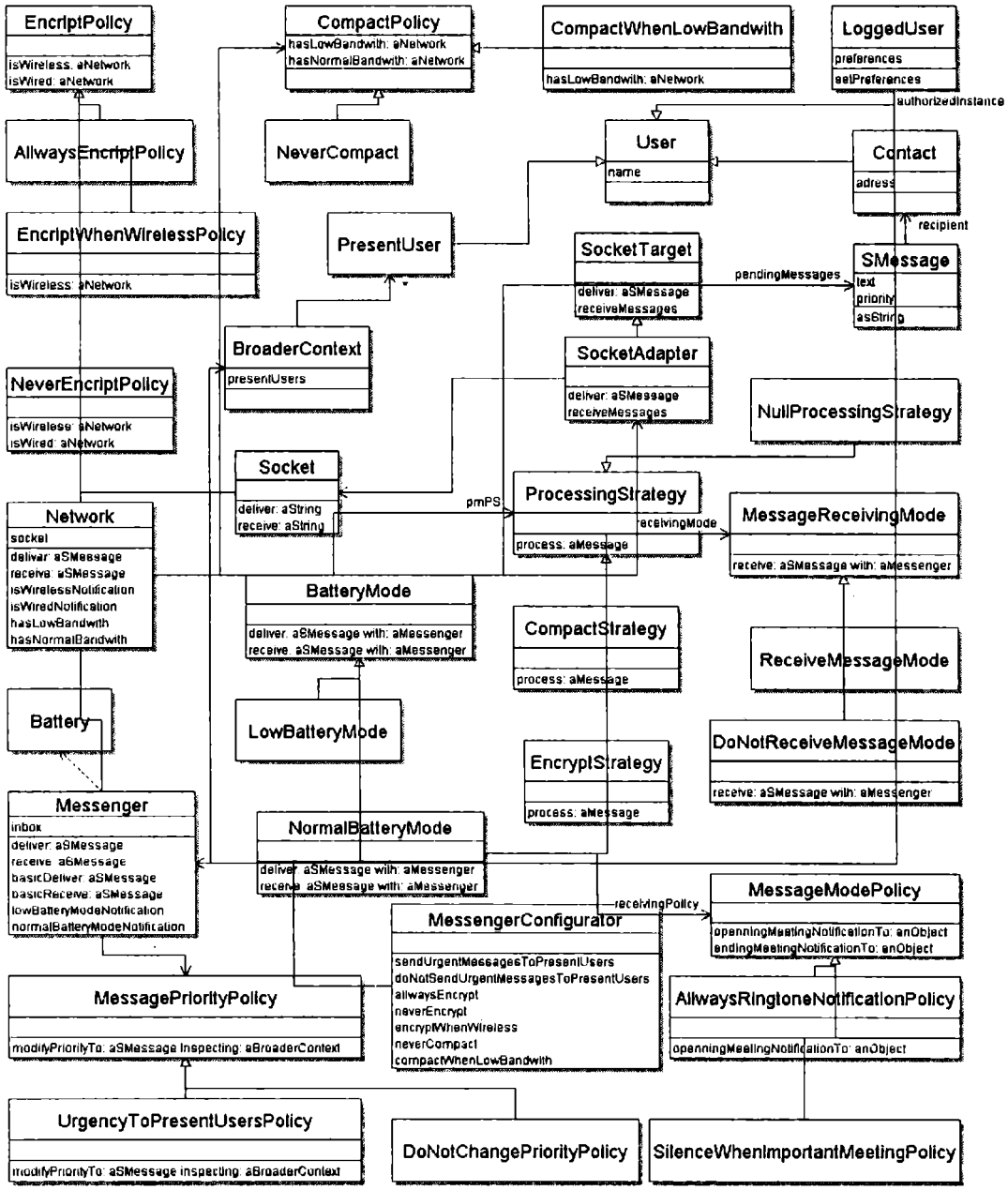

Figura 5.16: Diagrama con todos los elementos de diseño utilizados para modelar la aplicación.

Capítulo 6

DISEÑO CON COMPORTAMIENTO SUBJETIVO

6.1. INTRODUCCIÓN

En este capítulo presentaremos un segundo diseño de los requerimientos de adaptación resueltos en el capítulo pasado. Ahora utilizaremos comportamiento subjetivo como mecanismo de adaptación auxiliar. Recordemos que, a diferencia de las sentencias *if-case* utilizadas en el capítulo anterior, el comportamiento subjetivo desacopla e independiza automáticamente los diferentes comportamientos de un mensaje, por lo tanto, no será necesario realizar esfuerzos de diseño en ese sentido.

6.2. DISEÑO DE REQUERIMIENTOS DE ADAPTACIÓN

Partimos del diseño de requerimientos funcionales realizado en el capítulo 4. La figura 4.5 muestra el diseño funcional antes de sumar los elementos para los requerimientos de adaptación de la aplicación. A continuación, modelaremos nuevamente dichos requerimientos, pero ahora utilizando comportamiento subjetivo.

Encriptado y compactado de mensajes

Recordemos que los requerimientos de adaptación (i) y (ii) nos indicaban que los mensajes debían encriptarse, si la red de comunicaciones es inalámbrica y compactarse, si la red poseía poco ancho de banda. Habíamos dicho que estos requerimientos eran diferentes formas de procesar el mensaje y que dependían del

estado del medio de comunicaciones. Con el paradigma subjetivo, podemos tener un mensaje que encapsule las alternativas de procesamiento del mensaje antes de ser enviado. Llamaremos a este mensaje *#process: aSMessage*. Como vimos en la sección anterior, los procesamientos posibles son: encriptar el mensaje, compactar el mensaje o no efectuar ningún procesamiento. Cuando la conexión se establece el *socket* informa al *network* el estado de la red. El *network* almacena dicho estado y comienza a mandar los mensajes encolados. Luego, el mensaje *#process: aSMessage* deberá variar su estrategia dependiendo del estado de la red. Notemos además que los procesamientos deben sumarse, si la red tiene poco ancho de banda y es inalámbrica. Por ese motivo, definiremos la secuencia de métodos agregables del mensaje para que opere de la siguiente manera: si la red es inalámbrica y posee poco ancho de banda, el mensaje debe encriptarse y ser compactado; si es inalámbrica, pero el ancho de banda es normal, sólo debe encriptarse; si no es inalámbrica y tiene poco ancho de banda, deberá compactarse y por último, si tiene ancho de banda normal y no es inalámbrica no deberá efectuarse ningún procesamiento. La figura 6.1 muestra como queda el diseño de la clase *Network* con comportamiento subjetivo. El mensaje *#process: aSMessage* tiene definida una secuencia con dos métodos agregables. Recordemos que los métodos agregables los indicábamos con la palabra clave *method* y que el comportamiento de estos mensajes se ejecutaba sólo si su condición es verdadera. Si varios métodos agregables pueden ejecutarse en una invocación, se ejecutan todos de manera secuencial según fueron definidos. Si ninguno puede ejecutarse, ningún comportamiento es realizado. En nuestro ejemplo, si las dos condiciones son verdaderas, se ejecutan los dos procesamientos sobre el mensaje. Por contrario, si ninguna se cumple, el mensaje no realiza ninguna acción.

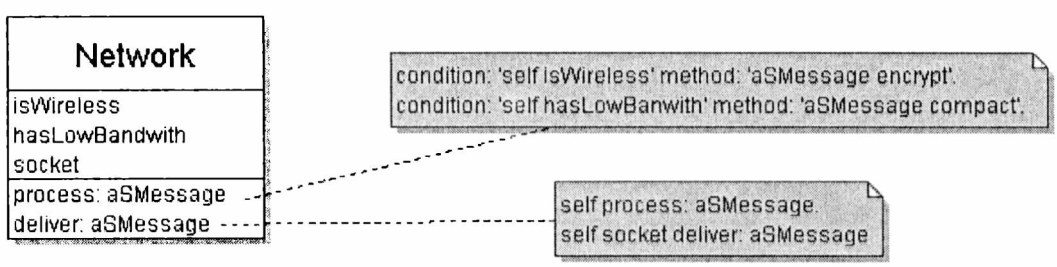

Figura 6.1: La clase Network define su mensaje #process de manera subjetiva. Cada estrategia de procesado se define en un método agregable distinto.

Notemos que la lógica condicional del mensaje es compleja, pero con comportamiento subjetivo su definición es sencilla. Tiene los beneficios de mantenimiento que conlleva la independencia de las implementaciones, pero no fue necesario para ello crear jerarquías de clases como en la solución del capítulo anterior.

Prioridad del mensaje a usuarios presentes

El requerimiento de adaptación (iii) indica que la prioridad del mensaje debe ser asignada como urgente, si la persona destinataria está presente en la habitación. Se había considerado que, dentro del software del dispositivo, existe un objeto que registra el contexto social del usuario. Dicho objeto se llamaba

broaderContext. Una de sus responsabilidades era mantener la información de los usuarios que se encuentran en la misma habitación.

Para modelar este requerimiento podemos considerar al cambio de prioridad como otro procesamiento del mensaje y cambiar la definición del mensaje subjetivo *Messenger>>#process: aSMessage* insertando en la secuencia un nuevo método. Dicho método cambiará la prioridad del mensaje a urgente, si el destinatario del mensaje está presente en la habitación. La condición de este nuevo método se implementa consultando el objeto *broaderContext*. Para que la máquina subjetiva encuentre la referencia a este objeto es necesario declararlo previamente como *acquaintance* del *network*, tal como se indica en el capítulo 3. La figura 6.2 muestra como queda definido el mensaje subjetivo con el nuevo requerimiento.

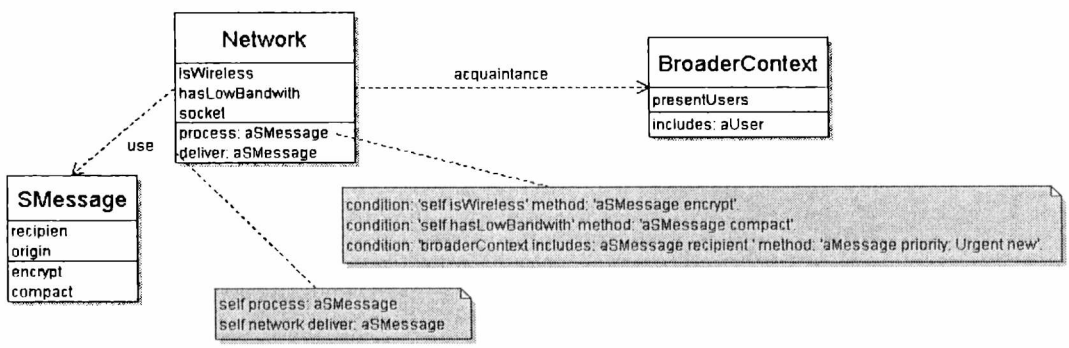

Figura 6.2: El mensaje subjetivo *#process* tiene un nuevo método agregable que modifica la prioridad del mensaje según el *broaderContext*.

Notemos con que facilidad fue posible agregar una nueva porción de comportamiento al mensaje *#process: aSMessage*. Al ser un método agregable este se suma a los comportamientos anteriores, pero conservando su independencia. Esta independencia hace que cualquier método agregable puede removerse o adjuntarse, sin influir en la implementación de los otros métodos de un mensaje.

Emisor autorizado

Pasemos ahora al requerimiento de adaptación número (iv). Este requerimiento decía que el usuario registrado era el único capaz de enviar mensajes. Por lo tanto, sólo la instancia *loggedUser* podía concretar el envío al ejecutar el mensaje *messenger>>deliver: aSMessage*. Recordemos que con comportamiento subjetivo el emisor del mensaje era accesible para construir las condiciones del mensaje. En la figura 6.3 vemos como queda definido el mensaje *Messenger>>deliver: aSMessage*. Tiene una única condición y un método agregable asociado. La condición verifica que el emisor del mensaje sea la instancia *authorizedInstance*. Si la condición se cumple, el envío es delegado al *network*. Si no se cumple, el messenger no realiza el envío.

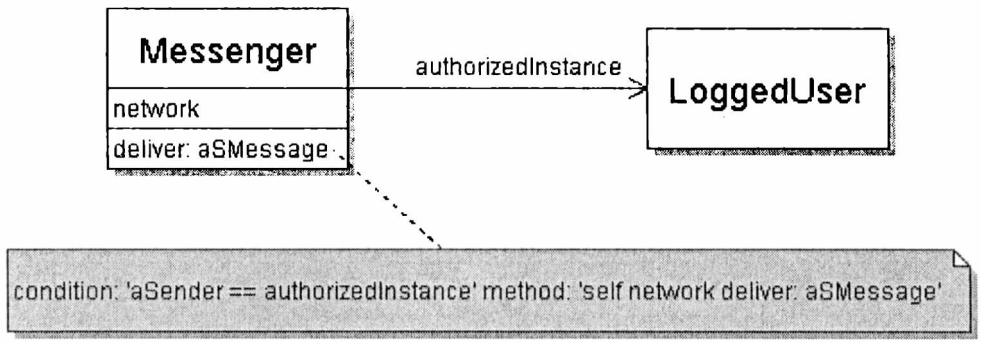

Figura 6.3: El mensaje subjetivo *#deliver:aSMessage* tiene un único método agregable que envía el mensaje a la red, si el emisor es la instancia autorizada.

Dado que el emisor del mensaje puede accederse para implementar condiciones, no fue necesario tener un nuevo parámetro en el mensaje. Como consecuencia, pudimos conservar el tipo del *messenger* definido para los requerimientos funcionales. Esto hace que las modificaciones introducidas por los requerimientos de adaptación sean transparentes para los clientes de la aplicación.

Recursos de batería

El requerimiento de adaptación (v) sugería adaptar el comportamiento de la aplicación según los recursos de batería. Si el dispositivo tenía poca batería, suspendía la recepción y el envío lo reservaba sólo para las direcciones de emergencia. Habíamos dicho también, que el dispositivo poseía un objeto *battery,* con información del estado de la batería. Con comportamiento subjetivo, el objeto *battery* será un *acquaintance* del *Messenger.* Dado que el recurso de batería afecta la recepción y el envío de mensajes, agregaremos nuevas condiciones a los correspondientes métodos agregables. El envío de mensajes se produce siempre que haya batería o si la dirección de mensajes es de emergencia, en tanto que la recepción únicamente se produce si el estado de batería es normal. La figura 6.4 muestra como queda el método agregable del envío y el nuevo método agregable para la recepción.

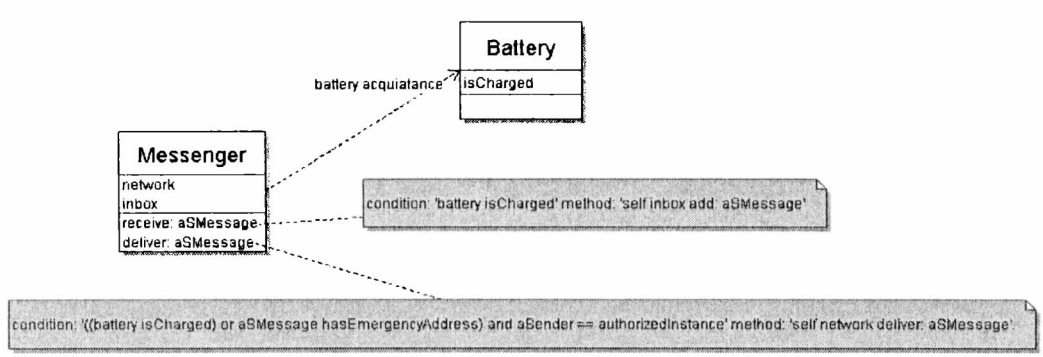

Figura 6.4: El mensaje subjetivo *#deliver:aSMessage* posee una nueva condición que verifica el estado de la batería.

Por ahora, la lógica del mensaje sigue siendo sencilla. Si al agregar más requerimientos de adaptación, se deben agregar más condiciones y la lógica de las condiciones del mensaje se torna complicada, es conveniente dividir el mensaje en dos mensajes subjetivos.

Notemos que no fue necesario agregar una jerarquía de comportamiento para encapsular el modo de comportamiento. El paradigma subjetivo, resuelve la aplicación de comportamiento a través de condiciones dentro del mensaje. Esto hace que no sea necesario desglosar las diferencias de comportamiento en nuevas clases, haciendo el diseño más compacto.

Salas de reunión

En el requerimiento de adaptación número (vi) el messenger debía adaptarse a notificaciones de dispositivos de sala. Era similar al requerimiento anterior en el cual el *network* debía adaptarse a notificaciones del estado de la batería. Ahora el messenger deberá suspender el aviso de llegada de mensajes, si el usuario se encuentra en una reunión importante. Para la solución con comportamiento subjetivo, suponemos que dichas notificaciones son detectadas por el dispositivo y almacenadas en el objeto *broaderContext,* que utilizamos en el requerimiento (iii). Cuando el usuario entra a una reunión, la propiedad *userInMeeting* del *broaderContext* tomará el valor verdadero y cuando la reunión termina tomará el valor falso. El *messenger* suspenderá el timbre de arribo de mensajes dependiendo de dicha propiedad y del tipo de reunión en la que se encuentre. Por lo tanto, es necesario que el *broaderContext* también sea *acquaintance* del *messenger.* Además, definiremos dos métodos agregables. Uno que agregue los mensajes en la bandeja de entrada sin desactivar el timbre y otro que los agregue silenciosamente, si el destinatario esta incluido en el *broaderContext.* La figura 6.5 muestra el diseño del mensaje *#receiveMesseges* con los nuevos métodos.

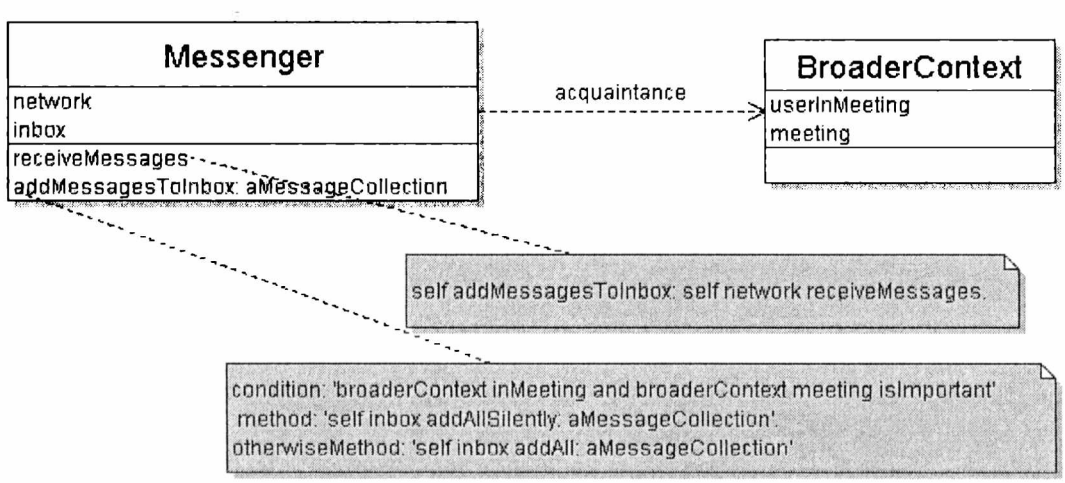

Figura 6.5: El primer método del mensaje subjetivo *#addMessageToInbox* verifica si el dispositivo se encuentra en una reunión importante. Si esto no ocurre, se ejecuta el comportamiento del método otherwise.

Notemos que como el caso anterior, no fue necesaria una jerarquía de comportamiento para separar las distintas formas de recepción de mensajes.

Aplicaciones de Red

El requerimiento de adaptación (vii) pretendía que la aplicación sea configurable para colaborar con cualquier aplicación que transmita información a través de un tipo de red particular. Para soportar este requerimiento, el modelo del capítulo anterior permitía que el *network* interactúe con objetos *sockets* de diversos tipos utilizando un objeto adaptador [GAM/95]. Además, si el objeto *network* utilizado poseía un tipo alguna vez configurado, nuestra aplicación debía adaptarse automáticamente. En el diseño del capítulo anterior, habíamos redefinido el mensaje de asignación del *socket* para que en dicho momento se analizara el tipo y se eligiera el adaptador de comportamiento apropiado para las futuras interacciones. La solución subjetiva de este capítulo es similar. Ahora, el mensaje *#socket: aSocket* será definido de manera subjetiva. Dicho mensaje tendrá una secuencia de métodos *otherwise* con el adaptador para cada tipo conocido. La figura 6.6 muestra el diseño de la clase *Network*.

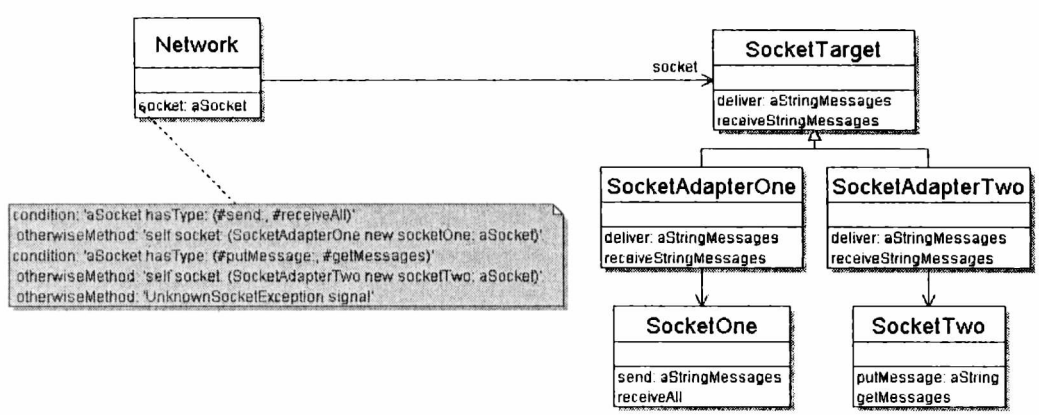

Figura 6.6: El mensaje de asignación del *socket* elige el adaptador correspondiente dependiendo del tipo del *socket* recibido. Si ningún método *otherwise* aplica, eleva una excepción.

Notemos que cada método *otherwise* realiza la configuración para cada tipo conocido Para evaluar el tipo de un objeto la máquina subjetiva provee el mensaje *#hasType: selectors*. Como habíamos visto, este mensaje recibe una colección de selectores y retorna verdadero, si la clase del receptor implementa todos los selectores de la lista.

Para configurar la aplicación con un nuevo tipo de adaptador deberá agregarse un nuevo método *otherwise* en el mensaje *#socket: aSocket* del *network*. Si ninguna condición del mensaje es cierta, aplica el último método *otherwise* elevando una excepción.

Preferencias del usuario

El último requerimiento decía que la aplicación debía poder configurarse con las preferencias del usuario. Habíamos visto que cuando el usuario se registraba en el dispositivo, la aplicación recuperaba la lista de sus preferencias de configuración. Estas preferencias estaban modeladas como una colección de selectores. Una vez que la lista era recibida, los mensajes eran enviados al objeto *messengerConfigurator,* que se encargaba de la configuración.

Recordemos que en el capítulo 3 se discutieron mensajes reflexivos, que permiten modificar las condiciones y las respuestas de un mensaje subjetivo. El *messengerConfigurator* podría utilizar dichos mensajes para cambiar el comportamietno de la aplicación de acuerdo a las preferencias elegidas por el usuario. Analicemos el diseño de cada preferencia la configuración. Recordemos la lista opciones que podía elegir el usuario:

a. mandar mensajes con prioridad urgente a los usuarios presentes.
b. no cambiar la prioridad de los mensajes,
c. encriptar siempre los mensajes enviados,
d. encriptar los mensajes, si la red es inalámbrica,
e. no encriptar nunca los menajes,
f. no compactar mensajes,
g. compactar mensajes cuando la conexión es lenta,
h. siempre notificar la recepción de mensajes,
i. y no notificar la recepción de mensajes en reuniones importantes.

Preferencias de prioridad de envío

Habíamos dicho que para implementar las preferencias a y b, teníamos que permitir dos formas de envío. Una forma debía cambiar la prioridad a urgente, si el destinatario se encontraba en la misma habitación y la otra no debía modificar la prioridad ya establecida. Teniendo en cuenta las técnicas reflexivas vistas, el *messengerConfigurator* debe modificar la secuencia de métodos agregables del mensaje *Network>>process: aSMessage* según la preferencia indicada.

Si el usuario quiere aumentar la prioridad de envío a los usuarios presentes, el *messengerConfigurator* ejecutará *#sendUrgentMessagesToPresentUsers* que agregará un método agregable al mensaje *Network>>process: aSMessage* para que realice dicho cambio de prioridad. Si el usuario prefiere que no se cambien las prioridades de los mensajes, el *messengerConfigurator* ejecutará el mensaje *#doNotChangeMessagePriority* borrando el método *Network>>process: aSMessage* que realiza el cambio de prioridad. La figura 6.7 muestra como queda la definición de ambos mensajes del *messengerConfigurator*.

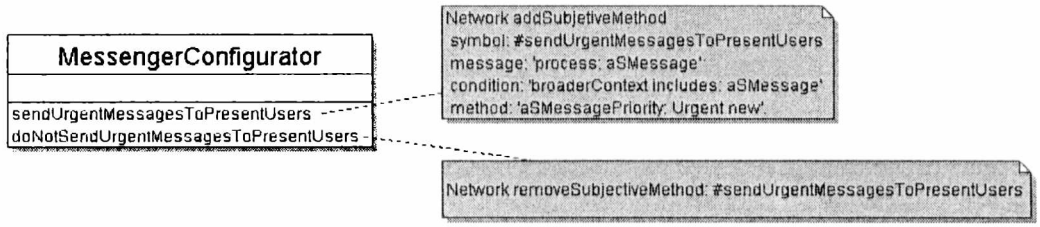

Figura 6.7: La directiva de *sendUrgentMessagesToPresentUsers* del *MessengerConfigurator* adjunta el método agregable *sendUrgentMessagesToPresentUsers*. La directiva

Preferencias de encriptado y compactado

Siguiendo con la lista de directivas, habíamos visto que las configuraciones c, d y e, marcaban diferentes políticas de encriptado, en tanto que f y g, eran políticas de compactado. En el diseño de la aplicación visto el mensaje es encriptado automáticamente, si el medio es inalámbrico. Lo mismo sucede con el compactado, si la red tiene poco ancho de banda.

Para incorporar los cambios de la política de encriptado elegida por el usuario, el *messengerConfigurator* deberá modificar también la secuencia de métodos del mensaje *Network>>process: aSMessage*. La figura 6.9 muestra como quedan implementadas las directivas de cambio de política para el encriptado. Si la directiva es *allwaysEncrypt* el *messengerConfigurator* agrega al mensaje un método agregable con condición siempre verdadera que encripta el mensaje recibido como parámetro. Si la directiva es *encryptWhenWireless*, el *messengerConfigurator* saca el método que siempre encripta y agrega un método que encripta cuando la red es inalámbrica. Por último, si la directiva es *neverEncrypt,* el *messengerConfigurator* saca todos los dos métodos agregables que encriptan los mensajes. Si estos métodos no estuviesen, la directiva no provocaría ningún cambio.

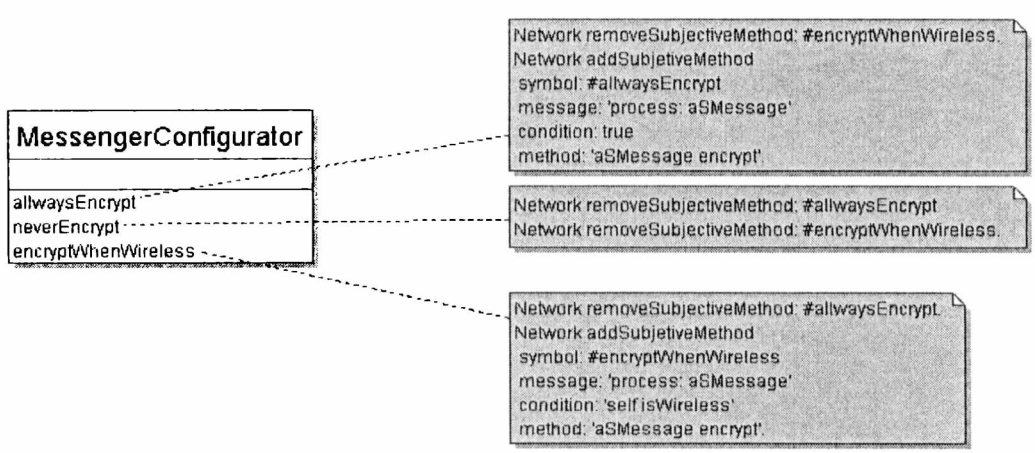

Figura 6.8: La directiva *allwaysEncrypt* adjunta el método agregable que encripta siempre y borra el que encripta sólo cuando la red es inalambrica. La directiva *encriptWhenWireless* hace lo inverso. Finalmente la directiva *neverEncript* borra los dos métodos agregables que encriptan.

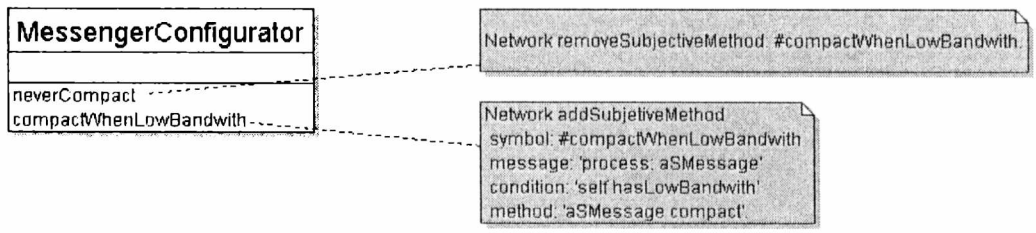

Figura 6.9: La directiva *compactWhenLowBandwith* adjunta el método agregable que compacta el mensaje si la red tiene poco ancho de banda. La directiva *neverCompact* lo borra.

Preferencias de recepción de mensajes

La preferencia (h) era siempre notificar la recepción de mensajes y la preferencia (i) era notificar la recepción de mensajes cuando el usuario se encontraba en una reunión importante. El diseño construido hasta ahora, suspende los mensajes si el usuario se encuentra en una reunión importante. Debemos permitir que el usuario pueda elegir entre aplicar esa política, o la de recibir siempre los mensajes con notificación. Para ello, el *messengerConfigurator* deberá modificar la secuencia de métodos del mensaje *Messenger>>sendMessagesToInbox*. Si suponemos que inicialmente dicho mensaje siempre envía los mensajes recibidos a la bandeja de entrada activando un timbre y que esta funcionalidad esta dada por un método *otherwise* que tiene una condición siempre verdadera. El *mesengerConfigurator* podría definir las directivas de configuración como lo indica la figura 6.10. Cuando la directiva *receiveMessageSilentlyInMeetings* es ejecutada modifica la secuencia del *sendMessagesToInbox* agregando en la primera posición un método que recibe en modo silencioso, si el usuario se encuentra en una reunión importante. Si este método aplica en la invocación del *receiveMessageSilentlyInMeetings* el *otherwise* que envía los mensajes activando el timbre no será ejecutado. La directiva *receiveMessageWithRingtone* borra simplemente el método que recibe en modo silencioso, dejando sólo en la secuencia el método *otherwise*.

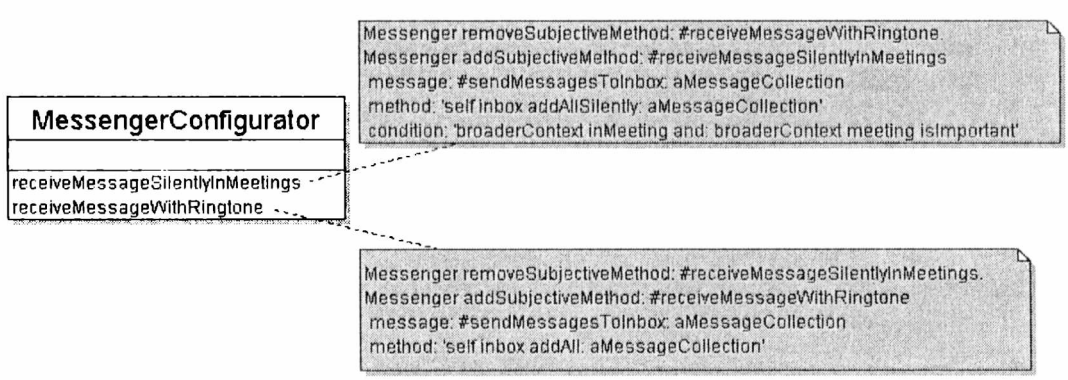

Figura 6.10: La directiva *receiveMessageAllways* adjunta el método agregable *receiveMessageAllways* y borra el método *doNotReceiveMessageInMeetings*. La directiva *doNotReceiveMessageInMeetings* hace lo inverso.

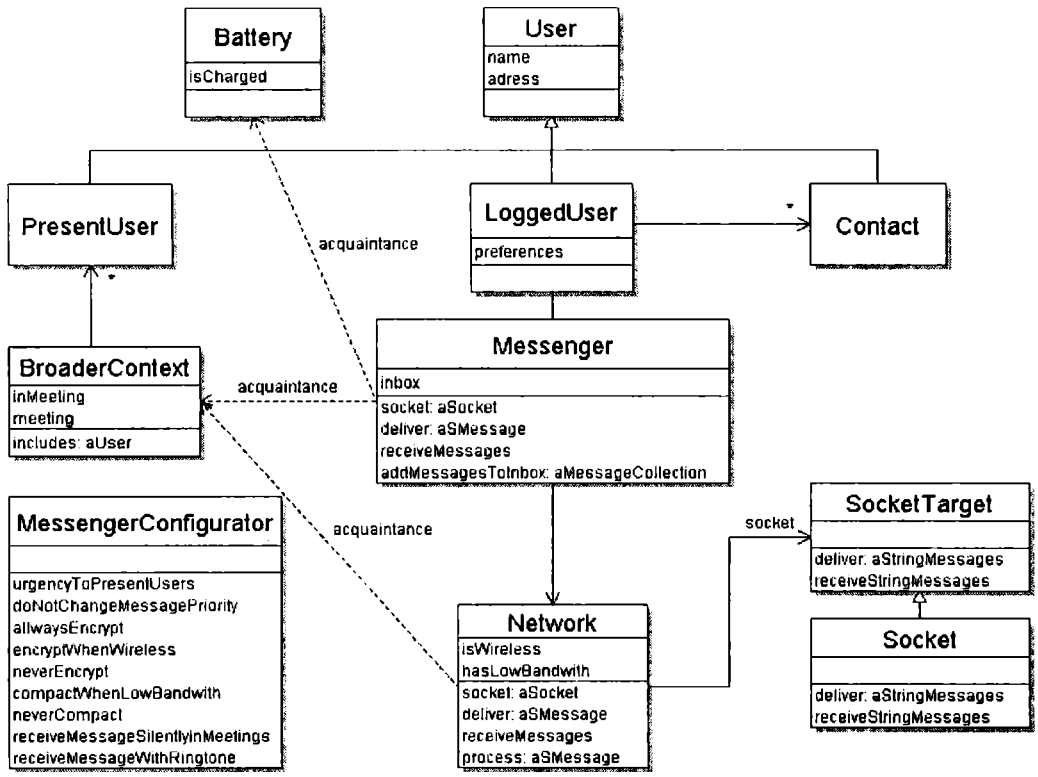

Figura 6.11: Clases, relaciones y mensajes del diseño con comportamiento subjetivo de los requerimientos de adaptación.

6.3. CONCLUSIÓN

En la figura 6.11 vemos el diseño subjetivo completo. Al compararlo con el diseño del capítulo anterior (Ver figura 5.16), notamos que el diseño con comportamiento subjetivo es más *liviano*, en el sentido de que utiliza menos elementos de diseño. Por otro lado, si revisamos el diseño de los requerimientos funcionales que se muestra en la figura 4.5, podemos ver que no hemos introducido grandes cambios. Esto quiere decir que el diseño obtenido es más legible con respecto a las responsabilidades principales de la aplicación. Por otro lado, además de obtener un diseño más compacto que el construido con despacho dinámico simple y sentencias if-then, los requerimientos de adaptación siguen siendo igual de intercambiables y modificables. Notemos además, que se incluyeron los requerimientos de adaptación conservando el tipo de los objetos de los requerimientos funcionales. Esto provoca que la incorporación de los requerimientos de adaptación sea transparente para los clientes de la aplicación.

Capítulo 7

CONLCUSIONES

7.1. COMPARACIÓN DE MECANISMOS DE ADAPTACIÓN

Hasta ahora hemos comparado el comportamiento subjetivo desde el punto de vista del diseño. En este capítulo vamos a comparar las capacidades de los mecanismos de adaptación teniendo en cuenta el grado de adaptación al entorno que ofrecen y sus facilidades de mantenimiento. Luego realizaremos conclusiones generales sobre el trabajo presentado en la tesis y analizaremos los resultados obtenidos. Los tres primeros mecanismos que vamos a analizar son ampliamente utilizados en los lenguajes orientados a objetos. El cuarto mecanismo no es popular, pero describe una manera de ampliar la adaptación utilizando predicados en el despacho de métodos. Por último, se analiza el comportamiento subjetivo, que intenta tomar las mejores características de los mecanismos vistos.

Para empezar el análisis, describiremos algunas características ideales de los mecanismos de adaptación. Sobre dichas características basaremos la comparación entre los mecanismos.

- **Independencia de respuestas**

Es preferible que los diferentes comportamientos del mensaje sean totalmente independientes para que lo modificación de uno no altere la definición de otro.

- **Lógica de decisión externa**

Es conveniente que la lógica que decide la respuesta de un mensaje no esté embebida dentro de las mismas implementaciones. De esta manera, dicha lógica se posiciona en un nivel de diseño distinto al de las implementaciones del mensaje, aumentando la claridad de la aplicación.

- **Lógica de decisión dinámica**

Si la lógica de decisión es externa, y tiene una representación dentro del ambiente de objetos, puede definirse y modificarse en tiempo de ejecución. Esta capacidad hace que una clase de objetos se adapte a características dinámicas de la aplicación.

- **Información que interviene en la decisión de la respuesta**

Los mecanismos de adaptación de comportamiento le permiten al diseñador definir la lógica de decisión de la respuesta de un mensaje. Esta lógica se evalúa en tiempo de ejecución o en tiempo de compilación. Si se evalúa en tiempo de ejecución, cuanto más objetos y propiedades de dichos objetos intervengan en la definición de la respuesta, más capacidad de adaptación al entorno adquiere el comportamiento de un mensaje.

- **Aplicación múltiple de comportamientos**

Si el mecanismo de adaptación permite que sólo el conjunto de sentencias de una misma condición aplique en una invocación, entonces no permite que un mismo pedazo de código se ejecute para varias circunstancias. Si no se toman medidas de refactorización de código, es posible que una misma secuencia de expresiones tenga que repetirse en la lógica de un mismo mensaje.

Mecanismo de Despacho Simple

A continuación vamos a analizar algunos mecanismos de adaptación según las características mencionadas. Comenzaremos con el despacho de mensajes dinámico simple. Si dicho mecanismo esta basado en clases, cuando un mensaje es invocado su método asociado se busca en la clase del receptor. Si no se encuentra, se busca en la superclase del receptor y así sucesivamente. Este mecanismo es externo a la implementación del mensaje y no puede modificarse por el diseñador, ni tampoco dinámicamente, ya que forma parte de la definición del lenguaje subyacente.
Si tenemos un conjunto de objeto polimórficos, las diferentes respuestas a los mismos mensajes se encapsulan en diferentes clases y son independientes una de otras. Por ejemplo, veamos la figura 1, donde tenemos dos clases de objetos polimórficos que implementan los mensajes *#deliver: aSMessage with: aMessenger* y *#receiveMessages with: aMessenger.*

Resumen del Mecanismo de Despacho Simple

- Provee mecanismos para la independencia de implementación.
- La lógica de decisión es externa a la implementación del mensaje.
- No puede modificarse dinámicamente.

- La única información que interviene en la decisión de la respuesta es la clase del receptor.
- Aplica una sola implementación como respuesta de un mensaje

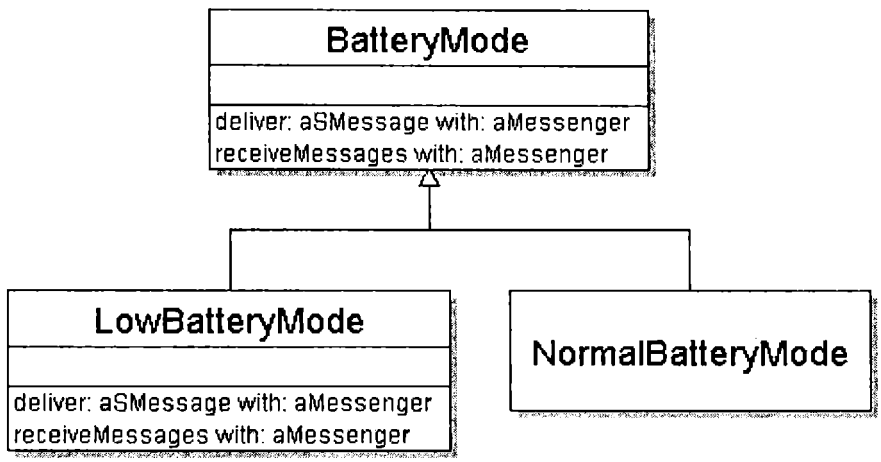

Figura 1: Clases de objetos polimórficos.

Construcciones *if-case*

En los lenguajes Orientados a Objetos con despacho simple suelen existir otros mecanismos de adaptación. El más común es el uso de sentencias if-case. Dichas sentencias evalúan ciertas condiciones del mensaje y resuelven qué pedazo de código ejecutar. Este mecanismo de decisión es definido por el diseñador y está embebido en la implementación del mensaje, junto con la implementación de los diferentes comportamientos. Esto provoca que las respuestas de un mensaje no sean independientes una de otras y que el mecanismo de decisión no tenga una representación propia en el ambiente de objetos. Esto impide que sea modificado dinámicamente a través de técnicas reflexivas. Por otro lado, puede definirse varias estructuras condicionales en un mismo mensaje, incluso anidarse. Esto tiene como ventaja que el mensaje puede ejecutar el mismo pedazo de código en distintos caminos lógicos, facilitando el reuso dentro del mensaje; pero, desafortunadamente, también complica la lógica condicional del mensaje haciéndolo difícil de mantener.

Resumen de Construcciones *if-case*

- No provee mecanismos para la independencia de respuestas.
- La lógica de decisión esta embebida en las implementaciones del mensaje.
- No puede modificarse dinámicamente.
- El mecanismo de decisión se basa en la implementación de condiciones. La implementación de las condiciones se construye consultando cualquier propiedad del receptor de sus colaboradores.
- Pueden aplicar varios comportamientos subordinados a distintas condiciones.

Multimétodos

Otro mecanismo de adaptación utilizado en los lenguajes Orientados a Objetos con despacho simple, es el despacho de mensajes múltiple (i.e. multi-methods), en este mecanismo todos los argumentos, no sólo el receptor del mensaje, son tratados en la elección del método a aplicar. En un lenguaje Orientado a Objetos basado en clases, pueden definirse varias implementaciones de un mismo mensaje dentro de una clase, para diferentes clases de parámetros. En el momento de ejecución del mensaje, la implementación se busca en la clase del objeto receptor y se elije la única que coincide con las clases o las superclases del resto de los parámetros recibidos.

Las diferentes respuestas de un mensaje pueden variar dependiendo de la clase del receptor y de las clases de los colaboradores externos del mensaje. En los lenguajes que no tienen despacho de mensajes múltiple puede desglosarse el comportamiento de un mensaje según el tipo de un argumento, utilizando despacho doble de mensajes (i.e. *double-dispatching*). Con esta técnica, se obtienen diferentes métodos para cada combinación de clase de receptor y clase de argumento. La desventaja de esta técnica es que los comportamientos alternativos del mensaje quedan diseminados en las clases de sus argumentos y no en la clase receptora. Esto puede complicar la lectura del diseño. En contrapartida, multimétodos permite desglosar el comportamiento de un mensaje en diferentes métodos, manteniendo las diferentes implementaciones en la clase receptora.
La lógica que decide el método a aplicar es independiente de sus implementaciones, aunque no es posible modificarla utilizando técnicas reflexivas, ya que este mecanismo de adaptación subyace en el lenguaje utilizado, sin que se provea una representación del mismo.

Resumen de Multimétodos

- Provee mecanismos para la independencia de implementación.
- La lógica de decisión es externa a la implementación del mensaje.
- No puede modificarse dinámicamente.
- La información que interviene en la decisión de la respuesta es la clase del receptor y las clases de los colaboradores externos.
- Aplica una sola implementación como respuesta de un mensaje.

Despacho por Predicado

Un trabajo realizado en ECOOP'98 describe un mecanismo de despacho de métodos basado en predicados denominado *Predicate Dispatching* [ERN/98]. Este mecanismo generaliza los mecanismos de despacho de mensajes, permitiendo que predicados arbitrarios controlen la aplicación de métodos y también usa las implicaciones lógicas cómo relaciones de sobreescritura. Una declaración de un método especifica su aplicabilidad mediante una expresión de predicados. Esta expresión es una fórmula lógica compuesta de un testeo de clase y de una expresión booleana arbitraria. Un método es aplicable cuando su expresión de predicado evalúa verdadera y sólo puede aplicar uno por mensaje,

por lo tanto la respuesta es única. Por otro lado, dado que la expresión lógica booleana es arbitraria la aplicabilidad del método, no sólo puede depender de las clases de los colaboradores externos como es el caso de multimétodos, sino también del estado de los colaboradores tanto externos como internos del receptor. De igual manera que en multimétodos, las diferentes implementaciones del mensaje se separan en métodos distintos y son totalmente independientes una de otras. Así mismo, la lógica de decisión de la respuesta es independiente de las implementaciones pero no se describen técnicas reflexivas para modificarla de manera dinámica. Por lo tanto, una vez definida por el diseñador, no puede ser modificada en tiempo de ejecución.

Resumen de Despacho por Predicado

- Provee mecanismos para la independencia de implementación.
- La lógica de decisión es externa a la implementación del mensaje.
- No puede modificarse dinámicamente.
- La información que interviene en la decisión de la respuesta es cualquier propiedad del receptor y de sus colaboradores.
- Aplica una sola implementación como respuesta de un mensaje.

Comportamiento Subjetivo

Con comportamiento subjetivo podemos definir cada implementación del mensaje en un método distinto. Como multimétodos, cada método subjetivo se ejecuta de manera independiente con respecto a los otros del mismo mensaje, fortaleciendo la independencia de las posibles respuestas. Adicionalmente, el mecanismo de despacho de mensajes propuesto por esta tesis, permite que varios métodos asociados a un mensaje puedan ejecutarse en la misma invocación. Esto permite que en una misma invocación se seleccionen diferentes métodos favoreciendo el reuso.
Por otra parte, la lógica de decisión de los métodos aplicar en un mensaje es definida por el diseñador y tiene una representación dentro del ambiente de objetos, por lo tanto, puede modificarse y definirse en tiempo de ejecución a través de técnicas reflexivas.

Cada método de un mensaje subjetivo tiene una condición asociada, dicha condición se implementa con los objetos *influyentes* que son los responsables de variar el comportamiento del receptor. Este rol puede ser ocupado por cualquier objeto del contexto de recepción del mensaje, estos pueden ser el receptor, sus colaboradores, tanto externos como internos, el emisor del mensaje, o por cualquier otro objeto de la aplicación que deba influenciar la respuesta del mensaje. Por último, las condiciones de un mensaje subjetivo pueden formarse armando predicados sobre los objetos influyentes, consultando cualquiera de sus las características esenciales: su identidad, su estado, y el tipo [GAM/95] que posea.

Resumen de Comportamiento Subjetivo

- Provee mecanismos para la independencia de implementación.
- La lógica de decisión es externa a la implementación del mensaje.
- No puede modificarse dinámicamente.
- La información que interviene en la decisión de la respuesta es cualquier propiedad del receptor, de sus colaboradores, del emisor del mensaje y de cualquier objeto de la aplicación que influencie el comportamiento del mensaje.
- Pueden aplicar varios comportamientos subordinados a distintas condiciones.

Conclusión de la comparación de mecanismos

Los mecanismos de despacho simple y multimétodos ofrecen independencia de implementaciones, pero la lógica de decisión de respuesta que permiten definir se basa en pocos recursos de información. Además no permiten que apliquen varias implementaciones, por lo tanto, si se quiere que un mismo conjunto de sentencias se ejecute en varios caminos de la lógica de decisión, deberán realizarse esfuerzos de diseño para refactorizar el código. Los lenguajes que utilizan los mecanismos despacho simple y multimétodos generalmente se complementan con las construcciones if-case. Dichas construcciones permiten utilizar más información para construir la lógica de decisión y ayudan a reusar sentencias de código que se repiten para varios caminos de la lógica del mensaje. La desventaja es que no proveen mecanismos para la independencia de implementaciones y que la lógica de decisión queda entrelazada entre las implementaciones del mensaje. Esto dificulta la claridad del mensaje y no permite que la lógica de decisión evolucione de manera separada. El despacho por predicado, intenta recuperar las mejores características de estos dos mecanismos, ya que provee recursos para la independencia de implementación, y ofrece una lógica de decisión externa basada en condiciones. Igualmente, dicho mecanismo también restringe la información a utilizar por la lógica de decisión. En contrapartida, el comportamiento subjetivo permite definir la lógica de decisión con características de cualquier objeto del contexto de recepción del mensaje, esto incluye al receptor, sus colaboradores, el emisor del mensaje y cualquier objeto que deba influenciar la respuesta. Además provee mecanismos para la independencia de implementaciones y permite que la lógica de decisión sea independiente y que pueda modificarse dinámicamente. Así mismo, varias implementaciones pueden aplicar en una misma invocación, ayudando al reuso de código.

7.2. RESULTADOS OBTENIDOS

El principal aporte de esta tesis es analizar las consecuencias en el diseño orientado a objetos de la utilización comportamiento subjetivo. Hemos demostrado mediante un ejemplo que el uso del comportamiento subjetivo en el diseño construye diseños más compactos e igual de mantenibles que los realizados con despacho de métodos dinámico simple.

Primero hemos observado que en algunas aplicaciones existen requerimientos funcionales que tienen adosados requerimientos de adaptación. Dichos requerimientos de adaptación hacen que una función cambie su comportamiento según ciertas condiciones de la aplicación. Además notamos que los requerimientos de adaptación son más inestables, y es preciso que la modificación de comportamiento que generan esté encapsulada, para hacer más fácil la modificación y el intercambio de las distintas implementaciones. Esta necesidad genera esfuerzos de diseño, que producen nuevos elementos de modelado que se entremezclan con los elementos de diseño originados por los requerimientos funcionales, complicando el diseño de la aplicación y por consiguiente su mantenimiento.

En contrapartida, el comportamiento subjetivo da soporte al diseño de requerimientos de adaptación sin generar nuevos elementos de diseño. Para demostrarlo se fabricó una especificación basada en el dominio de aplicaciones *context-aware*, dado que las aplicaciones *context-aware* se caracterizan por tener requerimientos de adaptación. La especificación contenía pocos requerimientos funcionales y muchos requerimientos de adaptación adosados. Primero se modelo la aplicación teniendo en cuanta sólo los requerimientos funcionales. Luego se incremento el diseño adosando los requerimientos de adaptación basándonos únicamente mecanismo de despacho de mensajes simple y sentencias if-case. Por último, se desarrollo otro diseño incrementado nuevamente el diseño de los requerimientos funcionales con los mismos requerimientos de adaptación, pero esta vez utilizando comportamiento subjetivo.

Como resultado, se obtuvieron dos diseños alternativos de los requerimientos de adaptación. En ambos, se independizaron las diferentes implementaciones del mensaje, de manera de facilitar la futura modificación de los requerimientos de adaptación, pero con comportamiento subjetivo esto no ocasionó incorporar nuevos elementos de diseño. En consecuencia, el diseño con comportamiento subjetivo no tiene grandes modificaciones según el diseño de los requerimientos funcionales. Esto hace que el diseño de la aplicación sea más legible con respecto a sus responsabilidades principales.

Otro aporte de esta tesis, es mejorar el mecanismo de respuesta. El árbol de decisiones expuesto en los trabajos previos de comportamiento subjetivo separa las condiciones de un mensaje de sus implementaciones, pero promueve el uso de construcciones condicionales anidadas, que dificultan la lectura del mensaje. El mecanismo expuesto por este trabajo, reemplaza el árbol de decisión, permitiendo organizar las implementaciones de manera secuencial facilitando la lectura de la lógica condicional del mensaje y por lo tanto su mantenimiento. Además, la lógica de decisión modelada con métodos agregables por condición, permite que varios métodos apliquen su comportamiento en una misma invocación. Esto permite que un mismo método aplique, en distintas circunstancias de recepción del mensaje permitiendo el reuso de código dentro del mensaje de manera natural.

Uno de los mecanismos de adaptación de comportamiento más flexible y ampliamente usado, son las construcciones if-case. Una de las desventajas de este mecanismo es que las condiciones quedan entrelazadas con implementación del

mensaje. Esto provoca que las diferentes respuestas de un mensaje no sean independientes, y que las condiciones estén definidas al nivel de implementación. Además, la lógica de decisión del mensaje no tiene representación separada y se torna difícil modificarla dinámicamente.

Los mecanismos de despacho simple y multimétodos ofrecen independencia de implementaciones, pero la lógica de decisión de respuesta que permiten definir se basa en pocos recursos de información. Además no permiten que apliquen varias implementaciones, por lo que deben realizarse esfuerzos de diseño para proveer reuso de código, si se quiere que un mismo conjunto de sentencias se ejecute en varios caminos de la lógica de decisión del mensaje. Los lenguajes que utilizan los mecanismos despacho simple y multimétodos generalmente se complementan con las construcciones if-case.

En contrapartida, el comportamiento subjetivo permite definir la lógica de decisión con características de cualquier objeto del contexto de recepción del mensaje, esto incluye al receptor, sus colaboradores, el emisor del mensaje y cualquier objeto que deba influenciar la respuesta. Además provee mecanismos para la independencia de implementaciones y permite que la lógica de decisión sea independiente y que pueda modificarse dinámicamente. Por otro lado, varias implementaciones pueden aplicar en la misma invocación, ayudando al reuso de código dentro del mensaje.

7.3. VENTAJAS Y DESVENTAJAS DEL COMPORTAMIENTO SUBJETIVO

La desventaja del comportamiento subjetivo es que la ejecución de los mensajes subjetivos es más lenta que ejecución de los mensajes con despacho dinámico simple. Esto se debe a que deben calcularse los dos contextos de ejecución, el de las condiciones y el de las implementaciones de los mensajes, y que además deben elegirse que métodos aplicar. El prototipo construido no tiene como fin bajar estos tiempos de ejecución y no se ha explorado de que manera hacer la ejecución de mensajes subjetivos más eficiente. De todas maneras se sabe que hay una ineficiencia inherente, debido a la evaluación de la lógica condicional que deben resolver estos mensajes. Sin embargo, esta ineficiencia puede justificarse. Como vimos en la sección anterior el diseño con comportamiento subjetivo es más compacto que el diseño con despacho de mensajes simple. Esta ventaja del comportamiento subjetivo hace que se necesiten menos clases de objetos y menos relaciones para completar el diseño. En tiempo de ejecución esto se traduce a menos invocaciones entre objetos de la aplicación. En este sentido la ineficiencia de la ejecución puede ser compensada.

Capítulo 8

REFERENCIAS

[BAN/02] Guruduth Banavar and Abraham Bernstein. "Software Infrastructure and Design Challenges for Ubiquitous Computing Applications". Commun. ACM 45, 12 (Dec. 2002), 92-96.

[BOB/86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In OOPSLA '86 Conference Proceedings, pp. 17-29, Portland, OR, September, 1986. Published as SIGPLAN Notices 21(11), November, 1986.

[BOB/88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In SIGPLAN Notices 23(Special Issue), September, 1988.

[BOO/98] G.Booch, I. Jacobson, J.Rumbaugh. "Unified Modeling Language – Reference Manual" Addison-Wesley 1998/1999

[CAL/99] C. Callegari, L. Eliaschev y C. Tanner, realizan su tesis de Licenciatura: "Subjetividad en un Ambiente de Objetos" 1999.

[CHE/99] CHEVERST, K., MITCHELL, K., AND DAVIES, N. Design of an object model for a context sensitive tourist GUIDE. Computers and Graphics 23, 6 (1999),

[DAR/02] B. Darderes, A. Grassano, M. Prieto. Subjective behavior of Objects. Position Paper in OOPSLA'02 workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE), Seattle WA, USA November 2002.

[DAR/04] B. Darderes, M. Prieto. Subjective Behavior: A General Dynamic Method Dispatch. Position Paper in OOPSLA'04 workshop on Revival On Dynamic Languages, Vancouver, British Columbia, Canada.

[DAR/05] B. Darderes, M. Prieto. "Comportamiento Subjetivo: Objetos Adaptables a Cambios de Contexto" Paper presentado en la JAIIO 2005. Rosario. Argentina

[DEY/00] A. Dey thesis: "Providing Architectural Support for Building Context-Aware Applications".

[ERN/98] M. Ernst , C. Kaplan , C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch, Proceedings of the 12th European Conference on Object-Oriented Programming, p.186-211, July 20-24, 1998

[FIN/02] Anthony C.W. Finkelstein, Andrea SavigniGerti Kappel, Werner Retschitzegger Ubiquitous Web Application Development -A Framework for Understanding

[FIT/02] Adrian Fitzpatrick. "Towards a Sentient Object Model". Position Paper in OOPSLA'02 workshop on Engineering Context-Aware Object Oriented Systems and Environments (ECOOSE), Seattle WA, USA November 2002.

[FOO/89] B. Foote and RE Johnson. Reflective facilities in Smalltalk-80. Proceedings of OOSPLA'89, New Orleans, LA, pp 327-335, October 1989

[GAM/95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *"Design Patterns - Elements of Reusable Object-Oriented Software"* Addison Wesley 1995

[GOL/83] A. Goldberg, D. Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA, 1983.

[GRA/03] A. Grassano, M. Prieto. Subjectivity in an Object Environment with Applications. Tesis de Licenciatura, final report, Facultad de Informática, Universidad Nacional de La Plata, November 2003.

[HAR/93] W. Harrison and H. Ossher. *Subject-oriented programming: a critique of pure objects.* In Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pages 411--428. ACM Press, 1993.

[HAR/94] W. Harrison, H. Ossher, R. B. Smith, and U. David, Subjectivity in Object-Oriented Systems, Workshop Summary , in OOPLSA'94. 1994, ACM.

[HAR/95] W. Harrison, H. Ossher, and H. Mili, "Subjectivity in object-oriented systems workshop summary," in Addendum to OOPSLA, 1995.

[HEN/02] HENRICKSEN, K., INDULSKA, J., AND RAKOTONIRAINY, A. Modeling context information in pervasive computing systems. In *LNCS 2414:Proceedings of 1st International Conference on Pervasive Computing* (Zurich, Switzerland, 2002), F. attern and M. Naghshineh, Eds., Lecture Notes in Computer Science (LNCS), Springer, pp. 167–180.

[HEN/04] HENRICKSEN, K., AND INDULSKA, J. Modelling and Using Imperfect Context Information. In WorkshopProceedings of the 2nd IEEE Conference on PervasiveComputing and Communications (PerCom2004)(Orlando, FL, USA, March 2004), pp. 33–37.

[KOR/00] Korkea-aho M., "*Context-Aware Applications Survey*". Internetworking Seminar, spring 2000.

[KRI/01] B. B. Kristensen. Subjective Behavior. International Journal of Computer Systems Science and Engineering, Volume 16, Number 1, (13-24), January, 2001.

[TAI/93] A. Taivalsaari. Object-Oriented Programming with Modes. Journal of Object Oriented Programing (JOOP). Junio 1993.

[PRI/95] M. Prieto, P. Victory. Real World Object Behavior. Workshop on Subjectivity in Object-Oriented Systems, The Object-Oriented Programming Systems, Languages and Applications Conference, 1995

[PRI/96] M. Prieto, P. Victory . Subjectivity: Towards True Polymorphism. Workshop on Subjects and Viewpoints throughout the Life Cycle, OOPSLA '96 (Object-Oriented Programming Systems Languages and Applications), October 1996, San Jose CA, USA

[PRI/97] M. Prieto, P. Victory . Subjective Object Behavior. Object Expert, a SIGS publication, March/April 1997, Vol. 2(3), pages 24-26.

[WAN/04] WANG, X. H., ZHANG, D. Q., GU, T., AND PUNG, H. K. Ontology Based Context Modeling and Reasoning using OWL. In *Workshop Proceedings of the 2nd IEEE Conference on Pervasive Computingand Communications (PerCom2004)* (Orlando, FL, USA, March 2004), pp. 18–22.