Un enfoque orientado a objetos para software context-aware

Andrés Fortier

Trabajo final para obtener el grado de

Licenciado en Informática

de la

Facultad de Informática,
Universidad Nacional de La Plata,
Argentina

Director: Dr. Gustavo Rossi
Co-director: Dra. Silvia Gordillo

La Plata, Octubre de 2005

Indice General

Indice de Figuras

Agradecimientos

On which we bid farewell to
absent friends, lost loves,
old gods and the season of mists.

Season of Mists - The Sandman
Neil Gaiman

Al tiempo que escribo estas palabras se cierra una de las etapas mas importantes de mi vida. Al igual que otras, estuvo llena de satisfacciones y decepciones, así como de un conjunto de personas que me acompañaron y a quienes hoy deseo agradecer.

En primer lugar quiero expresar el agradecimiento hacia mi familia, que se encuentra por sobre cualquier cuestión académica, aunque muchas veces he fallado en demostrárselo. A mis padres, por todo lo que me brindaron; por el esfuerzo que hicieron para educarme, por las infinitas preocupaciones que les causé y por el apoyo que me dieron durante mi carrera. A mi hermana, por el vínculo fraterno que gestó y porque, sin saberlo, muchas veces fue mi modelo a seguir.

Al comenzar la carrera tuve la suerte de encontrarme rodeado de excelentes personas a las que no puedo dejar de mencionar. A Natty, por los años que compartimos. A mis amigos de Quilmes, por todos los buenos momentos que pasamos juntos.

A lo largo de la carrera descubrí a muchas personas valiosas y de las cuales aprendí mucho, mas allá del ámbito académico. Particularmente quiero expresar mi gratitud a Juan, una de esas personas a las que cualquier agradecimiento le queda chico, por haberme alojado incontables veces en su casa, por su amistad incondicional, por las largas conversaciones y por su aguante infinito. Asimismo quiero agradecer a Fidel, por haberme despertado al mundo de la docencia y por haber puesto el esfuerzo de armar una cátedra como la de Programación Funcional en el año 2000. Siempre recordaré a las personas que participaron de esa experiencia, con la convicción que algo mejor es posible y que los castillos de arena pueden soportar los embates del mar.

En lo que respecta a mi breve carrera de investigación, quisiera agradecer al LIFIA, que me tomó desde los comienzos de mi carrera y me permitió crecer. A Gustavo, por dirigirme durante estos años y a Silvia por haber aceptado co-dirigir este trabajo. A Juan y Gaby por haberse tomado el trabajo de leer este informe y aportar con sus correcciones y sugerencias.

Por último, me quiero reservar un agradecimiento especial a Sabrina, por la ayuda y el apoyo que me brindó cuando decidí independizarme y por permitirme ser parte de su vida.

Capítulo 1

Introducción

This bequest I leave you she says
You will see what could be evergreen
Turn to copper and fade to gray

Summer's End
Amorphis

El avance en el desarrollo de hardware permite hoy contar con pequeños dispositivos de computación móviles (como por ejemplo PDAs), capaces de manejar aplicaciones que hace menos de 10 años sólo podían ejecutarse en computadoras de escritorio. Esta evolución permite imaginarnos un futuro en el cual el usuario es asistido por diversos tipos de computadoras para llevar a cabo sus tareas diarias. Para poder llegar a ésto, es necesario un replanteo general de las aplicaciones a desarrollar, tanto al nivel de interacción hombre-máquina como al nivel de funcionalidad. En este tipo de aplicaciones ya no se puede asumir un contexto fijo, como puede ser el lugar de trabajo o el hogar, sino que las situaciones por las que va pasando el usuario (y por ende su computadora) puede variar drásticamente a lo largo del día. Como respuesta a este replanteo se está gestando una nueva generación de aplicaciones, cuyo objetivo es adaptarse al contexto en el que se encuentran, para así poder brindar al usuario servicios en base a su situación.

1.1 Introducción y motivación

Durante los últimos 15 años, la tecnología en el desarrollo de hardware ha creado computadoras cada vez mas pequeñas, baratas y rápidas; de hecho, tanto se ha avanzado en el área, que el poder de cómputo de una máquina de escritorio construída hace 10 años, hoy se encuentra disponible en una PDA[1]. Gracias a estos

[1]La HP iPAQ hx2750, un modelo hoy estándar, está equipara con un procesador de 624MHz, 128MB de memoria ROM, 128MB de memoria RAM, placa de red inalámbrica incorporada y una pantalla con resolución de 640x480 píxeles.

avances, se ha desarrollado un mercado de manufactura y venta de dispositivos portátiles que crece año a año. Por cuestiones económicas, en nuestro país este fenómeno no ha tomado fuerza aún, pero basta analizar los estudios realizados en países económicamente desarrollados[2] para ver cómo estos dispositivos pasan a ser parte del día a día de las personas. A pesar de no poder apreciarlo en nuestro país en una escala tan importante, de alguna forma la noción de computación embebida ya nos toca directamente: no es extraño encontrar microprocesadores en los equipos de audio, automóviles o microondas.

A partir del ingreso de estos dispositivos en nuestra vida diaria, resulta interesante mirar hacia atrás y analizar los modelos de interacción hombre-máquina de las cuatro últimas décadas. En un principio, una computadora central (*mainframe*) debía ser compartida por una gran cantidad de personas, quienes competían por su uso y raras veces podían verla ejecutando un programa. Luego llegó la revolución de las PCs de escritorio, lo cual trajo la posibilidad de dedicar una computadora a una persona, permitiendo una relación más estrecha entre ambos y proporcionando un conjunto de facilidades para adaptar la computadora a las preferencias del usuario. El siguiente paso (que en este momento se está gestando) es un mundo en el cual una persona tiene a su disposición un gran conjunto de dispositivos de cómputo, presentados en diversos formatos y con distintos propósitos, pudiendo ser tanto fijos (como por ejemplo, llaves de luz comandadas digitalmente) como móviles (PDAs y *smartphones*). Este nuevo paradigma de interacción, no ocurre en un único lugar ni en un contexto de trabajo fijo (como es el caso del trabajo con aplicaciones de escritorio), sino que abarca una gran cantidad de situaciones y lugares, como puede ser una clase teórica en el aula de una universidad, un colectivo lleno de gente o una reunión de amigos en una plaza. Asimismo, los recursos disponibles y los requerimientos de una aplicación móvil varían dinámicamente: cuando un usuario se mueve al aire libre, se suele utilizar una conexión inalámbrica brindada por un proveedor de telefonía celular en conjunto con un mecanismo de posicionamiento GPS. Dicho mecanismo puede ser utilizado para ubicar al usuario en un mapa, con un error aceptable de ± 2 metros. Al ingresar a un espacio cerrado, los mecanismos de comunicación y sensado suelen variar: la conectividad estará dada por una red inalámbrica basada en *access points* y la ubicación del usuario se determinará por medio de un sistema basado en sensores infrarrojos, con una precisión mucho mayor al GPS. Adicionalmente, al ingresar en un laboratorio del edificio, el usuario puede tener acceso a una impresora bluetooth o a un listado de personas que se encuentran en ese ambiente en ese momento. Por último, la PDA que acompaña al usuario debe adecuarse a distintas "configuraciones" sociales, como pueden ser una reunión de trabajo o una salida al cine.

[2]Según el artículo publicado por Barkhuus y Dourish [2], en un campus universitario el 94% de los estudiantes posee teléfono celular, el 55% tiene reproductor de MP3 portátil, el 65% posee una laptop y el 42% una PDA.

De la misma forma que este nuevo modelo presenta una cantidad infinita de oportunidades para el desarrollo de nuevas aplicaciones, también trae un conjunto de interrogantes respecto de cómo se va a llevar a cabo la interacción entre un usuario y ese conjunto de dispositivos, siendo probablemente uno de los fines más importantes lograr una integración transparente con los dispositivos de cómputo durante nuestra vida cotidiana. Es de esperarse que este cambio en la utilización de computadoras impacte en el tipo de aplicaciones que se desarrollan y por lo tanto en el modelo de construcción de éstas. Una de las áreas que han surgido como consecuencia de este nuevo modelo, es la de aplicaciones sensibles al contexto (*context-aware*). A grandes rasgos, este tipo de aplicaciones tiene como objetivo presentar información o modificar su comportamiento en base a la situación en que se encuentre el usuario, la cual puede estar dictada por diversas configuraciones de hardware o software, el lugar en que se encuentra, las personas que lo rodean, la actividad que se está llevando a cabo, etc.

La motivación del presente trabajo de grado viene dada por la necesidad de contar con una arquitectura flexible que soporte el desarrollo de aplicaciones context-aware. A pesar que al momento se han desarrollado una gran cantidad de aplicaciones y se han planteado diversas arquitecturas y frameworks, todavía no se ha encontrado una que pueda manejar uniformemente todos los aspectos de contexto ni se ha hallado una metodología de desarrollo estándar. Por ser ésta una disciplina reciente, quedan todavía muchos interrogantes y campos de investigación por explorar; se espera que este trabajo pueda brindar una visión arquitectural alternativa y novedosa para desarrollar aplicaciones context-aware.

1.2 Objetivos del trabajo

Uno de los problemas asociados a las aplicaciones context-aware, es que no hay una definición universalmente aceptada que indique qué es el contexto de una aplicación. Cómo veremos más en detalle en el siguiente capítulo, hasta el momento se han planteado dos visiones de contexto:

- La positivista, que lo ve como un problema de representación. Su principal objetivo es encontrar una representación formal del contexto y codificarlo en una máquina.
- La fenomenológica, que lo ve como un problema de interacción. Bajo esta visión el contexto es un emergente de la actividad que no puede ser formalizado.

En este momento la tendencia es tomar al contexto como un conjunto de información pasiva (como un problema de representación), sobre la cual el sistema tiene que razonar y tomar decisiones. A pesar que no es el objetivo de este trabajo entrar en la discusión de qué es el contexto ni cómo debe definirse, sí se

planteará un modelo de contexto desde un punto de vista distinto al utilizado actualmente. En lugar de pensar al contexto como una entidad completamente definida, la idea es descomponerlo en un conjunto de aspectos [3] de contexto (modelados con objetos) y sus interacciones. Esto significa que el contexto deja de ser modelado como una entidad única y formal y pasa a ser un emergente de la interacción entre los objetos, similar la visión fenomenológica. Este modelo surge del trabajo centrado en ambientes puros de objetos y de una visión de contexto basada en la interacción hombre-máquina [19]. Concretamente, el primer objetivo de esta tesis es comprobar que esta forma de ver al contexto resulta adecuada para lograr diversos tipos de adaptaciones, al tiempo que se mantiene un modelo de objetos correcto.

El segundo objetivo del trabajo es proponer una arquitectura que responda a la visión de contexto anteriormente mencionada. Esta arquitectura debe ser independiente de la plataforma, así como de los dispositivos sobre los que se esté ejecutando (PDA, PCs de escritorio, tablet PCs, etc) y debe poder adaptarse al trabajo sobre redes heterogéneas. Dado que no es posible adelantarse a todos los posibles contextos ni a la forma en que la aplicación debe reaccionar, este trabajo hará especial hincapié en la escalabilidad y la adaptabilidad a los cambios.

Para validar las suposiciones respecto del modelado del contexto y demostrar que la arquitectura se adapta satisfactoriamente a diversos requisitos, se construyó un framework para evaluar una serie de servicios basados en diversos aspectos contextuales. Dicho framework muestra, con una granularidad más fina, los componentes constitutivos de la arquitectura planteada, permitiendo apreciar una serie de objetos, roles y estereotipos (en el sentido UML) que deben estar presentes en toda aplicación context-aware.

1.3 Estructura del trabajo

En el capítulo 2 se darán los fundamentos necesarios para comenzar a trabajar con aplicaciones context-aware: se verán algunos trabajos precursores en el área (como la computación móvil), se estudiarán las definiciones actuales de contexto y se analizarán las dos principales visiones de contexto. Por último se verán tres modelos utilizados para representar el contexto en una máquina.

En el capítulo 3 se sintetizará el estado del arte, incluyendo una descripción de las aplicaciones mas relevantes, así como un análisis de frameworks y arquitecturas que se están utilizando actualmente.

En el capítulo 4 se presentará un modelo de contexto alternativo al utilizado en la mayoría de los desarrollos, el cual se basa en el comportamiento de objetos contextuales y no en los datos de contexto. Para lograr ésto, el modelo de con-

[3] Es importante aclarar que el modelo basado en aspectos de contexto no tiene nada que ver con la programación orientada a aspectos (AOP).

texto se basa en la interacción de un conjunto de *aspectos de contexto*, que son los responsables de disparar los cambios de contexto. Una vez planteado el modelo de contexto se mostrará el diseño de una arquitectura para desarrollar aplicaciones context-aware, estructurándolas en una serie de capas con responsabilidades bien definidas. Asimismo se presentará un framework para la construcción de aplicaciones context-aware, el cual se adecua a la arquitectura propuesta. Con el objetivo de trabajar en un caso concreto, el capítulo tendrá como foco el desarrollo de aplicaciones location-aware.

En el capítulo 5 se describirán las mejoras necesarias al framework para poder construir aplicaciones que se adapten a distintos aspectos de contexto y no sólo a la ubicación del usuario. En particular se mostrará como un manejo basado en eventos ayuda a llegar a un modelo genérico de aspecto de contexto. Al final del capítulo se contrastará el trabajo realizado con otras implementaciones.

Por último, en el capítulo 6 se presentarán las conclusiones el trabajo a futuro.

Capítulo 2

Fundamentos de context-awareness

A stupid man's report of what a clever man says
can never be accurate,
because he unconciously translates what he hears
into something he can understand.

The History of Western Philosophy
Bertrand Russell

En los últimos años ha crecido el interés en lograr que las aplicaciones adapten su comportamiento a diversos factores externos, asistiendo al usuario en forma particular dependiendo de la situación en la que se encuentre. Gran parte de este fenómeno se debe al avance en los dispositivos de computación portátiles, que son cada vez más pequeños y con mayor poder de procesamiento. Este avance plantea importantes cambios, así como nuevos retos para desarrollar software.
En el presente capítulo se brindan las bases necesarias para comenzar a trabajar con aplicaciones context-aware; para ésto se comentarán algunos trabajos previos a la computación context-aware, que como veremos más adelante, resultan ser el punto de partida para esta disciplina. Asimismo se plantearán diversas definiciones de contexto y context-awareness, así como las dificultades que aparecen al trabajar con el contexto. Por último se verán tres enfoques básicos para modelar al contexto.

2.1 Un cambio de paradigma

En un futuro cercano, el consumidor estándar de productos informáticos no estará sentado frente a su máquina de escritorio en su hogar, sino que deberá manejar un conjunto de pequeñas computadoras diseminadas geográficamente. Estos dispositivos deberán adaptarse a las actividades del usuario, integrándose de forma tal que el usuario no tenga que realizar un esfuerzo intelectual para

utilizarlos. De hecho, se espera que con el paso del tiempo su uso sea tan natural como puede ser hoy el habla o la escritura.

Este nuevo paradigma de trabajo es lo que Weiser plantea en sus trabajos [55, 56], imaginando cómo debería ser la próxima revolución en la era de la computación (lo que él denominó *the age of calm technology*). En este momento nos encontramos dando los primeros pasos para llegar a ese futuro, pero es claro que todavía existen muchas piezas que no encajan y que debemos ir descubriendo y moldeando. Una de estas piezas (tal vez una de las más importantes) es comprender cómo y por qué los humanos tenemos la facilidad de adaptarnos al contexto y las máquinas no. Una vez que hayamos comprendido ésto, podemos pensar en formas de aumentar a las computadoras con la capacidad de ser sensibles al contexto, para luego indicarles cómo actuar en diversas situaciones.

En su tesis doctoral, Dey [18] indica que la diferencia entre humanos y máquinas puede ponerse en términos de tres aspectos: captura, procesamiento y salida de información. Claramente las computadoras no pueden capturar la cantidad de información ni interpretarla en la forma que lo hacemos los humanos. A pesar de tener un poder de cómputo que es en algunos aspectos (por ejemplo, matemático) mayor que el de los humanos, todavía hay cosas en las que las máquinas no puden igualar a los humanos. Esto se debe principalmente a que una máquina sólo puede realizar lo que el programador le indique, lo cual limita automáticamente la cantidad de cosas que puede realizar; cosas como comprender el lenguaje natural o interpretar la situación en la que se encuentra una persona no son tareas sencillas para una máquina. Es interesante notar que las computadoras pueden manejar mucho mejor la salida de información que la captura; mientras que para mostrar información una máquina puede usar símbolos, imágenes y sonido, la captura de información viene dada por lo que el usuario ingresa **explícitamente**, generalmente en forma de texto y en un lenguaje que no es el que utiliza para comunicarse con sus pares. De ésto se desprende que, para lograr una interacción más natural y permitir que una máquina comprenda el contexto en que se encuentra, es necesario enriquecer los mecanismos de captura de información así como la forma de interpretarla y manipularla.

Pensar en mejorar la forma en que una máquina puede manipular y hasta "razonar" es un área de investigación con historia propia, siendo probablemente los campos de inteligencia artificial y redes neuronales los más conocidos. Alternativamente, el esfuerzo en mejorar la interacción entre el humano y las computadoras ha sido atacado desde dos ópticas distintas:

- Mejorar el lenguaje que utilizan los humanos para comunicarse con la máquina.
- Aumentar la cantidad de información contextual disponible para una computadora.

En el primer caso, lo que se intenta es lograr que el humano se comunique con la máquina de una forma que le resulte natural. Para esto se pueden utilizar métodos como reconocimiento de gestos o análisis de voz. A pesar de la ventaja que plantea el ingreso de información en forma natural, todavía nos encontramos con el problema que es necesario hacerlo en forma explícita. Alternativamente, la segunda visión apunta a brindarle a la máquina información contextual que no deba ser ingresada explícitamente por el usuario. Ejemplos de información que podrían ingresarse implícitamente son las personas con las que se encuentra el usuario en un determinado momento, donde ha estado a lo largo del día o qué actividades ha llevado a cabo y con quién.

Es importante notar que estas dos técnicas no son excluyentes, sino que son complementarias; mientras que la primera apunta a facilitar el ingreso de información a la máquina, la segunda hace hincapié en aprovechar información de contexto que generalmente es desperdiciada, la cual puede a su vez resultar crucial para comprender la información explícita que el usuario provee. A lo largo de este trabajo nos centraremos en la segunda técnica, con el objetivo de incorporar información contextual en las aplicaciones e indicar cómo deben reaccionar ante los cambios de contexto. Al realizar ésto, no debemos perder de vista que el objetivo está centrado en facilitar la comunicación y aumentar la utilidad de los dispositivos que nos rodean, por lo que claramente no podemos esperar que el usuario esté constantemente describiéndole a una computadora cuál es la situación en la que se encuentra. Para hacer un uso efectivo del contexto, la computadora debe contar con mecanismos para adquirirlo y manipularlo sin la intervención explícita del usuario, por lo que es necesario brindar una arquitectura que permita crear aplicaciones que puedan manejar el contexto y tomar decisiones por el usuario.

Esta necesidad de manejo de contexto resulta más importante cuando pensamos en aplicaciones móviles, donde al usuario se le plantea la posibilidad de acceder a los servicios que requiera, en el momento que lo desee sin importar donde esté. Si nos imaginamos un escenario en el cual tenemos constantemente dispositivos de computación a nuestro alcance, es de esperarse que dos usuarios quieran acceder a un mismo servicio en distintas situaciones; para estos usuarios, el servicio debería presentarse acorde a su contexto. Un ejemplo muy sencillo está dado por los portales que poseen un diseño web para máquinas de escritorio y otro para PDAs; si tomamos que el dispositivo es parte del contexto en que se está ejecutando la aplicación, y en base a ésto el servidor http decide qué página web enviar, tendremos una forma muy elemental de aplicación context-aware.

En general, cualquier aplicación que tenga en cuenta su contexto, ya sea que se ejecute en una PC de escritorio o en un dispositivo móvil, se denomina *context-aware*. La disponibilidad de diversos mecanismos de *sensado*[1], así como el avance en términos de velocidad de computación, memoria y redes inalámbricas hace posible distribuir el contexto para que pueda ser aprovechado por diversos dispositivos. La combinación de estos aspectos con los dispositivos móviles, hace que el usuario pueda utilizar información que se encuentra en lugares geográficamente distantes en una gran variedad de situaciones, las cuales tienden a estar en constante cambio.

2.2 Trabajos previos

El diseño de una arquitectura que permita aprovechar la información de contexto, está influenciado por los desarrollos en otras áreas de investigación. Una de las principales influencias está dada por la computación móvil (*mobile computing*), ya que en la mayoría de los casos tendremos que manejar la comunicación (probablemente intermitente) entre dispositivos, así como la heterogeneidad de hardware. En segundo lugar debemos "aumentar" la experiencia del usuario por medio de la utilización de diversos dispositivos, en forma similar a lo que plantea la computación ubicua (*ubiquitous computing*). Por último, ya que uno de los aspectos de contexto que resultan más relevantes es la ubicación del usuario, aquellas cosas relacionadas con la computación basada en la locación (*location-based computing*) nos serán de suma utilidad.

2.2.1 Computación móvil

El tipo de conectividad, las características puntuales de una red y la configuración de los periféricos, son aspectos que cambian mucho mas frecuentemente en un ambiente móvil que en un sistema de escritorio. Idealmente, el desarrollador desearía que estos aspectos resultaran transparentes para sus aplicaciones, no teniendo que preocuparse si éstas van a ejecutarse en una PC de escritorio o en un dispositivo móvil; el foco de atención del desarrollador debería centrarse en el modelo de aplicación. Se han planteado diversos enfoques a los problemas de computación móvil para un dominio en particular, como por ejemplo la posibilidad de manejar la desconexión de la red de un sistema de archivos distribuido por medio de archivos cacheados [20]. Otras investigaciones han llevado a plantear redes virtuales [35], las cuales se estructuran como una base de datos distribuida, que permite relacionar a los nodos virtuales con los nodos

[1] Lamentablemente el lenguaje castellano no posee una traducción para la palabra inglesa *sensing*, que indica la adquisición de datos por medio de un sensor. Por este motivo se ha adoptado en la comunidad la palabra *sensado* como una castellanización de sensing.

físicos. Asimismo, este sistema aprovecha la noción de localidad para hacer un seguimiento eficiente de los dispositivos móviles.

A pesar que los trabajos realizados en dominios particulares pueden ser de mucha ayuda y brindar experiencias valiosas, no alcanzan para ser generalizados y llevarlos al nivel de un framework. Lo que se requiere es un mecanismo de distribución de propósitos generales, que resulte transparente para el desarrollo de aplicaciones y que brinde la posibilidad de configurarlo y ajustarlo para alcanzar los niveles de performance necesarios en cada aplicación. Idealmente, esta adaptación debería realizase en la etapa final del desarrollo de la aplicación y no durante el modelado.

2.2.2 Computación ubicua

La idea de la computación ubicua es enriquecer la vida en el "mundo real" por medio de la integración de dispositivos de computación a nuestra vida diaria. Como Weiser indica en uno de sus trabajos pioneros en el área [55], las tecnologías más profundas y aceptadas, son aquellas que logran desaparecer; se mimetizan con nuestro día a día hasta el punto que no notamos su presencia. Esta reflexión es por demás interesante, ya que si la analizamos, encontraremos un patrón que se repite con los grandes avances tecnológicos. Hoy por hoy no resulta extraño ver a una persona hablando por un teléfono celular mientras camina, aunque eso hubiera resultado impensable en 1870, cuando recién se ideaban los primeros prototipos de teléfonos.

Un punto importante a tener en cuenta es que la aceptación de este tipo de tecnologías está ligado a la psicología humana: en el momento que una persona aprende algo lo suficientemente bien, deja de ser consciente de ello y lo incorpora como algo natural. En el momento que esto sucede, uno puede dejar de pensar explícitamente en el dispositivo o herramienta y puede comenzar a utilizarlo para alcanzar metas más ambiciosas.

Volviendo entonces al futuro que imaginó Weiser, es muy probable que en éste momento, imaginar una sala con cientos de dispositivos con capacidad de cómputo a nuestro servicio suene un tanto ficticio (y hasta casi intimidatorio), pero si realmente logramos comprender esta tecnología, al punto que nos sea invisible, esos cientos de dispositivos pasarán inadvertidos como hoy lo hacen los electrodomésticos que se encuentran en cualquier hogar.

Hasta el momento se han implementado una cantidad importante de sistemas de este estilo, pudiendo nombrarse, entre otros, a los que abren la puerta automáticamente al reconocer a la persona que trabaja en una determinada oficina, centrales telefónicas que redireccionan las llamadas al teléfono más cercano al que se encuentre el destinatario y agendas qué deciden cómo enviar recordatorios a su portador dependiendo de la actividad que esté llevando a cabo. A pesar de ésto, todavía queda un largo camino por recorrer; no existe hasta el

momento una metodología que nos indique cómo se deben desarrollar este tipo de aplicaciones, así como tampoco contamos con herramientas lo suficientemente genéricas y flexibles para facilitar su construcción. Por último, todavía debemos determinar cómo realizar la transición para que la sociedad adopte en forma gradual este tipo de tecnologías.

2.2.3 Computación basada en la posición del usuario

Uno de los primeros trabajos en aplicaciones basadas en la posición del usuario es el realizado en torno al sistema *active badge* [54]. En este sistema, los usuarios llevan consigo un pequeño dispositivo (del tamaño de una credencial) que está constantemente emitiendo, en la gama del infrarrojo, un identificador único para el sistema. Luego, por medio de un conjunto de receptores y una red de hosts, se genera un mapa que permite identificar la ubicación de las personas dentro de un edificio. Este tipo de tecnología luego permitió el desarrollo de diversas aplicaciones, como la posibilidad de tener una pantalla que indique donde se encuentra cada persona, el desarrollo de *scritps* de control personal, así como la posibilidad de ejecutar un comando Unix en el momento que un usuario ingresaba o salía de una determinada área (asimismo el comando era ejecutado en la workstation que se encontrara más cercana al usuario).

Las aplicaciones que modifican su comportamiento en base a la posición del usuario son, de alguna forma, el origen de lo que llamamos aplicaciones context-aware. El trabajo de Schilit [51], uno de los precursores en el área, se basó en la adaptación de las aplicaciones en base a un elemento particular del contexto, que es la posición del usuario. La utilización de dispositivos inalámbricos con el agregado del conocimiento de la posición del usuario, nos permite diseñar una gama muy amplia de sistemas context-aware, lo que hace que este tipo de aplicaciones tengan una especial atención dentro del área.

2.3 ¿De qué hablamos cuando hablamos de contexto?

En general, todos tenemos una idea intuitiva de lo que significa el contexto. De hecho, hasta el momento hemos mencionado al contexto sin dar una definición formal del mismo. En un sentido amplio, el contexto indica el entorno (ya sea político, cultural, histórico, etc) en el que se considera un hecho o se lleva a cabo una actividad[2]. A pesar que esta definición captura la idea, dado que el contexto se va a encontrar en el foco de nuestro estudio, es necesario ahondar un poco más y comprender realmente a que nos referimos cuando hablamos de

[2]Notar que no nos referimos al contexto en el sentido lingüístico, del cual depende el sentido o valor de una palabra o frase.

contexto. Una vez que tengamos ésto claro podremos plantear los requisitos para desarrollar la arquitectura deseada.

Para comprender más en detalle qué es el contexto, resulta útil hacer un análisis de las definiciones de contexto que han dado diversos autores hasta el momento. Es importante marcar que, dado que la disciplina es relativamente nueva [3] y está siendo desarrollada paulatinamente, las definiciones de contexto muchas veces están dadas para adecuarse las necesidades de una aplicación o proyecto en particular.

Las definiciones de contexto pueden ser separadas en dos grandes familias: en un primer plano aparecen las que intentan dar una definición por extensión, enumerando aquellas cosas que deben ser consideradas parte del contexto. En un segundo plano, aparecen aquellas que no son más que definiciones alternativas a las que podemos encontrar en un diccionario, por lo que no brindan ningún valor agregado.

Dentro de la primera familia de definiciones, encontramos por ejemplo la que brinda Schilit en su trabajo [50] en la que se refiere al contexto como la ubicación, identidad de las personas y objetos en las cercanías y los cambios en dichas entidades. En forma similar, Brown [7] define al contexto como la ubicación, la identidad de las personas alrededor del usuario, la hora, la temperatura, la estación del año, etc. En forma un poco más elaborada, Dey [15] define al contexto como el estado emocional del usuario, su foco de atención, su ubicación y orientación, la fecha, hora y los objetos que se encuentran dentro del ambiente del usuario.

A pesar de ser concretas, estas definiciones plantean un problema a la hora de decidir si un determinado tipo de información, que no se encuentra listado, es efectivamente parte del contexto o no. Por este motivo, esta clase de definiciones son candidatas a quedar obsoletas en cuanto incorporemos nuevos aspectos de contexto a nuestras aplicaciones.

En la segunda familia de definiciones encontramos que el contexto debe ser visto como el ambiente o la situación actual. Estas definiciones son tan amplias que ni siquiera especifican si se refieren al ambiente o situación del usuario o de la aplicación. En [6] Brown define al contexto como todos aquellos elementos del ambiente del usuario que pueden ser de interés para otros usuarios o para alguna aplicación. Franklin [24] ve al contexto como la situación del usuario, al tiempo que Ward [54] lo define como el estado del entorno de la aplicación.

A diferencia de las anteriores, estas definiciones tienen una naturaleza abstracta; de hecho, son tan abstractas que no proveen más información que la que ya sabemos en forma intuitiva, por lo que no agregan ningún valor sustancial a nuestra investigación.

[3] A pesar que se habían realizado algunos trabajos previos, las primeras referencias de aplicaciones context-aware datan del año '95 con el trabajo de Schilit [51].

A medida que fue pasando el tiempo se ha refinado la noción de contexto, intentando lograr un balance entre las dos familias de definiciones. Por ejemplo, en una de sus publicaciones Schilit [49] indica que contexto está constantemente cambiando durante la ejecución de una aplicación y que se compone de tres grandes aspectos:

- El ambiente a nivel de computación, que incluye a los procesadores disponibles, los recursos de conectividad, la capacidad de la red, etc.
- El ambiente del usuario, que indica cosas como su ubicación, las personas a su alrededor, la situación social, etc.
- El ambiente físico, como por ejemplo la temperatura, el nivel de ruido, el espacio disponible para que el usuario se mueva, etc.

Por su parte Dey [16] define al contexto como cualquier tipo de información que indique el estado físico, social o emocional del usuario.

A pesar de no hacer una enumeración puntual, las anteriores definiciones siguen siendo demasiado específicas. Dado que el contexto abarca a la situación completa en la que se encuentra inmerso el usuario, y las situaciones varían constantemente, cualquier definición que nos ate a una situación en particular puede ser descartada automáticamente.

En vistas de ésto, Dey [18] opta por dar la siguiente definición:

> El contexto es cualquier información que pueda ser utilizada para caracterizar la situación de una entidad. Una entidad puede ser una persona, un lugar o un objeto que es considerado relevante para la interacción entre el usuario y una aplicación, eventualmente incluyendo al usuario y a la aplicación misma.

Como complemento luego Dey indica que algunos tipos de contexto son más importantes que otros, como por ejemplo la ubicación, la identidad del usuario, la hora y la actividad que se está llevando a cabo. Ésto se debe a que este tipo de información no solo puede ser utilizada para responder interrogantes directos (como por ejemplo, saber donde está ubicado el usuario) sino también para obtener información en forma indirecta (por ejemplo, quienes se encuentran cerca[4] del usuario). Esta última definición de contexto, a pesar de no ser completamente satisfactoria, brinda un balance bastante adecuado entre las dos familias mencionadas anteriormente y es la que utilizaremos a lo largo del trabajo.

[4]Notar que la noción de cercanía es subjetiva y generalmente depende de la aplicación.

2.4 ¿Qué es *context-awareness*?

Al igual que las definiciones de contexto, la noción que una aplicación esté consciente de su contexto ha pasado por una serie de etapas. En un conjunto de primeras definiciones, los autores indicaban que una aplicación era context-aware si podía *adaptarse* o *responder* ante los cambios en su contexto ([5, 50] entre otros). Definiciones más elaboradas definían a la computación context-aware como la habilidad que poseen los dispositivos de detectar, sensar, interpretar y responder a los aspectos locales al ambiente de un usuario [43]. Siguiendo esta línea, Dey [15] define la noción de context-awareness como el uso del contexto para automatizar un sistema de software, modificar su interface y proveer la máxima flexibilidad en términos de servicios.

Una segunda categoría de definiciones se centra en la idea que una aplicación es context-aware si se puede *adaptar* al contexto, lo cual pone al sistema en una posición un tanto más pasiva que en el caso anterior. Un ejemplo claro es la definición dada por Kortuem [38], donde una aplicación context-aware es aquella que pueda variar o adaptar dinámicamente su comportamiento en base al contexto. Por último, en una forma un poco más subjetiva, Dey [18] define la noción de un sistema context-aware como aquel que utiliza al contexto para proveer información relevante y/o servicios al usuario, donde la relevancia depende de la tarea que está llevando a cabo el usuario. Esta última definición es la mas interesante, ya que presenta al sistema como un proveedor (tanto de servicios como de información) y lo sitúa en un contexto activo, que es la tarea que está realizando.

2.5 Características de las aplicaciones context-aware

Es natural pensar que, junto con las definiciones de contexto y context-awareness, se intente desarrollar una taxonomía de aplicaciones context-aware. El primer intento de realizar este tipo de clasificación apareció en el trabajo de Schilit [49], en el cual se presentaban dos dimensiones ortogonales: una dimensión indicaba si la aplicación debía recolectar información o ejecutar algún tipo de acción. La segunda dimensión indicaba si la activación era manual o automática. En base a estas dos dimensiones se definen cuatro tipos de aplicaciones:

Selección por cercanía: describe a aquellas aplicaciones que brindan información de contexto al usuario en forma manual. Estas aplicaciones se basan en un paradigma de interacción en el cual la relevancia en que se presentan los objetos al usuario depende de la distancia a la que se encuentren.

Reconfiguración de contexto automática: son aquellas aplicaciones que brindan información de contexto al usuario en forma automática. Un ejemplo de este tipo de aplicaciones son las que indican que recursos se encuentran disponibles para el usuario, y se actualizan automáticamente ante los cambios de contexto.

Comandos contextuales: engloba a las aplicaciones que permiten ejecutar comandos basados en el contexto en forma manual. La disponibilidad de los servicios depende del contexto del usuario, pero es éste y no el sistema quien decide ejecutar el comando.

Acciones activadas por contexto: define a las aplicaciones que ejecutan comandos automáticamente en base al contexto. Son modeladas como un conjunto de reglas de la forma if-then, indicando qué comando ejecutar en el caso que el contexto actual coincida con el condicional de la regla.

Por su parte, en lugar de clasificar las aplicaciones context-aware por su forma de manejar el contexto, Pascoe [43] define un conjunto de características genéricas que se encuentran en sistemas context-aware:

Sensado del contexto: es el nivel más básico de context-awareness que puede tener una aplicación. Su única función es detectar los estados del ambiente y presentarlos al usuario en forma apropiada, aumentando efectivamente el sistema sensorial del usuario.

Adaptación conextual: es la capacidad que muestra una aplicación de modificar su comportamiento en base al contexto, adaptándose así a la situación del usuario.

Descubrimiento de recursos: denota la capacidad de descubrir y utilizar otros recursos que se encuentran disponibles en el contexto de la aplicación. Notar que esto es una actividad constante y dinámica, ya que depende fuertemente del contexto actual.

Aumento contextual: es la capacidad de enriquecer al ambiente con información adicional. Dependiendo de la visión del usuario, podemos tomarlo como un aumento de la información física con información digital o viceversa.

Por último Dey [18] también propone una categorización, combinando las ideas de los trabajos previamente mencionados:

Presentación, ya sea de información o servicios. Sería equivalente a combinar la selección por cercanía y los comandos contextuales que aparecen en la taxonomía de Schilit.

Ejecución automática de servicios, que sería equivalente a las acciones activadas por contexto de Schilit o a la adaptación contextual de Pascoe.

Etiquetado del contexto, con el objetivo de ser examinado posteriormente. Es el equivalente a la noción de aumento contextual de Pascoe.

Esta última clasificación tiene la peculiaridad que no distingue, en principio, entre información y servicios. Asimismo, no considera al descubrimiento de recursos como una característica separada, sino como parte del proceso de selección de servicios en base al contexto.

En general, el objetivo de este tipo de clasificaciones es poder determinar si una aplicación es efectivamente context-aware o no, y en caso de serlo, establecer sus principales características para saber que debemos tener en cuenta a la hora de desarrollarla.

2.6 Dificultades al tratar con aplicaciones context-aware móviles

De la misma forma que las aplicaciones que trabajan sobre un dominio común (por ejemplo, aplicaciones bancarias) deben solucionar determinados problemas generales, las aplicaciones context-aware deben resolver, al menos, las siguientes dificultades:

1. Hay una diferencia de impedancia importante entre los datos recogidos por los sensores y los elementos de contexto que debe manipular una aplicación. Un ejemplo de esta diferencia es decidir, en base a la posición de usuario y la cantidad de personas presentes en una sala, si el usuario encuentra o no en una reunión. Es importante notar que ésto no solo implica un paso de abstracción de datos, sino también la necesidad de combinar los valores adquiridos por distintos sensores, que pueden estar distribuidos geográficamente.

2. La separación de responsabilidades se vuelve un punto crítico a tratar en aplicaciones context-aware, ya que están compuestas por una gran cantidad de componentes ortogonales que deben evolucionar independientemente. A modo de ejemplo, los mecanismos de sensado no deberían influir en el modelo de contexto, así como un cambio en los servicios no debería implicar un cambio en el modelo de aplicación.

3. El contexto es naturalmente dinámico, lo que implica que los cambios deben ser capturados en tiempo real y las aplicaciones deben responder en forma acorde. En estas aplicaciones no es concebible que los servicios se presenten en forma tardía.

4. Al igual que en otras aplicaciones, es necesario contar con algún sistema de persistencia, lo cual implica tomar decisiones respecto de donde se debe guardar la información (en el dispositivo de usuario o en un servidor centralizado).

Además de los problemas propios que plantean las aplicaciones context-aware móviles, también tenemos que lidiar con un conjunto de problemas provenientes de otras disciplinas:

1. *Interfaces gráficas y HCI:* trabajar con dispositivos móviles como pueden ser PDAs representa un cambio importante en el modelo de interacción hombre-máquina, respecto del que encontramos al utilizar una máquina de escritorio. Una de las características que surgen a primera vista es el tamaño de la pantalla, lo que impacta fuertemente en sistemas como el que se plantea en este trabajo, ya que hay que presentar una cantidad de información potencialmente grande en un espacio gráfico reducido. Otro tema a tener en cuenta es el cambio en la interacción hombre-máquina dado por los dispositivos de entrada utilizados en las PDAs: mientras que en una PC de escritorio utilizamos mouse y teclado, las PDAs utilizan un lápiz óptico. Asimismo, la atención que el usuario le pueda prestar a una aplicación, el tiempo de concentración y las diversas situaciones en las que éste se encuentre afectan directamente al diseño de la aplicación.

2. *Mecanismos de sensado:* para lograr que las aplicaciones modifiquen su comportamiento en base a su contexto es necesario brindarle medios para adquirir información sobre éste. Hoy por hoy la tecnología de sensado ha avanzado mucho y ha alcanzado un nivel de madurez que hace posible pensar en agregar mecanismo de sensado a dispositivos móviles. A pesar de ésto, no tenemos una experiencia importante en la utilización de estos dispositivos, lo cual hace difícil de predecir los tiempos de desarrollo de aplicaciones. Asimismo debemos tener en cuenta la relación funcionalidad-costo así como el hecho que muchos dispositivos deben ser importados de otros países. En el Apéndice C se puede encontrar un relevo de los mecanismos de sensado de posición para ambientes cerrados disponibles en el mercado.

3. *Comunicación:* un punto importante a tener en cuenta es la forma en que se va a llevar a cabo la comunicación entre dispositivos y cuál es modelo adecuado para este tipo de aplicaciones. En la mayoría de las aplicaciones context-aware se plantea un modelo cliente-servidor, donde en los casos más extremos el servidor es quien se encarga de hacer todo el trabajo y el dispositivo móvil es simplemente una interfaz gráfica (generalmente una página web).

4. *Manejo de dispositivos externos:* en muchos casos, una aplicación context-aware depende de dispositivos externos, como puede ser el caso de un GPS para indicar la posición del usuario o las redes inalámbricas para obtener información de los lugares que el usuario visita. En cualquier caso no se puede asegurar que no habrá fallas, por lo que el sistema debe ser lo suficientemente robusto como para manejar dichas fallas. A modo de ejemplo, un sistema de guía turístico debería soportar una conexión wireless intermitente, así como una pérdida de señal del sensor GPS sin que la aplicación finalice abruptamente.

2.7 Un enfoque alternativo de contexto

El objetivo de esta sección es resumir el trabajo realizado por Dourish [19], quien plantea una forma de analizar al contexto desde el punto de vista de la interacción hombre-máquina (*HCI*). Como puede verse en las secciones anteriores, la gran mayoría de los desarrollos informáticos abordan la noción de contexto en términos de los problemas que surgen al querer manejarlo, como por ejemplo, el manejo de sensores, los problemas relativos a la conectividad de red y las formas distribución de información. Al mismo tiempo, lo que uno desearía al agregar un soporte para manejo de contexto en los sistemas interactivos, es que éstos se tornen más sensibles al usuario, ésto es, en términos humanos. El riesgo que corremos al tomar un enfoque tan cercano a la máquina para resolver problemas relativos a lo humano, es que podemos perder de vista que el entorno está formado en gran parte por el marco social y no por un conjunto de señales captadas por sensores.

2.7.1 Dos visiones de contexto

La noción de contexto dentro del área de computación ubicua tiene un origen dual. Por un lado tiene una base tecnológica, cuyo objetivo es ofrecer a los desarrolladores herramientas para incorporar aspectos humanos a sus aplicaciones. Por otro lado, tiene un origen en las ciencias sociales, en la cual se le presta atención a una serie de aspectos del entorno social en que está inmerso el usuario. Para analizar esta dualidad, Dourish plantea dos enfoques para el manejo del contexto: el *positivista* y el *fenomenológico*.

El razonamiento positivista, planteado originalmente por Comte, indica que los estudios científicos deben acotarse al análisis de las relaciones entre los hechos que sean directamente observables y comprobables. Este pensamiento, a todas luces racionalista, intenta establecer un método científico de trabajo análogo al que se utilizan en las ciencias duras, donde se busca reducir los fenómenos complejos a un modelo matemático ideal. Llevar esta idea a nuestro trabajo implica buscar descripciones independientes y objetivas de los fenómenos sociales,

descartando aquellos detalles subjetivos para llegar a un modelo formal.

En contraste con este modelo, las teorías fenomenológicas se dedican a describir las estructuras de las experiencias humanas, en base a cómo se presentan ante nuestra consciencia. Es importante notar que, a diferencia del modelo positivista, este análisis se realiza en términos de la persona que es el foco de la experiencia, resultando naturalmente subjetivo. Asimismo, la teoría fenomenológica rechaza la utilización de mecanismos de deducción, inducción o cualquier aserción proveniente de otras disciplinas formales. Bajo esta óptica, los fenómenos sociales no poseen ninguna realidad objetiva que vaya mas allá de la habilidad que tengan los grupos de reconocerlos y orientarse hacia ellos. Dichos fenómenos aparecen como propiedades emergentes de la interacción y no como estructuras preconcebidas que una persona o grupo está obligada a respectar. Asimismo, estos fenómenos están en un constante proceso de interpretación y redefinición, lo cual implica descartar la idea de un modelo del "mundo real" estable, para pensarlo como una cuestión de interpretación de cada persona.

La principal importancia en dejar clara esta distinción es que los enfoques basados en técnicas de ingeniería suelen tomar una posición positivista, al tiempo que los análisis sociales que son relevantes en el área de la interacción hombre-máquina tienden a adoptar una posición fenomenológica.

2.7.2 El contexto como un problema de representación

En términos de un pensamiento positivista, el manejo del contexto es un problema de representación. De hecho, los sistemas basados en computadoras se basan en la representación de modelos en una máquina, por lo que resulta natural intentar pensar al contexto como un conjunto de símbolos codificados en una máquina. Independientemente de la definición de contexto que usemos, veremos que todas comparten cuatro características:

- *El contexto es una forma de información.* Tal vez esta sea la característica más importante, ya que establece que el contexto es algo que puede ser completamente conocido y por lo tanto codificado.

- *El contexto es determinable.* Esto implica que, para una aplicación o conjunto de aplicaciones, podemos definir qué cosas son parte del contexto y qué cosas no. Esto a su vez nos permite planificar cómo se va a comportar nuestra aplicación *a priori*.

- *El contexto es estable.* A pesar que los elementos de contexto que tomamos en cuenta pueden variar entre aplicaciones, la definición de los mismos se mantiene estable, esto es, no dependen de una actividad o evento.

- *El contexto y la actividad pueden separarse.* Bajo este paradigma, una actividad es algo que sucede *en* un contexto; el contexto describe un conjunto de características del ambiente en el que se lleva a cabo una actividad, pero que se encuentra separado de la actividad en si misma.

Independientemente de cualquier definición de contexto que usemos, estas cuatro propiedades, asignadas al contexto, se encuentran presentes en la mayoría de los desarrollos. En particular, la idea que el contexto consiste de un conjunto de características del entorno alrededor de las cuales se llevan a cabo actividades y que a su vez estas características pueden ser codificadas, resulta algo común en la mayoría de los sistemas.

2.7.3 El contexto como un problema de interacción

Desde el punto de vista de las ciencias sociales, la crítica que se hace al método positivista para modelar el contexto es que, un modelo basado en los cuatro principios anteriores, no expresa lo que el contexto realmente es. Bajo esta óptica se plantean principios opuestos:

- *El contexto no es información.* Alternativamente, lo que existe es una relación de *contextualidad* entre dos objetos o actividades. No es una cuestión de que algo sea o no contexto, sino que sea contextualmente relevante a una actividad u objeto.

- *El ámbito de las propiedades contextuales varía dinámicamente.* Esto implica que el contexto no puede definirse ni molderase de antemano.

- *El contexto es una propiedad ocasionada.* Esto implica que el contexto no es estable y que varía de situación en situación, dependiendo de las instancias particulares de cada acción y objeto que participa en ésta.

- *El contexto surge de la actividad.* Bajo este paradigma no hay una distinción entre la actividad y los "datos" del contexto, sino que el contexto es algo que se produce y modifica activamente, como emergente de llevar a cabo una actividad.

Si analizamos al contexto como un problema de representación, las preguntas básicas que debemos responder son "¿Qué es el contexto?" y "¿Cómo lo puedo codificar?". En esta nueva propuesta, donde el contexto se ve como un problema de interacción, la pregunta que nos debemos responder es "¿Cómo y por qué, al interactuar con el medio, las personas logran establecer y mantener un acuerdo mutuo respecto del contexto de sus acciones?". Esto deja bien en claro las diferentes posturas: mientras que desde la perspectiva positivista el contexto es una cualidad estable del mundo, independiente de las acciones de las personas,

la visión fenomenológica indica que la *contextualidad* existe cuando dos o más partes que interactúan la reconocen.

2.8 Modelos de contexto

Como hemos visto en las secciones anteriores, no sólo no existe un consenso respecto de qué tipo de problema estamos atacando al querer incorporar el contexto a nuestras aplicaciones (representación vs. interacción), sino que, entre los diversos investigadores que lo ven como un problema de representación, no se ha llegado a una definición consensuada de qué es el contexto. En esta sección veremos tres enfoques para modelar el contexto, cada uno representativo de un grupo mayor de trabajos.

2.8.1 Modelos formales basados en datos

Como indican Henricksen *et al.* [32, 31], una posibilidad para modelar el contexto es utilizar alguna técnica formal ya probada, como pueden ser los diagramas Entidad-Relación o UML. A pesar de ésto, los autores señalan que los modelos existentes no resultan naturales ni apropiados para esta tarea, ya que las construcciones no tienen la riqueza semántica necesaria. Por este motivo se define un modelo formal *ad-hoc*, con un conjunto de conceptos particulares para el modelado de contexto, junto con una notación gráfica.

La técnica presentada aborda el modelado de contexto con una visión basada en objetos. La información de contexto se estructura en torno a un conjunto de entidades, donde cada entidad describe a un objeto, ya sea físico o conceptual. Las propiedades de las entidades, como puede ser el nombre de una persona o la cantidad de puertas de un auto, se representan por medio de atributos. Por último, una entidad se encuentra unida a sus atributos por medio de asociaciones, que son relaciones unidireccionales que indican propiedad (una entidad *es dueña de* un atributo). Como se puede ver en la figura 2.1, se puede utilizar una notación gráfica para expresar éstos conceptos.

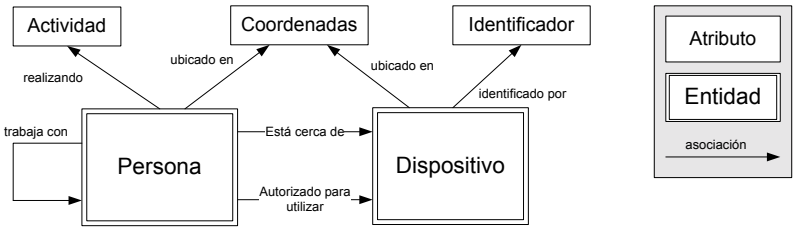

Figura 2.1: Notación gráfica para el modelo

En el modelo planteado, las asociaciones pueden clasificarse según su tipo para agregar riqueza semántica:

Estáticas: Son aquellas relaciones que se mantienen constantes con el paso del tiempo respecto de la entidad que la posee.

Dinámicas: Son contrarias a las estáticas y se clasifican dependiendo de la fuente.

- *Sensadas*, cuyos valores son obtenidos en base a sensores de hardware o software. Este tipo de información suele tener ruidos y puede estar desactualizada al momento de ser procesada por la aplicación.
- *Derivadas*, se obtienen en base a otras asociaciones por medio de una función de derivación. La función puede ser tan simple como una ecuación lineal o tan compleja como un algoritmo de IA.
- *En base al perfil de usuario*, lo que implica que éste debe crearlas e ingresar sus valores explícitamente. Este tipo de información suele ser de confianza, aunque puede estar desactualizada.

Las asociaciones pueden ser a su vez clasificadas en simples o compuestas, siendo éstas últimas subdivisibles en colecciones, alternativas y temporales. En la figura 2.2 se muestra un diagrama con las extensiones planteadas.

Por último, el modelo planteado permite especificar relaciones de dependencia y aplicar a cada parámetro una métrica (por ejemplo de confianza o calidad).

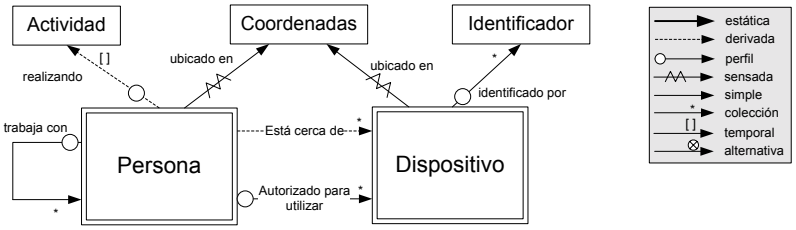

Figura 2.2: Distintos tipos de asociaciones

Es importante notar que, a pesar de plantear un modelo alternativo al de Entidad-Relación, el modelo presentado en esta sección tiene una transformación directa al modelo relacional de bases de datos.

2.8.2 Contextors: datos y transformaciones

El modelo planteado en la sección 2.8.1 está orientado a datos (entidades y atributos) y las relaciones (asociaciones) que se establecen entre éstos. En su trabajo, Coutaz y Rey [11] se separan un poco de la visión del contexto como un conjunto de datos e indican que un contexto debe estar siempre relacionado a un usuario realizando una determinada tarea. En base a ésto, el contexto de un usuario U para una tarea T en un instante de tiempo t, viene dado por la composición de los distintos estados por los que fue pasando el usuario U al llevar a cabo T a lo largo del tiempo.

$$contexto^{U,T}(t) = COMPOSICION(situacion^{U,T}(t_0), \ldots, situacion^{U,T}(t))$$

donde

$situacion^{U,T}(t)$ indica la situación del usuario U en un momento t. Una situación es descripta por el conjunto de valores relevantes para el usuario llevando a cabo una tarea T, que pueden ser observados en el momento t.

Dado que una situación es básicamente una *n-upla*, lo que debemos agregar es una forma de actuar sobre ésta. Con este objetivo los autores crearon la idea de *contextor*, que es un dispositivo de software que maneja cuatro canales de comunicación: *entrada de datos, salida de datos, entrada de control y salida de control*. Como se puede ver en la figura 2.3, un contextor modela básicamente una transformación, tomando el(los) valore(s) por su canal de entrada de datos y retornando el resultado por su canal de salida de datos. Los contextors, a su vez, poseen un canal de entrada de control para recibir órdenes (por ejemplo, comenzar la ejecución) y un canal de salida de control para poder comandar otros contextors.

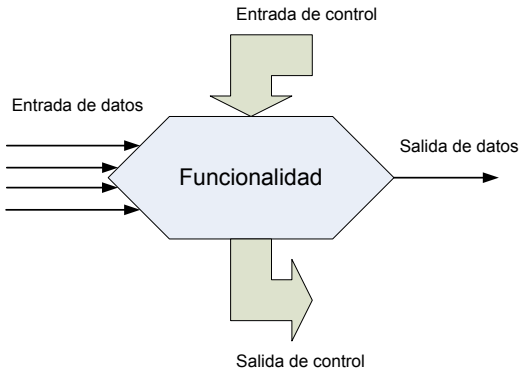

Figura 2.3: Esquema de un contextor

Para construir aplicaciones context-aware en forma sencilla se utiliza una taxonomía de contextors predefinida. Por cuestiones de espacio solo se mostrarán algunos ejemplos de contextors:

Elemental. Encapsula un sensor físico, por lo que no posee un canal de entrada. El canal de salida retorna los valores medidos y el canal de control puede utilizarse para configurar al sensor (por ejemplo, el intervalo de muestreo que debe utilizar).

Historial. Se utiliza para llevar un historial de valores. Los tipos de valores de sus canales de entrada y salida son idénticos.

Traductor. Representa una función en el sentido matemático: dado un valor de entrada realiza una conversión para generar un valor de salida, el cual generalmente es de un dominio distinto al del valor entrada.

Este modelo hace una separación clara entre datos y los procesos que actúan sobre éstos, que, desde la perspectiva de los paradigmas de programación, se asemeja a lo que plante el paradigma procedural estructurado.

2.8.3 Un modelo orientado a objetos

El JCAF (*Java Context Awareness Framework*) [1] busca lograr una infraestructura liviana y robusta para crear aplicaciones context-aware. Dado que el objeto de estudio de esta sección es la forma de modelar el contexto, no analizaremos al framework en toda su extensión, sino que nos concentraremos en el modelo de contexto que plantea.

El manejo del contexto se basa en tres conceptos fundamentales: *entidades*, *contextos* y *elementos de contexto* (ver figura 2.4). Las entidades se utilizan para modelar objetos de la vida real, como personas, camas o automóviles. Una entidad tiene asociado un contexto, al cual conoce directamente (esto es, una entidad puede manipular su contexto). Un contexto es simplemente una colección de elementos de contexto, el cual debe ser especializado en cada aplicación (por ejemplo, se debe definir un contexto para un hospital, para una cama, etc). Los elementos de contexto representan aquellos datos que conforman el contexto de una entidad; estos elementos poseen un identificador único y es posible saber el nivel de precisión de sus datos, así como si la conexión es segura.

Para adquirir valores externos se utiliza un objeto de la clase `Monitor`, el cual se encarga de proveer a un objeto entidad los elementos de contexto. En el caso que haya que realizar alguna transformación entre dominios (por ejemplo, convertir un valor GPS en una dirección) se utilizan objetos *transformadores*; éstos toman un elemento de contexto de un dominio y retornan otro de un dominio distinto (notar la similitud entre monitores y transformadores y los contextors elementales y traductores). Por último, las entidades se encuentran

en un repositorio de entidades, almacenadas en una estructura de la forma *llave-valor* que permite que otros objetos puedan realizar búsquedas.

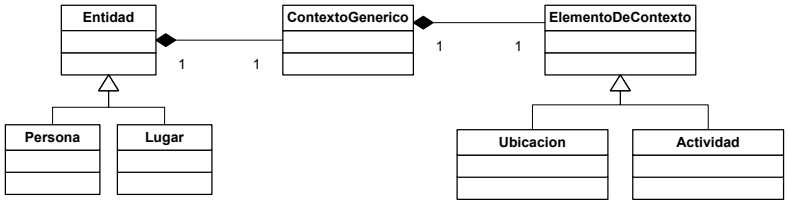

Figura 2.4: Diagrama de clases del JCAF

Lamentablemente el modelo planteado por Bardram no puede escapar a la definición de *orientado* a objetos, ya que, a pesar de utilizar las construcciones del paradigma OO, sigue pensando el términos de datos y procesamiento sobre éstos. Los ítems de contexto, así como el contexto mismo, no son más que datos sobre los cuales otros objetos realizarán algún tipo de procesamiento. A pesar de ésto, el JCAF es un paso adelante respecto de las representaciones basadas en datos y funciones

2.9 Conclusiones

La posibilidad de tener un conjunto de pequeños dispositivos de computación transportables e interconectados por redes inalámbricas, permite el desarrollo de un nuevo tipo de aplicaciones. Estas aplicaciones deberían integrarse a la vida del usuario, asistiéndolo en sus tareas diarias. Dentro de este nuevo tipo de aplicaciones aparecen aquellas que pueden reaccionar al contexto en el que se están ejecutando, con el objetivo de brindar mejores servicios al usuario.

En términos del contexto, no existe una definición ampliamente aceptada, así como tampoco existe un consenso respecto de cómo debemos encarar el manejo de éste (problema de representación vs. problema de interacción). Actualmente se trabaja con una visión de contexto proveniente de las ciencias exactas, intentando enumerar sus propiedades y representarlo formalmente en una computadora.

En términos de dificultad, las aplicaciones context-aware no sólo deben lidiar con un conjunto de problemas propios (diferencia de impedancia entre los datos sensados y los que utilizan las aplicaciones, componentes ortogonales que evolucionan independientemente, etc.) sino que también deben manejar problemas que se encuentran en otras disciplinas (interfaces gráficas, comunicación entre máquinas, etc.). Esto hace que las aplicaciones context-aware alcancen niveles de complejidad importantes, lo que hace aún mas importante la necesidad de contar con una arquitectura flexible y escalable.

Capítulo 3

Estado del arte

Todos los animales son iguales,
pero algunos son más iguales que otros.

Rebelión en la granja
George Orwell

Desde los primeros experimentos en el área de aplicaciones context-aware se han planteado diversas arquitecturas, toolkits y librerías para facilitar el desarrollo de este tipo sistemas. El objetivo del capítulo es presentar algunos de desarrollos realizados en el área, para así poder comprender a que nos enfrentamos, cuáles han sido los problemas encontrados y las soluciones propuestas hasta el momento. Para ésto comenzaremos viendo algunos ejemplos de aplicaciones context-aware concretas y luego pasaremos a ver a soluciones mas generales en forma de arquitecturas y frameworks.

3.1 Investigación de aplicaciones context-aware

El objetivo de la presente sección es mostrar un conjunto de aplicaciones context-aware que ilustren cómo se puede utilizar el contexto para brindar servicios al usuario o mejorar la funcionalidad de aplicaciones existentes. Dado que un relevamiento exhaustivo de todas las aplicaciones excede a este trabajo, se describirán aquellas que sean representativas de un conjunto más amplio de sistemas.

3.1.1 Cyberguide

El objetivo de CyberGuide [41] es proveer al usuario un asistente context-aware en su visita a un laboratorio en el cual se va a llevar a cabo un demostración. El tipo de aplicaciones que ayudan al usuario durante su recorrido se encuentran probablemente dentro de las aplicaciones context-aware más desarrolladas, ya que han sido objeto de investigación en diversas implementaciones [4, 12, 22, 53, 59]. En estas aplicaciones, los visitantes llevan consigo una PDA, las cuales

muestran un mapa indicando aquellos lugares que pueden ser de interés para la persona. A medida que el usuario se mueve, el sistema debe proveer indicaciones de su posición, reubicar el mapa para ser consecuente con la ubicación del usuario y mostrar aquella información relevante al lugar donde se encuentra. En este tipo de aplicaciones, la información de contexto que se utiliza está muy acotada, siendo generalmente la posición y orientación del usuario.

Para esta aplicación en particular los autores construyeron una versión para trabajar tanto en espacios cerrados (detectando la posición por medio de mecanismos de sensado infrarrojo) como en espacios al aire libre (utilizando GPS como mecanismo de ubicación).

Uno de los grandes problemas identificados por los desarrolladores de Cyberguide es que el sistema requería una red de emisores infrarrojos para detectar la posición de usuario en espacios cerrados, al tiempo que se usaban como mecanismo de conexión inalámbrica entre las PDAs y los hosts. Por la forma en que fue diseñada, la aplicación presentaba un fuerte acoplamiento entre los mecanismos de sensado y los servicios, haciendo que un cambio en los dispositivos de adquisición de datos (por ejemplo, el pasaje a GPS) implicara la re-escritura casi completa del sistema.

3.1.2 Forget-me-not

Forget-me-not [39] se encuentra dentro de un conjunto de aplicaciones context-aware que se utilizan para aumentar la memoria de su usuario [30, 46]. Este sistema fue diseñado para ayudar al usuario en los problemas diarios, como encontrar documentos, recordar nombres y fechas, etc. Para ésto, el trabajo se basa en utilizar al contexto como un marco de búsqueda para encontrar información olvidada.

La idea usada en este sistema no es nueva y ha sido estudiada ampliamente en el campo de la psicología, siendo denominada *memoria autobiográfica* [3]. Según la observación de los psicólogos, los humanos organizamos nuestra memoria para eventos pasados en base a otros episodios, como por ejemplo en términos de las personas que estaban con nosotros, que hicimos antes y/o después o algún hecho que se haya producido en ese mismo momento. Esto motivó a los autores a pensar en algo similar a una prótesis para la memoria, que permita registrar aquellos eventos y episodios que pasan a lo largo de nuestro día, para luego poder usarlo para buscar información que hemos olvidado. Luego, si fuera posible construir un dispositivo que acompañara al usuario constantemente y que pudiera registrar en forma automática toda la información de contexto relevante a su actividad (y estructurarla en forma similar a la que estructuramos nuestra memoria autobiográfica), lograríamos efectivamente una extensión a nuestra memoria.

Para el desarrollo del prototipo, los autores utilizaron como dispositivo móvil un ParcTab [52], que básicamente consiste de un display de 128x64 píxeles. Para la captura de datos se consideró la posición del usuario y los encuentros con otras personas, el lugar de los éstos y la fecha. También se registró la actividad del usuario en su máquina desktop, así como el intercambio de archivos entre personas, los documentos que el usuario había impreso y las llamadas teléfonicas que había realizado. En base a esta información el sistema genera la biografía del usuario, la cual puede ser consultada en cualquier momento. En la figura 3.1 se muestra un ejemplo de este tipo de biografías. Es importante notar que, dado que contaban con muy poco espacio gráfico, la interfaz tuvo que tener un diseño icónico y el mecanismo de interacción con el usuario basado en drag-and-drop.

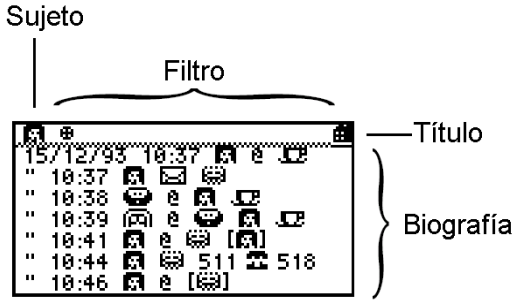

Figura 3.1: Ejemplo de una biografía

Al momento de desarrollar el sistema, los autores identificaron dos grandes problemas:

Tecnológicos: mucha de la información que debían capturar no podía ser sensada. Por ejemplo, los llamados telefónicos debían ser ingresados a mano. Hoy por hoy la tecnología ha avanzado lo suficiente (por ejemplo, existen celulares con PDAs incorporadas) como para pensar en sistemas de este tipo con mayor independencia del usuario.

Volumen de datos: la biografía crece muy rápidamente en el tiempo, lo que requiere de una gran capacidad de almacenamiento y cómputo. Esto plantea un problema particularmente importante para este sistema, ya que la arquitectura utilizada es cliente-servidor y éste se convierte en un cuello de botella. Hoy por hoy, con mayor capacidad de cómputo en los dispositivos personales y la utilización de arquitecturas distribuidas, el problema parecería ser mas tratable. A pesar de ésto, la tasa de crecimiento de los datos históricos sigue siendo un tema crítico.

3.1.3 Audio Aura

El objetivo de esta aplicación es proveer una conexión entre las actividades de una persona en el mundo físico, con información tomada de un mundo virtual [42]. Para ésto, los diseñadores tomaron un enfoque alternativo al clásico paradigma de interacción desktop, donde el diálogo entre el usuario y su máquina es explícito, y establecieron un mecanismo para crear un diálogo implícito. Con este objetivo, los autores identificaron tres requerimientos básicos:

- La información se provee por medio de sonidos.
- La información provista, a pesar de ser relevante y útil, no es crítica para realizar una tarea.
- La información se entrega al usuario como respuesta a una acción en el mundo físico.

La base de Audio Aura se encuentra en aprovechar la capacidad que tenemos los humanos de estar constantemente "sensando" inconscientemente y reaccionar en el caso que algo nos parezca relevante. A modo de ejemplo, durante el día escuchamos sonidos de todo tipo mientras trabajamos, pero (siempre y cuando estén dentro de los límites normales) éstos no nos distraen de nuestra tarea en curso; si de pronto alguien conocido se aproxima y escuchamos su voz, es muy probable que dejemos por un momento nuestra tarea para saludar al recién llegado. Basados en esta idea, los autores del proyecto se aprovecharon nuestras capacidades naturales, para crear una interfase que enriquezca nuestro mundo físico, sin que nos distraiga de la tarea que debemos realizar.

En lugar de tener música funcional, Audio Aura provee una especie de flujo de información de fondo que el usuario puede aprovechar; con este mecanismo, el usuario elige que información utilizar, sin que ésta invada su vida. Debe quedar claro que el tipo de información que provee el sistema no debe ser crítica, ya que el usuario puede optar por ignorarla; es el tipo de información que uno aprecia tener, pero de la cual uno no debería depender.

Como es de esperarse, el estudio de los sonidos a utilizar es crucial: se debe evitar utilizar clásico paradigma de sonidos de alerta (que es el estándar en la mayoría de las aplicaciones) reemplazándolo por un flujo apacible. A modo de ejemplo, para aprovechar este sistema en una oficina, los autores plantean un modelo auditivo de una playa: la cantidad de mails de una persona se transforman en los graznidos de gaviotas, un emisor en particular de mails puede ser asociado con el sonido de algún animal de la fauna marina y la cantidad de personas en un aula, con la frecuencia que rompen las olas en el mar. El lector interesado puede encontrar más información respecto de este tipo de aplicaciones en las publicaciones de Gaver *et al.*[57], Hudson y Smith[34] e Ishii y Ullmer[36].

3.1.4 In/Out Board

In/Out Board [48] es una aplicación sencilla que se utiliza para saber que personas se encuentran dentro o fuera de un edificio y, dependiendo de esta información, su hora de entrada o salida. Para la visualización de la información se utiliza un panel como el que se muestra en la figura 3.2. A pesar de no ser una aplicación compleja (de hecho la información de contexto que usa está muy acotada), resulta representativa de un conjunto de aplicaciones que básicamente se encargan de mostrar la información contextual que reciben. Para detectar la presencia de una persona, los autores utilizaron Java iButtons junto con un lector de iButton o bien un PinPoint 3D-iD [44], basado en radio frecuencia.

Figura 3.2: Interfase gráfica del In/Out Board

Uno de los puntos interesantes de esta aplicación viene dado por las distintas interfaces por las que el sistema puede ser accedido:

- Por medio de una aplicación de escritorio, la cual se muestra en un monitor en la entrada del edificio (figura 3.2).
- Por medio de browser web, lo cual permite acceder remotamente y saber que personas se encuentran en el edificio.
- Por medio de una aplicación que se ejecuta en una PDA.

Es importante notar la diferencia entre la tercera vista del sistema y las otras dos: dado que la aplicación se está ejecutando junto con el usuario, la noción de estar dentro o fuera viene dada por la propia ubicación de la PDA y por el lugar contra el cual se esté comparando. En los otros casos las aplicaciones funcionan como contenedores de información que son actualizados en forma externa.

3.1.5 Conference Assistant

La aplicación Conference Assistant [17] es probablemente una de las más complejas que se han planteado hasta el momento en el desarrollo de sistemas context-aware. El objetivo de esta aplicación es ayudar a una persona en su asistencia a una conferencia. Para ésto, la aplicación debe llevar cuenta de las personas que asisten a la conferencia, sus colegas, las presentaciones que pueden ser de interés, el cronograma de la conferencia, noción temporal y las actividades que se están llevando a cabo, etc. Para comprender el objetivo de la aplicación, a continuación se presenta una descripción del uso típico que se le daría al sistema: supongamos que una persona se registra para una conferencia. Al llegar y registrarse el primer día, el usuario provee información personal (nombre, e-mail, etc) así como una lista de temas que le interesan y el conjunto de colegas con los que asistirá a la conferencia. A cambio de ésto, nuestro usuario recibirá una PDA para manejarse durante la conferencia. La primera tarea de la aplicación es mostrar una grilla con el cronograma de las presentaciones y sus aulas, resaltando aquellas que son del interés del usuario. Basado en ésto, el usuario selecciona una presentación y se dirige al aula correspondiente. Al ingresar al aula, el sistema automáticamente le provee de aquellas cosas relacionadas con la presentación, como los datos del orador y las transparencias que va a mostrar. Asimismo, se le indica si la presentación va a ser grabada o filmada.
Cuando el orador comienza a dar la charla el usuario va siguiendo desde su PDA la transparencia actual que se está mostrando; asimismo, por medio del asistente, puede marcar una transparencia y adjuntarle notas, las cuales puede utilizar más tarde para consultarle cosas al orador. Una vez que la charla ha finalizado, durante la ronda de preguntas, el usuario puede indicar por medio del sistema que la transparencia que está viendo en su PDA pase a ser mostrada en la pantalla principal, facilitando así la comunicación con el orador y el resto del auditorio. Al finalizar la charla el usuario puede ranquear la charla en base a su evaluación personal.
Al salir de la sala, el usuario puede ver las recomendaciones del sistema para tomar otra charla; asimismo puede ver las charlas que tomarán sus colegas y, en el caso que aplique, como ha sido ranqueado por sus colegas el presentador de futuras charlas.
Al finalizar el día, el usuario puede repasar los hechos de la conferencia, obtener el material presentado así como las grabaciones de las charlas, mezclarlos con sus anotaciones y conclusiones y generar un reporte con muy poco esfuerzo.

3.2 Frameworks context-aware

En la sección anterior se presentaron diversas aplicaciones con objetivos concretos, para las cuales la noción de contexto se encontraba perfectamente delineada. Este tipo de aplicaciones están optimizadas para trabajar en un conjunto de escenarios predeterminados y no tienen que ser flexibles o adaptarse rápidamente a cambios en los requerimientos. A pesar que estas aplicaciones resultan muy valiosas para hacer los primeros pasos en el área, lo que realmente se necesita para simplificar el desarrollo de aplicaciones context-aware es un framework o arquitectura con un mayor nivel de abstracción. Dicho framework no solo debería presentar guías para el desarrollo de este tipo de aplicaciones, sino que también debería adaptarse a distintos mecanismos de sensado, permitir registrar nuevos servicios y soportar naturalmente los cambios de requerimientos. En esta sección se presentan diversos frameworks desarrollados con este objetivo, indicando sus principales características.

3.2.1 Schilit's System Architecture

La tesis doctoral de Bill Schilit [51] es uno de los trabajos con mayor influencia en el área de aplicaciones context-aware, tanto por ser uno de los pioneros como por el tipo de arquitectura que plantea. A pesar de las diferencias de diseño e implementación, muchos de los conceptos que plantea Schilit en su tesis pueden ser encontrados en el presente trabajo de grado.

En los sistemas desarrollados (y por ende, en su arquitectura), Schilit se centró especialmente en la ubicación del usuario, diseñando lo que el llamó *active maps*, con la idea de obtener aplicaciones capaces de imprimir un documento en la impresora más cercana o avisar al usuario de la presencia de una persona en particular dentro de la misma habitación. Lamentablemente, esto resulta en una noción de contexto bastante pobre, impidiendo luego adaptar la arquitectura a otros aspectos de contexto.

Para recolectar información sobre el usuario y otros dispositivos, la arquitectura fue planteada en términos de los siguientes componentes:

User agent: los agentes de usuarios funcionan como manejadores y procesadores de información. Por ejemplo, un agente de usuario es el encargado de determinar la ubicación del usuario en base a un conjunto de datos tomados de sensores; al mismo tiempo, el agente puede publicar esta información en el mapa para que sea accedida por otros agentes. A la hora de desarrollar una aplicación, ésta debe registrarse al ambiente del usuario para ser notificada de los cambios.

Device agent: son responsables de monitorear y manejar un dispositivo en particular (por ejemplo, una PDA). Cuando algún atributo del dispositivo cambia, el agente del dispositivo es el encargado de modificar al objeto que representa lógicamente a ese dispositivo. En el caso de estar controlado por un usuario, el agente puede publicar nuevos objetos en el agente de ese usuario; caso contrario lo puede hacer en un mapa global.

Active map: guarda información pública de regiones geográficas y permite que se realicen consultas sobre los objetos que contiene. Asimismo, acepta subscripciones a dicha información. El mapa es actualizado en forma constante y concurrente por los diversos agentes (usuarios y de dispositivos). Para soportar diversos modelos de posicionamiento, los sistemas implementados trabajaban con un híbrido entre sistemas de locación jerárquicos y grafos de distancia.

A pesar de la clara separación conceptual que marca la arquitectura, no provee un mecanismo sencillo para la adquisición y transformación de la información de contexto, lo que implica que el desarrollador de aplicaciones debe programar y acoplar fuertemente los sensores a los objetos correspondientes, lo cual atenta contra la escalabilidad de un sistema. Asimismo, la noción de contexto con la que trabaja está bastante acotada, no permitiendo trabajar en forma sencilla con aspectos de contexto como tiempo o temperatura. Para lograr esto, los usuarios y agentes de dispositivos deberían ser reescritos por completo.

3.2.2 Context Management Framework

El trabajo presentado por Korpipää *et al.* [37] está centrado en el reconocimiento, en tiempo real, de datos provistos por un conjunto de sensores en presencia de incertidumbre, ruido y factores que cambian constantemente. La arquitectura de este sistema se basa en un modelo *blackboard* para la comunicación entre las entidades del framework; en dicho modelo, existe un repositorio centralizado de datos (blackboard) el cual es accedido por entidades (en principio independientes) que poseen conocimiento de un dominio específico. Para organizar la interacción de los componentes existe un coordinador, que se encarga de dictar el orden en que las fuentes de conocimiento accederán al repositorio. Para la representación de la información y posterior toma de decisiones, el framework utiliza un modelo de ontologías [27] y un mecanismo de inferencia.

El framework se encuentra dividido en cuatro grandes módulos (ver figura 3.3):

Context manager: funciona como un servidor central para la comunicación entre las entidades; básicamente cumple el rol del repositorio central en el cual se almacena toda la información relativa al contexto. Los clientes pueden consultar directamente al manejador de contexto o bien suscribirse para ser notificados por cambios en el contexto.

Resource server: son los encargados de conectarse a cualquier tipo de información de contexto y colocar dicha información en el repositorio. En el caso que sea necesario, el manejador de contexto puede post-procesar la información (por medio de servicios de reconocimiento de contexto) para luego diseminarla entre sus clientes.

Context recognition service: se encarga de construir, en base a los datos obtenidos por los sensores, abstracciones de mayor nivel. Este tipo de servicios pueden ser agregados o eliminados en *run-time*; para soportar esto, cada servicio de reconocimiento debe especificar en el momento de la registración cuáles son sus entradas y salidas. De esta forma, la colaboración entre los servicios de reconocimiento y el manejador de contexto permiten convertir datos de bajo nivel en información perteneciente a la ontología modelada.

Security: se encarga del manejo de la seguridad de la información. A diferencia de los módulos anteriores, éste no funciona como cliente del manejador de contexto, sino que es el manejador quien le pide los servicios de seguridad a éste.

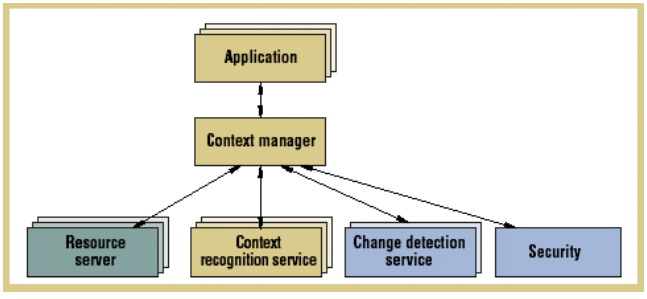

Figura 3.3: Arquitectura del sistema

La forma en la que se estructura el framework permite adjuntar al blackboard el mecanismo de inferencia que resulte más adecuado para el contexto que se desee modelar. La arquitectura basada en un repositorio centralizado

permite distribuir los resultados del proceso de inferencia en forma transparente a los clientes. En sus pruebas los autores han trabajado con un clasificador bayesiano, obteniendo buenos resultados para ambientes acotados. Para lograr el reconocimiento automático de la situación, la máquina es sometida a un proceso de aprendizaje supervisado por el humano, esperando que abstraiga un patrón y se comporte adecuadamente en nuevas situaciones. Un ejemplo sería sensar temperatura, humedad, luz y ruido para determinar si el usuario se encuentra en un ambiente cerrado o al aire libre.

El enfoque presentado por este trabajo brinda una forma interesante de atacar el problema de interpretar el contexto y actuar en base su información. A pesar de ésto, el sistema se encuentra restringido al formalismo dado por un sistema de reglas, cosa que no siempre puede describir la naturaleza evolutiva del contexto. Asimismo, asumiendo que el contexto es estable y puede ser derivado de un conjunto de parámetros, este tipo de sistemas siempre presenta el problema de mediciones ambiguas sobre las cuales no se pueden tomar decisiones. Por último, los autores remarcan que el procesamiento requerido por el repositorio central resulta muy voluminoso y es propenso a generar un tráfico de red importante; por este motivo es deseable que los clientes pre-procesen la información antes de colocarla en el repositorio.

3.2.3 Stick-e Notes

Stick-e Notes [6] es un framework diseñado para manejar distintos tipos de aplicaciones context-aware. A diferencia de otros desarrollos, el foco de este framework no está puesto en la adquisición y distribución de información contextual, sino en brindar soporte para que personas que no son programadores puedan crear servicios context-aware en forma relativamente sencilla. Para lograr ésto, el framework provee mecanismos de alto nivel para especificar qué elementos de contexto afectan a una determinada aplicación, permitiendo disparar acciones por medio de reglas del estilo if-then. El concepto que explota este sistema es el de un stick-e note, el cual modela una nota tipo *post-it* en su versión electrónica. A continuación se presenta un ejemplo de nota, la cual establece que cuando el usuario esté parado entre (38 ° 11,59 ° 20) y (38 ° 15,59 ° 21), orientado entre 150 y 200 grados y en el caso que el mes sea Diciembre, se le debe mostrar información sobre la catedral:

```
<note>
  <required>
    <at> (38.11,59.20)..(38.15,59.21)
    <facing> 150..210
    <during> Diciembre
  <body> Usted se encuentra mirando la catedral.
```

Todo aquello que aparezca dentro del tag `<required>` y antes del tag `<body>` indica cuáles son los campos que deben coincidir para que el texto se muestre. Los tags `<at>`, `<facing>` y `<during>` se utilizan para describir una instancia particular de contexto en que el usuario se puede encontrar. En el tag `<body>` se indica cuál es la acción que el sistema debe realizar en el caso que la descripción del contexto de la regla coincida con el contexto actual del usuario.

Stick-e Notes plantea un enfoque interesante, ya que permite prototipar aplicaciones en forma rápida y brindar la posibilidad a una persona que no es un programador de diseñar los aspectos contextuales de su aplicación. A pesar de ésto, el modelo resulta bastante limitado, ya que el poder expresivo de las reglas se encuentra acotado, al tiempo que no existen mecanismos de inferencia automáticos. Por último, sólo se puede manejar datos discretos y no se adecúa bien al manejo de datos que estén cambiando constantemente.
A pesar que el producto resultante no escala y no puede ser utilizado para programar cualquier tipo de aplicaciones context-aware, es importante tener en cuenta el poder que le brinda al usuario de configurar la forma en que se comportará su aplicación.

3.2.4 Service-Oriented Context-Aware Middleware (SOCAM)

SOCAM [29] es una arquitectura diseñada para construir y prototipar en forma rápida aplicaciones context-aware móviles. Para esto utiliza un modelo basado en ontologías, lo que brinda una rica representación semántica, razonamiento en base al contexto y la posibilidad de compartir conocimiento.

La arquitectura se centra en un modelo basado en ontologías para representar al contexto. Este modelo brinda una representación básica de los objetos y sus relaciones, que puede ser interpretada por una máquina y realizar un análisis formal para producir conocimiento de alto nivel. Una de las ventajas de este enfoque es que la definición de contexto es independiente de la plataforma así como del lenguaje de programación. Para aliviar la carga del dispositivo móvil, la ontología se estructura en dos capas: por un lado se define una capa de bajo nivel, la cual define los conceptos que se aplican a todo tipo de aplicaciones context-aware. La segunda capa, de mayor nivel de abstracción, se divide en un conjunto de sub-dominios disjuntos (por ejemplo, el dominio de la oficina, del hogar, etc), la cual se va ajustando al usuario a medida que su contexto cambia; por ejemplo, cuando el usuario se baja del auto y entra a su oficina, el dominio deja de ser el del automóvil y pasa a ser el de su lugar de trabajo. De esta forma, las estructuras y propiedades de cada dominio son expresadas por medio de ontologías.

Sobre la idea arquitectural presentada se crea una capa middleware llamada SOCAM. Esta capa está compuesta por cinco grandes módulos:

Context-aware Database: provee los mecanismos de persistencia necesarios para manejo de la Knowledge Base.

Context Providers: son los encargados de tomar información de bajo nivel y convertirla para que pueda ser manipulada por la aplicación. Estos proveedores deben registrarse a un servicio que permite que los proveedores sean descubiertos por otros componentes de la arquitectura. Un ejemplo de proveedor sería un receptor de GPS que convierte, mediante un conjunto de reglas, a un predicado del estilo "Me encuentro cerca del supermercado".

Context Interpreter: actúan como proveedores de contexto en el sentido que generan abstracciones de mayor nivel en base a interpretaciones de elementos de contexto de bajo nivel. La diferencia respecto de los anteriores es que usan una KB y un mecanismo de razonamiento. Esta arquitectura permite agregar mecanismos propios de razonamiento así como agregar nuevas reglas a las ya existentes. Asimismo la KB provee de una API para que otros servicios puedan ejecutar consultas, y ABMs de tuplas.

Service Locating Service: permite a los servicios y aplicaciones localizar a los diversos proveedores de contexto, adaptándose dinámicamente a los cambios en el contexto(por ejemplo, un proveedor de contexto podría agregar o quitar sensores físicos).

Context-aware Mobile Services: son básicamente aplicaciones que hacen uso de la información de contexto en diferentes niveles y adaptan su comportamiento a éstos. Los proveedores de contexto son ubicados utilizando el servicio de descubrimiento. Una vez que se obtiene una referencia a un proveedor, se los puede consultar por sus valores o registrarse para escuchar los eventos que éste puede generar. Una de las ventajas que provee SOCAM es que permite ejecutar acciones como resultado de la evaluación de un conjunto de reglas, las que son evaluadas cuando hay un cambio de contexto.

SOCAM presenta una estructura similar al Context Management Framework y se basa en una idea similar al modelar el contexto con ontologías y utilizar mecanismos de razonamiento. En sus publicciones [29, 28] los autores indican resultados en performance muy satisfactorios. A pesar de ésto, el enfoque planteado parece no dar mucha importancia a la integración de los mecanismos de razonamiento con la aplicación en sí misma, cosa que no es nada trivial en este tipo de desarrollos.

3.2.5 Hydrogen

El objetivo de Hydrogen [33] es lograr una arquitectura en capas para el desarrollo de aplicaciones context-aware sobre dispositivos móviles. Para ésto, los autores plantean los siguientes requerimientos:

- La arquitectura debe tener en cuenta el poder de cómputo limitado de los dispositivos móviles.
- Soporte para conexión a mecanismos de sensado remoto.
- La arquitectura debe ser lo suficientemente robusta para soportar conexiones intermitentes.
- Debe soportar el manejo de meta-información, permitiendo indicar la precisión de los datos sensados, privacidad de la información, etc.
- El contexto debe poder ser compartido entre dispositivos.

Una de las principales premisas en el diseño del framework es lograr una clara separación de responsabilidades, permitiendo (idealmente) que el programador de aplicaciones se concentre sólo en la lógica de su dominio. Como puede verse en la figura 3.4 el framework se divide en tres capas:

Adaptor Layer: es responsable de enviar la información que leen los sensores al servidor de contexto (*context server*). Asimismo, esta capa controla el acceso concurrente de varios dispositivos a la misma información de sensado. Por último, esta capa puede enriquecer los datos primitivos con información contextual, para que luego sean procesados en un nivel de abstracción más alto.

Management Layer: se encarga de manejar y proveer, en forma centralizada, toda la información de contexto, lo que implica que todos los dispositivos deben conectarse a ella. El módulo central de esta capa es el servidor de contexto, que es el encargado de administrar a bajo nivel toda la información. El servidor ofrece dos formas de interacción: por medio de consultas directas o por medio de un mecanismo de suscripción, que permite que los dispositivos sean notificados cuando algún aspecto de contexto cambia.

Application Layer: se encarga de implementar la funcionalidad específica de la aplicación context-aware, interactuando con la capa de manejo de contexto. Para no hacer tan pesado el trabajo del programador, el framework provee un conjunto de clases predefinidas.

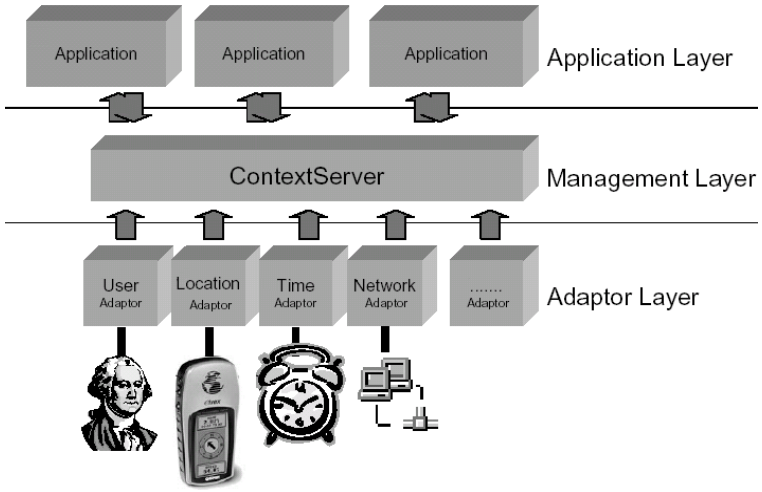

Figura 3.4: Arquitectura en tres capas

3.2.6 Cooltown

Cooltown [9] es una infraestructura diseñada para soportar presencia web (*web presence*) de objetos, personas y lugares. Para ésto, cada elemento del mundo real que se considere relevante, se relaciona con una página web por medio de una URL [45].

En sus publicaciones, los autores plantean una ciudad ideal en el sentido de la computación ubicua[1], donde las personas se encuentran rodeadas de una gran cantidad de dispositivos con capacidad de cómputo; de hecho, la arquitectura se basa en la posibilidad de trabajar con servidores web embebidos en elementos como impresoras o dispositivos para controlar la luz.
A nivel de hardware, la posición del usuario es determinada por medio de *beacons*, que son pequeños dispositivos que emiten constantemente un identificador. La transmisión se realiza en forma inalámbrica por infrarrojo, aprovechando que la mayoría de las PDAs de hoy en día poseen un puerto infrarrojo de lectura.

Para comprender el trabajo y su alcance, se presenta a continuación el ejemplo del museo de Cooltown. Supongamos que un usuario que posee una PDA visita el museo; al pararse cerca de una escultura o pintura recibe la señal de un beacon, indicando la URL con la que se encuentra relacionado el objeto de atención del usuario. Al recibir esta URL, el sistema guarda en un historial la

[1]Del lat. ubi-que, en todas partes.

- Dicho principalmente de Dios: Que está presente a un mismo tiempo en todas partes.
- Dicho de una persona: Que todo lo quiere presenciar y vive en continuo movimiento.

página que se está mostrando en ese momento y carga la página correspondiente a la obra, que puede ser interna al museo o un redireccionamiento a un recurso externo. Con este esquema, a medida que el usuario navega físicamente el espacio, también lo hace virtualmente.
Al llegar al final de su recorrido, el usuario ingresa en la librería del museo, en la cual cada libro posee un beacon para identificarlo y así brindar información adicional. En este caso, el usuario podrá obtener el precio, recomendaciones y críticas que ha recibido el libro, etc. Si el usuario lo desea, puede imprimir una página web por medio de una impresora preparada para servir requerimientos web.

Una de las ventajas de este modelo es que su construcción no requiere de una gran infraestructura:

- El dispositivo que lleva el usuario sólo debe poder ejecutar un web browser levemente modificado.
- El dispositivo debe aceptar entradas por el puerto infrarrojo, cosa que ya es un estándar en las PDAs.
- Los beacons son dispositivos de bajo costo. Asimismo, en el caso que no puedan transmitir una URL completa, con un identificador único bastará, ya que luego la URL puede ser buscada en una tabla.

A pesar de estas ventajas, este desarrollo presenta los siguientes puntos débiles:

- El principal problema que presenta este enfoque es la naturaleza de las conexiones web, ya que no se mantienen en el tiempo y son unidireccionales. Esto implica que el comportamiento es disparado por el cliente, no pudiendo ser actualizado por cambios en el servidor. Asimismo no permite trabajar con sensores de bajo nivel, ni activar servicios en respuesta a los cambios en el contexto.
- Asume que un servidor web puede ser embebido en cualquier dispositivo. A pesar que se han logrado servidores web de menos de 100Kb, pensar en integrar un servidor web dentro de cualquier dispositivo todavía está lejos de ser una realidad.
- Asume la posibilidad de colocar beacons en cualquier dispositivo. A pesar de las ventajas que presentan los beacons, éstos ocupan un volumen y requieren de alimentación para funcionar. Luego, colocar un beacon en objetos como un libro resultaría bastante incómodo.

3.2.7 CASS - Middleware for Mobile Context-Aware Applications

La arquitectura de CASS [21] está basada en una capa middleware que consiste de un servidor central: los clientes envían requerimientos al servidor para que sean procesados en forma remota. Básicamente la arquitectura planteada en este trabajo se divide en tres capas:

Nodos de sensado: proveen información de sensado a la capa middleware. Un nodo de sensado es básicamente una computadora que controla uno o más sensores. Los nodos de sensado pueden ser estáticos o móviles.

Middleware: incluye los mecanismos de interpretación de contexto, motor de ejecución de reglas y manejo de bases de datos.

Cliente: se conecta con la capa middleware para obtener información de contexto de alto nivel. Los dispositivos móviles se conectan al servidor por medio de redes inalámbricas (*wireless*). Es importante notar que en esta arquitectura los clientes no pueden interactuar entre sí; la única forma de comunicación es por medio del servidor.

El servidor CASS provee un mecanismo de acceso a una base de datos que permite almacenar información en forma persistente. En dicha BD, se puede guardar información de contexto histórica, información particular de la aplicación que se esté construyendo, reglas de inferencia y cualquier otro tipo de información que deba ser presentada al usuario. Para manejar la BD se optó por utilizar el lenguaje SQL. Para el manejo del contexto se utiliza un mecanismo de inferencia basado en una KB, el cual utiliza un conjunto de reglas. Cada vez que ocurre un cambio en el contexto se activa el motor de inferencia para buscar aquellas reglas cuyo condicional coincidan con los datos del contexto; una vez encontradas, éstas son evaluadas con el objetivo de llegar a una conclusión. Para ésto, CASS utiliza una técnica llamada *forward chaining*, en la cual los hechos ya conocidos son utilizados para inferir nuevos hechos; una vez establecidos, éstos pueden ser utilizados para inferir a su vez otros nuevos. A modo de ejemplo, para determinar si el usuario se encuentra dentro de un espacio cerrado o al aire libre, en la BD encontraríamos columnas del estilo *humedad, luminosidad, temperatura, resultado*, siendo una tupla posible (alta, baja, frío, aire libre).

En este sistema, dado que todo el procesamiento se realiza en un servidor no existen problemas de memoria ni de velocidad de procesamiento, lo que permite trabajar con grandes bases de datos y mecanismos de inteligencia artificial. Por otro lado, como en todas las arquitecturas centralizadas, el servidor se puede convertir en un cuello de botella.

3.2.8 Sentient Object Model

El modelo utilizado en Sentient Object Model se basa en un sistema descentralizado, donde existen una gran cantidad de mecanismos de sensado disponibles y los dispositivos se encuentran constantemente conectados por medio de redes inalámbricas. Los autores plantean un ambiente cargado de componentes móviles independientes, que estarán constantemente obteniendo información de sensores y modificando el ambiente por medio de actuadores. Asimismo, éstos componentes deben contar con la inteligencia suficiente para actuar en forma autónoma y coordinar acciones con otros componentes. Esta idea de componentes es denominada *sentient objects* [23], que es básicamente un objeto que encapsula a una entidad que puede interactuar con el mundo que lo rodea por medio de sensores y actuadores.

Una de las características principales de este sistema es que se espera que los sentient objects interactúen usando un paradigma de comunicación entre procesos basado en eventos. Gracias a esto, se aseguran un bajo acoplamiento entre los componentes, lo cual facilita la adaptación a nuevos requerimientos. Dentro del este sistema de eventos, los autores los separan en *eventos de software* y *eventos del mundo real*:

Eventos de software: son la principal forma de interacción entre las entidades modeladas, permitiendo realizar una comunicación anónima.

Eventos del mundo real: representan todas aquellas cosas que suceden en el ambiente que impactan en un sentient object. Esto puede ser un cambio en el estado interno del objeto o el resultado de un actuador que haya modificado el ambiente.

Basados en esta primera clasificación, los autores proponen una segunda taxonomía, en la cual los objetos son categorizados dependiendo del tipo de eventos que producen y consumen:

Consumidores del mundo real, productores de software: son objetos que producen eventos lógicos en respuesta a eventos físicos. Su función es básicamente transformar una ocurrencia de un hecho físico en información que pueda ser procesada por otros objetos. A este tipo de objetos se los denomina **sensores**. En forma general, un sensor es un dispositivo que puede responder a un estímulo físico produciendo algún tipo de señal. En este modelo, el sensor encapsula las transformaciones necesarias entre las señales de entrada (por ejemplo, una posición (latitud,longitud)) y los eventos que produce (por ejemplo, “Aula 105”).

Consumidores de software, productores de software: esta clase de objetos consume y produce eventos lógicos, por lo que todo el flujo de información se realizará por medio de la capa middleware basada en eventos. En esta capa es donde justamente vamos a encontrar la lógica de la aplicación y son estos objetos los que proveen las partes sobre las que construiremos nuestro programa. A este tipo de objetos se los denomina **sentient objects**. Dado que todo el procesamiento que hacen estos objetos es lógico, los autores restringen la conectividad entre sentient objects, obligando que éste se encuentre entre uno o más sensores y uno o más actuadores.

Consumidores de software, productores del mundo real: este tipo de objetos son la antítesis de los sensores, ya que se encargan de actuar sobre el medio en base a eventos de software. A estos objetos se los denomina **actuadores** y se define generalmente como cualquier dispositivo capaz de modificar o controlar algún aspecto del medio. Al igual que el caso de los sensores, los actuadores encapsulan la transformación lógico-física.

Consumidores del mundo real, productores del mundo real: Este tipo de objetos son básicamente un subsistema del mundo real, que no tiene conexión por medio de la capa middleware con otros sistemas de software. A estos objetos se los llama **sentient systems**.

Para poder crear una aplicación context-aware, los autores identifican tres grandes componentes **dentro** de un sentient object:

1. Captura de información: un sentient object obtiene información de contexto de uno o más sensores a los que se encuentra lógicamente conectado. Las señales de éstos sensores son utilizadas para lograr una representación del contexto actual.

2. Representación del contexto: los datos de bajo nivel del sensor deben ser interpretados para lograr mayor expresividad. Esas transformaciones suelen ser llevadas a cabo por el sensor, pero también lo puede hacer un sentient object. Este componente de representación permite que otros sentient objects trabajen sobre ese contexto.

3. Mecanismo de inferencia: dado que los sentient objects deben funcionar en forma autónoma, requieren de algún mecanismo para tomar decisiones. Este requerimiento se encuentra modelado en el componente de inferencia, el cual es básicamente un programa que puede realizar razonamientos basado en un conjunto de reglas y una KB, para llegar a un evento resultante. Generalmente, las reglas son capturadas en base a la experiencia de seres humanos que puede ser codificada formalmente.

En la figura 3.5 podemos ver un diagrama con la relación entre los sentient objects, los sensores y actuadores.

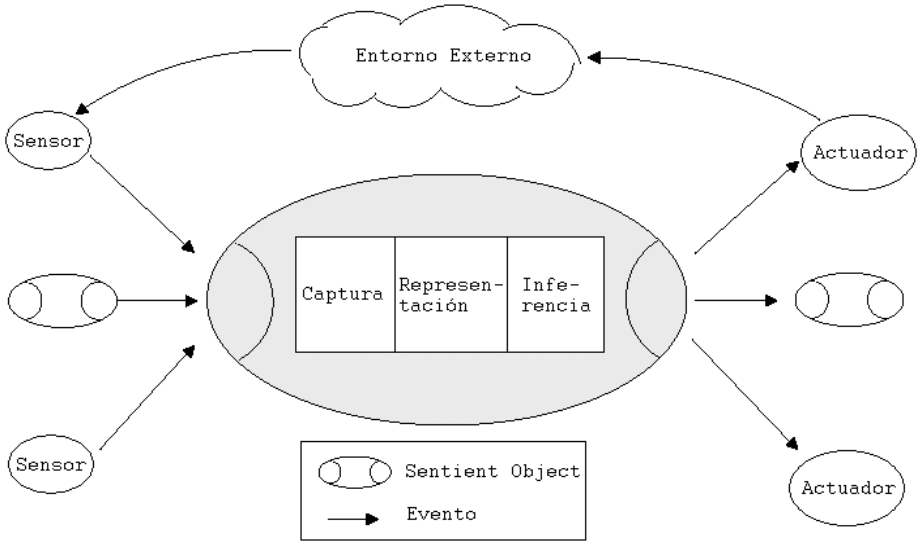

Figura 3.5: Estructura interna de un sentient object y su relación con sensores y actuadores

3.2.9 Context Toolkit

Context Tolkit [13, 14] es un framework context-aware que da un paso hacia el desarrollo de aplicaciones context-aware distribuidas, utilizando una arquitectura peer-to-peer. Dicho framework se basa en tres abstracciones básicas: *widgets*, *interpreters* y *aggregators*.

Context widgets: son componentes que permiten a las aplicaciones acceder a la información de contexto del usuario. Generalmente, dicha información es obtenida del ambiente en que se encuentra inmerso el usuario, ya sea por medio de sensores físicos o lógicos. El objetivo principal de un widget es encapsular la complejidad del sensado en entidades con interfaces bien definidas; los widgets encapsulan su estado y proveen un conjunto de *callbacks* basados en eventos. Gracias a este diseño, la aplicaciones pueden optar por pedir explícitamente la información de sensado o bien registrarse para ser notificadas en el caso que haya nueva información disponible. Asimismo los widgets proveen una facilidad para mantener información histórica de los valores sensados.

Context interpreters: se encargan de convertir en información de alto nivel a los datos de bajo nivel obtenidos por los widgets. La interpretación puede ser una transformación entre dominios (por ejemplo, de una posición dada por un par *(latitud,longitud)* a una dirección) o la combinación de información de varias fuentes para generar nuevas representaciones.

Context aggregators: surgen de la necesidad de tratar como una unidad a un conjunto de datos que están lógicamente relacionados, pero que son adquiridos en forma separada. Notar que al utilizar aggregators, no sólo logramos mover la complejidad fuera de la capa de aplicación, sino que dejamos la información disponible para ser utilizada por una o varias aplicaciones en cualquier momento. Supongamos como ejemplo que la aplicación debe ejecutar algún comportamiento particular en el caso que el usuario se encuentre despierto después de la 1am, habiendo trabajado en su computadora de escritorio más de tres horas seguidas sin haber cenado. Para que la aplicación realice este comportamiento, debe registrarse a varios widgets y realizar un conjunto de consultas complejas para determinar si se cumple la situación. En este tipo de diseños resulta claro que, mezclar la detección de la situación con el comportamiento de la aplicación hará que sea difícil de mantener y adaptar ante un cambio de requerimientos. Si podemos separar esta responsabilidad de la aplicación, ambos podrán evolucionar independientemente. Desde el punto de vista de la aplicación, un aggregator es como un composite [25] de widgets; al igual que éstos permiten registrarse por medio de callbacks y guardar información histórica.

Services: los componentes mencionados hasta el momento (*widgets*, *interpreters* y *aggregators*) se encargan de adquirir y diseminar la información de contexto. Dado que el objetivo de las aplicaciones context-aware es disparar comportamiento basado en esta información, es necesario contar con algún mecanismo que nos permita ejecutar acciones. Los servicios (*services*) son los encargados de llevar adelante dichas acciones. Los servicios pueden ser vistos como la contraparte de los widgets: mientras que los widgets se encargan de tomar información de los sensores para brindársela a la aplicación, los servicios se encargan de controlar o modificar al ambiente por medio de actuadores en base a lo que dicte la aplicación.

Discoverers: son los encargados de llevar cuenta de las capacidades del framework en todo momento; esto incluye saber que widgets, interpreters, aggregators y services se encuentran disponibles. Cuándo cualquiera de estos componentes comienzan su ejecución, se debe registrar con el *discoverer* indicando sus propiedades (por ejemplo, un widget debe indicar que tipo de contexto puede proveer). En el caso que alguno de estos componentes falle, es responsabilidad del discoverer detectarlo.

El toolkit presenta una serie de ventajas importantes en lo que a modelado y separación de responsabilidades se refiere. Se basa en un modelo de objetos con claras taxonomías, lo que ayuda al desarrollo de las aplicaciones. Asimismo presenta un avance importante en la descentralización de la arquitectura. A pesar de sus claras ventajas respecto de otro tipo de arquitecturas o frameworks, el Context Toolkit no llega a lograr un manejo totalmente distribuido, ya que requiere que el discoverer se encuentre centralizado. Asimismo, se queda a mitad de camino en el modelado con objetos, ya que sigue estructurando el manejo del contexto en términos de datos sobre los que el framework debe actuar.

3.3 Conclusiones

A lo largo del capítulo se han presentados diversas arquitecturas y frameworks para la construcción de aplicaciones context-aware. Para esto se han planteado diversos enfoques, entre los que encontramos arquitecturas centralizadas, capas middleware, sistemas expertos, sistemas distribuidos y modelos basados en la web. A pesar de sus diferencias, encontramos algunos puntos en común:

- Una aplicación context-aware debe manejar un conjunto de aspectos ortogonales que evolucionan en forma independiente con el tiempo; por lo tanto, si el objetivo es lograr una arquitectura escalable, debe haber una clara separación de responsabilidades.
- Uno de los puntos claves para lograr esta separación es independizar la adquisición de datos de sensores de la aplicación.
- El sistema debe permitir que los aspectos de contexto varíen dinámicamente mientras la aplicación se esta ejecutando.
- Sería deseable que el sistema provea facilidades para inferir aspectos de contexto de alto nivel en base a datos de sensado.

A pesar que muchos de estas características se resuelven por los sistemas presentados, en general no se presta atención a la relación que debe existir entre la arquitectura y la aplicación (siendo probablemente las excepciones el Context Toolkit y Hydrogen). Por este motivo, uno de los puntos en los que se enfoca este trabajo de grado es lograr que el desarrollador de aplicaciones se puede concentrar en su modelo de dominio, intentando minimizar el impacto que tenga en su sistema los aspectos de contexto.

Capítulo 4

Diseño de aplicaciones location-aware

How do you know I'm mad? -said Alice.
You must be, -said the Cat- or you wouldn't have come here.

Alice's Adventures in Wonderland
Lewis Carroll

En este capítulo se dará una descripción general de la arquitectura desarrollada a lo largo del trabajo. Para lograr una mejor comprensión de ésta, a lo largo del capítulo se aplicará la arquitectura al trabajo de aplicaciones location-aware. En el siguiente capítulo se mostrará como esta arquitectura puede generalizarse para trabajar con aplicaciones context-aware.

Al comienzo del capítulo se definirá el alcance del trabajo, indicando qué dificultades de las aplicaciones context-aware atacará la arquitectura. A continuación se presentará un nuevo modelo de contexto, el cual será particularizado para el trabajo con aplicaciones location-aware. Una vez realizado ésto, se describirán las capas que constituyen la arquitectura, indicando las abstracciones más relevantes y su comportamiento asociado.

4.1 Alcance del trabajo

Como vimos en la sección 2.6, existe un conjunto importante de problemas asociados a las aplicaciones context-aware, tanto los que son propios de la disciplina, como los que pertenecen a otros campos de la informática. Dado que el objetivo del presente trabajo de grado es desarrollar una arquitectura flexible para la construcción de aplicaciones context-aware, sólo se tratarán aquellos problemas que resulten relevantes a esta tarea.

A continuación se enumeran los temas en los que el presente trabajo hará hincapié:

1. Lograr una clara separación de responsabilidades, tal que los aspectos ortogonales de la aplicación puedan evolucionar en forma independiente.

2. Brindar un modelo de contexto dinámico, que no sólo pueda ser fácilmente adaptado para cada aplicación, sino que pueda variar a lo largo de la vida de la aplicación.

3. Definir un esquema flexible que permita convertir los valores de bajo nivel obtenidos por los sensores en objetos con mayor valor semántico.

4. Brindar un mecanismo de distribución transparente, tal que el desarrollador no tenga que pensar en cómo distribuir sus objetos ni donde residirán físicamente. Asimismo, sería deseable que el desarrollador pueda realizar ajustes finos respecto de la distribución, para así alcanzar los niveles de performance requeridos por la aplicación.

En contrapartida, este trabajo no abarcará los siguientes problemas:

1. *Interfaces gráficas y HCI.* El área de las interfaces gráficas en dispositivos con pantallas pequeñas es un tema de investigación en sí mismo. Dada la envergadura y complejidad de éste, se encuentra fuera del alcance del presente trabajo. Por este motivo, para el desarrollo del framework se utilizó una interfaz sencilla que permite indicar los servicios disponibles, con la posibilidad de activarlos en una sub-ventana embebida.

2. *Tolerancia a fallas.* Tanto para el manejo de las comunicaciones como para la adquisición de datos por medio de los sensores se asume una respuesta perfecta. Esto implica que no se tendrá en cuenta cosas como fallas en la red, conexiones intermitentes o pérdida de señal de sensor.

4.2 Soporte para el framework desarrollado

Como se planteó en la sección anterior, el sistema propuesto debe proveer un esquema de distribución no intrusivo, así como un mecanismo para tomar los los valores capturados por los sensores. Asimismo, dado que se ha tomado un enfoque de objetos puros, sería deseable trabajar en un ambiente que maneje en forma homogénea los conceptos de la programación orientada a objetos (lo cual deja fuera de la elección a cualquier lenguaje híbrido del estilo de Java o C++). Por estos motivos se optó por implementar al framework (denominado *Balloon*) utilizando el ambiente Smalltalk, particularmente la implementación provista por Cincom (VisualWorks Smalltalk).

En lo que respecta a la distribución de objetos, dado que Smalltalk mantiene en todo momento los conceptos del paradigma de objetos en forma pura (*todo* es un objeto y los objetos sólo se comunican enviándose mensajes), capacidades como la de *reflection* o modificaciones al sistema en *run-time*, son naturalmente soportadas por el ambiente [26]. Esto permite que existan frameworks de distribución y persistencia no intrusivos, lo que permite desarrollar el modelo de aplicación independientemente de como luego éste será distribuido o persistido en disco. Para este trabajo en particular la distribución se realizó por medio del framework Opentalk, que permite conectar un conjunto de imágenes Smalltalk por medio de brokers locales y distribuir transparentemente sus objetos, de forma que puedan enviarse mensajes sin importar en que máquinas residen físicamente los objetos (una explicación más detallada puede encontrarse en el Apéndice B).

El segundo requerimiento importante para desarrollar el framework es contar con una forma de interactuar con sensores externos. Generalmente todos los ambientes Smalltalk proveen alguna facilidad para realizar conexiones con software de bajo nivel. En el caso de VisualWorks, existe una extensión al ambiente (DLLCConnect) que permite realizar conexiones a librerías externas (escritas, por ejemplo, en C) y así interactuar con las APIs que proveen los fabricantes de dispositivos de sensado.

4.3 Modelo de contexto

Como hemos visto en el capítulo 2 (más específicamente en las secciones 2.7 y 2.8), la mayoría de los enfoques toman una postura positivista para modelar el contexto. Bajo esta postura, la idea es detectar lo antes posible en la etapa de diseño qué información del contexto es relevante, para luego codificarla en una computadora. Una vez realizado ésto, se establece un mecanismo de adquisición de datos, para luego codificar un programa que realice algún procesamiento sobre éstos.

Una excepción a esta forma de manejar el contexto está dada por el trabajo de Coutaz y Rey [11], en el cual se hace hincapié en que todo contexto se encuentra dentro de una **situación** en la que participa un usuario. A pesar que esta visión representa *conceptualmente* un avance respecto de otros modelos, la implementación se resuelve en forma similar a otros enfoques: una situación se representa con una n-upla de valores sensados, sobre la cual se ejecutará alguna función de transformación.

Al desarrollar la arquitectura presentada, se buscó un modelo de contexto que se encuentre más cercano a la visión fenomenológica que a la positivista [19]. Uno de los problemas que surgen al intentar esto es la incompatibilidad que existe entre la visión fenomenológica y la necesidad de representar al contexto

en una computadora, ya que esto último implica algún tipo de formalismo. Por este motivo se optó por buscar un modelo de contexto que tuviera las siguientes propiedades:

1. *El contexto no es un elemento pasivo.* El contexto debe tomar un rol activo en el sistema y no ser una mera colección de datos.

2. *El contexto no puede estar acotado.* Se debe evitar cualquier modelo que impida la evolución del contexto en formas que no hayan sido previstas en la etapa de diseño.

3. *Los aspectos de contexto pueden variar dinámicamente.* Esto significa que en un determinado momento, un aspecto de contexto es relevante y más adelante puede que deje de serlo. A modo de ejemplo, es posible que al estar caminando por la calle, los aspectos de contexto relevantes del usuario sean la posición y la hora del día. Luego, al ingresar a su lugar de trabajo, los aspectos relevantes pasan a ser la actividad que está realizando y las personas con las que se encuentra.

4. *El contexto es independiente de los mecanismos de sensado.* A pesar que la mayor parte de la información de contexto se obtiene por medio de sensores, el modelo debe ser independiente de los dispositivos utilizados. A modo de ejemplo, si un aspecto del contexto modela la posición del usuario, no debería afectarle que ésta se obtenga por medio de un GPS o por un sistema de beacons infrarrojos.

En conclusión, *el modelo de contexto debe ser inherentemente dinámico y permitir la evolución en forma natural.* Con este objetivo, en lugar de pensar al contexto como un todo, el modelo propuesto en este trabajo lo divide en distintos *aspectos de contexto*, cada uno responsable de implementar un comportamiento particular. Estos aspectos de contexto se aplican sobre algún sujeto, que en la mayoría de los casos resulta ser el usuario.

En este modelo, el contexto del usuario no es visto como una entidad formal ni acotada, sino que es *la unión de un conjunto de aspectos de contexto, pudiendo estos actuar en conjunto o ser desarrollados independientemente.* A modo de ejemplo, supongamos que deseamos hacer aplicaciones que dependen de la posición del usuario; para lograr esto se debe enriquecer la representación del usuario con un aspecto de contexto que modele su posición. Luego, en base a los cambios en la posición del usuario, el sistema deberá reaccionar para presentar los servicios adecuados. Como se verá más adelante, diversos aspectos pueden ser agregados gradualmente para enriquecer el modelo de contexto de la aplicación; de hecho, los aspectos tenidos en cuenta pueden variar dinámicamente, ajustándose así a las propiedades mencionadas anteriormente.

4.4 Un escenario representativo

Muchos de los sistemas context-aware que se van a desarrollar en el futuro no van a ser construidos desde cero, sino que partirán de sistemas ya construidos a los que hay que agregarle características context-aware. A modo de ejemplo, supongamos que queremos aumentar las capacidades del sistema de una universidad para que provea información en base al contexto del usuario, asumiendo que éste posee algún dispositivo de computación móvil (como por ejemplo una PDA). Para ésto partimos de un sistema que ya ha sido diseñado y que se encarga de la inscripción de los estudiantes a las cátedras, del manejo de aulas y horarios, de la organización del material de cada cátedra, etc. Este sistema puede ser utilizado (por ejemplo, por medio de una interfaz web) tanto por profesores como por estudiantes. Un diagrama de clases reducido de dicho sistema se muestra en la figura 4.1.

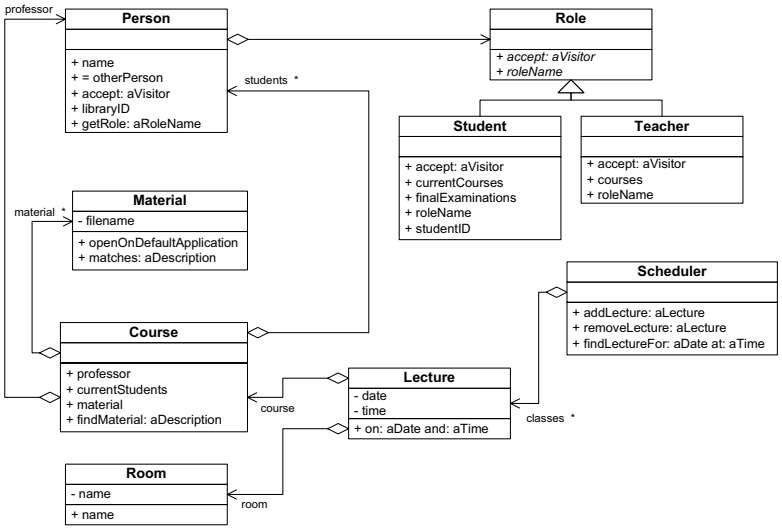

Figura 4.1: Diagrama de clases del modelo de una universidad

Al agregarle comportamiento context-aware al sistema, cuando un estudiante ingresa al edificio se le muestran automáticamente los servicios correspondientes, como por ejemplo un mapa de aulas o un recordatorio del aula y el horario de su próxima cursada. Al ingresar a un aula se le suman nuevos servicios a los que ya tenía, como por ejemplo uno que presente el material de la clase que se está dictando en ese momento. Luego, al salir del aula y pasar cerca del comedor en el horario del mediodía, el usuario obtiene el menú del día. Asimismo uno esperaría que estudiantes y profesores tengan acceso a distintos servicios: en el caso de estar tomando un parcial es lógico que el profesor pueda acceder a las

respuestas del examen, al tiempo que el estudiante sólo debería ver el enunciado.

El objetivo de este capítulo es presentar una arquitectura que pueda resolver uno de los problemas más desafiantes en el diseño de aplicaciones context-aware: cómo integrar transparentemente, a una aplicación existente, comportamiento que depende del contexto. Por cuestiones de simplicidad, para explicar la arquitectura sólo se tomará en cuenta el aspecto de ubicación del usuario; en el capítulo siguiente se mostrará como extrapolar estas ideas a cualquier aspecto de contexto, así como algunos detalles más específicos del framework.

4.5 Visión general de la arquitectura

Los sistemas context-aware se caracterizan por sus cambios en los requerimientos y por su constante evolución. Para manejar este tipo de sistemas se diseñó una arquitectura que combina un diseño basado en capas [8] con un fuerte uso de dependencias en el estilo del pattern Observer [25]. En la figura 4.2 se muestra un diagrama UML con la estructura de paquetes y sus relaciones. Es importante notar que la separación en capas no se hace en una sola dimensión (como es común en este tipo de sistemas), sino que se plantean dos dimensiones ortogonales. En una dimensión se extiende a la aplicación con los aspectos de contexto, mientras que en la otra se obtiene la información de dispositivos de hardware. El punto de convergencia de estas dos dimensiones el aspecto de contexto, el cual interactúa con la capa de servicios.

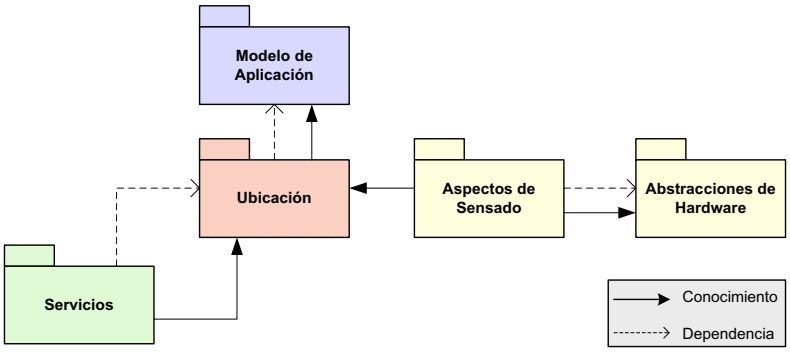

Figura 4.2: Arquitectura de una aplicación location-aware

En la figura 4.3 se muestra la interacción entre los paquetes presentados en la figura 4.2. Éstas son:

Servicios-Ubicación. La capa de servicios observa a los aspectos de contexto (en este caso la ubicación del usuario). En el momento que algún aspecto de contexto cambie, por medio del mecanismo de dependencias, la capa de servicios se enterará y tomará las acciones adecuadas (como disparar un servicio o presentar información).

Ubicación-Modelo de Aplicación. A pesar que en algunos casos los aspectos de contexto son independientes del modelo, en la mayoría de las ocasiones esto no sucede. En nuestro ejemplo, la ubicación se refiere a la posición de un usuario, el cual está representado en el modelo como una persona, que puede cumplir los roles de estudiante o profesor. En estos casos, los aspectos de contexto *extienden* al modelo agregándole comportamiento.

Ubicación-Aspectos de Sensado-Abstraccioens de Hardware. En la capa de abstracciones de hardware se modelan los sensores. Cuando alguno de éstos recibe valor nuevo, notifica por medio del mecanismo de dependencias a todos aquellos aspectos de sensado interesados. Luego, éstos se encargan de transformar los valores obtenidos por los sensores en objetos con mayor contenido semántico, para luego indicarle al aspecto de contexto que se ha producido un cambio. En nuestro ejemplo, el objeto que modela al puerto infrarrojo recibiría el identificador del aula en la que ingresó el usuario; como respuesta a ésto el aspecto de sensado convertiría este identificador en un objeto que represente un aula y le indicaría al aspecto de ubicación del usuario que ahora se encuentra en ese aula.

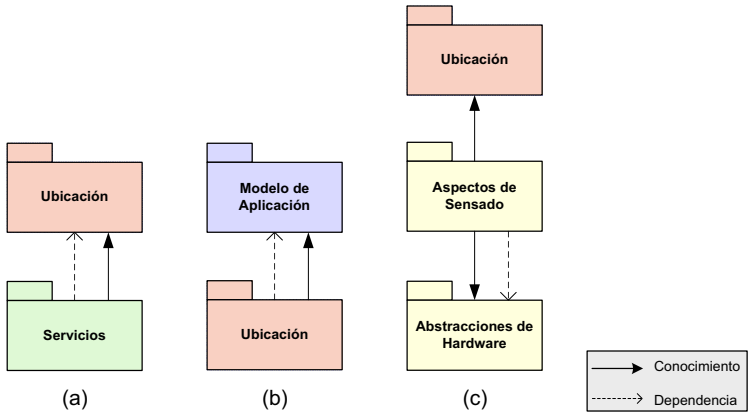

Figura 4.3: Interacciones entre paquetes

Gracias a esta separación podemos lograr un bajo acoplamiento entre los distintos modelos (de aplicación, de contexto y de sensado), evitando que los cambios en uno se propaguen al resto.

4.5.1 Capas de la arquitectura

En el **Modelo de Aplicación** se especifican aquellas clases que tienen el comportamiento "estándar" del modelo; esto implica que éstas clases no tienen noción de ningún aspecto de contexto, como puede ser la ubicación. En nuestro ejemplo de la universidad encontraríamos clases como `Course`, `Student`, `Material`, etc.

En la capa de **Ubicación** se encuentran los objetos que agregan la noción de ubicación a los objetos del modelo, lo que permite al sistema reaccionar en base a la posición del usuario. A modo de ejemplo, para decir que el usuario se encuentra en el aula A primero debemos crear una abstracción en términos de la ubicación para el aula A. Notar que el modelo de aplicación ya contiene una clase `Room` que modela el aula, pero ésta no posee ningún comportamiento relacionado con su ubicación física. Luego, lo que hacemos en esta capa es agregar ese aspecto que no está modelado en la capa de aplicación. De esta forma logramos una clara separación entre la lógica de aplicación y los aspectos de contexto que queremos incorporar a nuestro sistema, permitiendo que evolucionen independientemente.

Es importante notar que esta capa no sólo agrega comportamiento a objetos existentes en el modelo de aplicación, sino que también modela objetos que son puramente de la capa de ubicación y que no tienen una contraparte en el modelo de aplicación. A modo de ejemplo, en el caso del sistema de la universidad, puede que nos interese modelar un mapa en el cual aparecen las aulas y los pasillos que las conectan. A pesar que las aulas tienen una representación en el sistema de la universidad, los pasillos resultan irrelevantes para este modelo, por lo que la clase `Corridor` no tendrá un representante en el modelo de aplicación. En la figura 4.4 se muestra un ejemplo de la capa de ubicación y su relación con el modelo de aplicación. Como veremos más adelante, utilizando un concepto similar, la capa de ubicación puede ser generalizada para trabajar con otros aspectos de contexto como tiempo, temperatura, actividad, etc.

Las clases de la capa de ubicación mencionadas hasta el momento (como `Corridor` o `Room`) se utilizan para generar un mapa físico de alto nivel del dominio de aplicación. Sobre dicho mapa se deben realizar operaciones relativas a la ubicación, como saber si dos aulas son adyacentes o si un aula se encuentra dentro de un edificio. Para ésto se trabaja con distintos modelos de posicionamiento de bajo nivel, los cuales se desacoplan por medio del tipo `Location`. Con esta distribución, la capa de ubicación se compone de un modelo de posicionamiento de bajo nivel y de un mapa de alto nivel semántico.

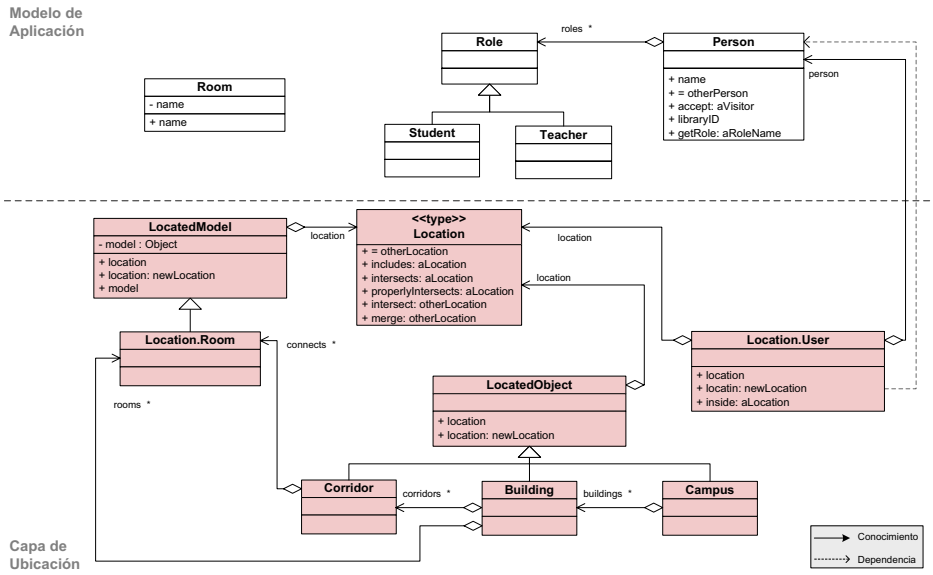

Figura 4.4: Relación entre el modelo de aplicación y la capa de ubicación

Gracias al uso de tipos, los modelos de posicionamiento de bajo nivel pueden ser intercambiados sin impactar en el diseño del mapa. De hecho, al ser independientes de la aplicación, pueden ser reutilizados en diversos desarrollos. Ejemplo de esto son los modelos geométricos y simbólicos [40], los cuales son implementados como paquetes externos e incorporados a la capa de ubicación según sea necesario. Teniendo como base un modelo de posicionamiento, se construye sobre éste un mapa que agrega comportamiento geográfico a los objetos de la aplicación.

En la capa de **Servicios** se encuentra modelada la funcionalidad context-aware propiamente dicha. Para ésto se utiliza la noción de servicio (`UserService`), que es un objeto que puede ser asociado con determinadas áreas geográficas. En esta capa también se encuentra un objeto que maneja los servicios disponibles para el usuario (clase `Service.User`); en particular, esta clase maneja la suscripción y activación de los servicios, así como la coordinación de los distintos aspectos de contexto del usuario[1]. En la figura 4.5 se muestra la relación entre los elementos de las capas de servicios y de ubicación. Es importante notar la forma en la que indicamos que un servicio está disponible en una determinada área geográfica: dado que no queremos cargar a la capa de ubicación con cuestiones relacionadas a los servicios, la relación lógica entre los servicios y las áreas geográficas se expresa por medio de las áreas de servicio (`ServiceArea`). Un área

[1]La clase `User` en este modelo cumple un rol similar al componente de contexto descripto en [18].

de servicio sabe que región ocupa por medio de la relación de conocimiento que indica su ubicación (*location*). La principal responsabilidad de esta clase es saber qué servicios se encuentran disponibles en un determinado momento, de forma tal que cuando un usuario se encuentre físicamente dentro del área, se le presenten los servicios disponibles. Desde el punto de vista del diseño, un área de servicio actúa como un Mediador [25] entre un conjunto de servicios y el área geográfica en la que se encuentran disponibles.

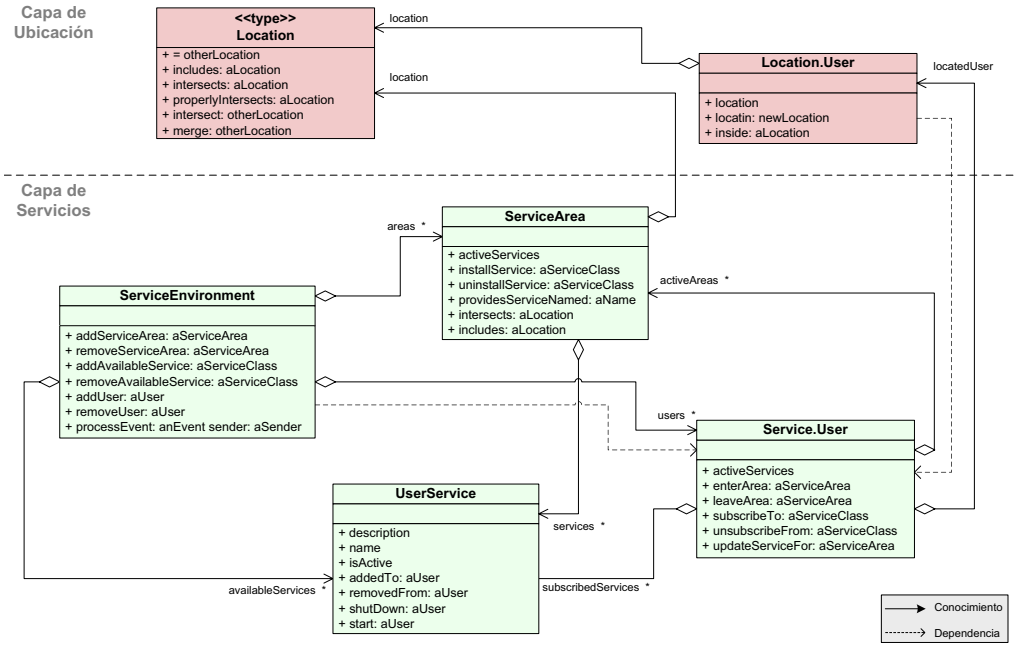

Figura 4.5: Relación entre la capa de servicios y la capa de ubicación

En la capa de **Abstracciones de Hardware** se encuentra la representación en objetos de los dispositivos de hardware que se utilizan para obtener información del entorno. Ejemplos de este tipo de dispositivos son `IButton`, `IPort`, `GPSSensor`, etc. Este tipo de abstracciones cumplen un rol similar a los widgets mencionados por Dey [18].

La capa de **Aspectos de Sensado** contiene un conjunto de aspectos encargados de realizar la conexión entre los datos obtenidos de los sensores (por medio de las abstracciones de hardware) y los aspectos del contextos modelados (en este caso el de ubicación). El desacople generado por esta capa (que puede ser considerado como una mejora de los interpreters de Dey [18]) garantiza que la capa de ubicación y la de sensado puede evolucionar independientemente. Por ejemplo, para la ubicación en espacios cerrados podríamos utilizar en principio un modelo de ubicación simbólico [40] y un mecanismo de sensado por medio

de beacons infrarrojos. Más adelante, podríamos optar por dejar de lado los beacons infrarrojos y utilizar un mecanismo de posicionamiento por radiofrecuencia, lo cual sólo impactaría en la capa de abstracciones de hardware y en la configuración de los aspectos de sensado, pero no sería necesario modificar el modelo de ubicación.

4.5.2 Mecanismos de comunicación

Las relaciones entre los objetos de las diversas capas utilizan dos estilos de comunicación:

- Relaciones de conocimiento directo, como puede ser la que tiene la clase `Room` en la capa de ubicación con la representación del aula en el modelo.
- Relaciones de dependencia, en el estilo del pattern Observer [25] que permite notificar a los objetos interesados de un cambio en el observado.

Ambos tipos de relaciones pueden verse en la figura 4.6, que muestra un refinamiento de la arquitectura presentada en la figura 4.2.

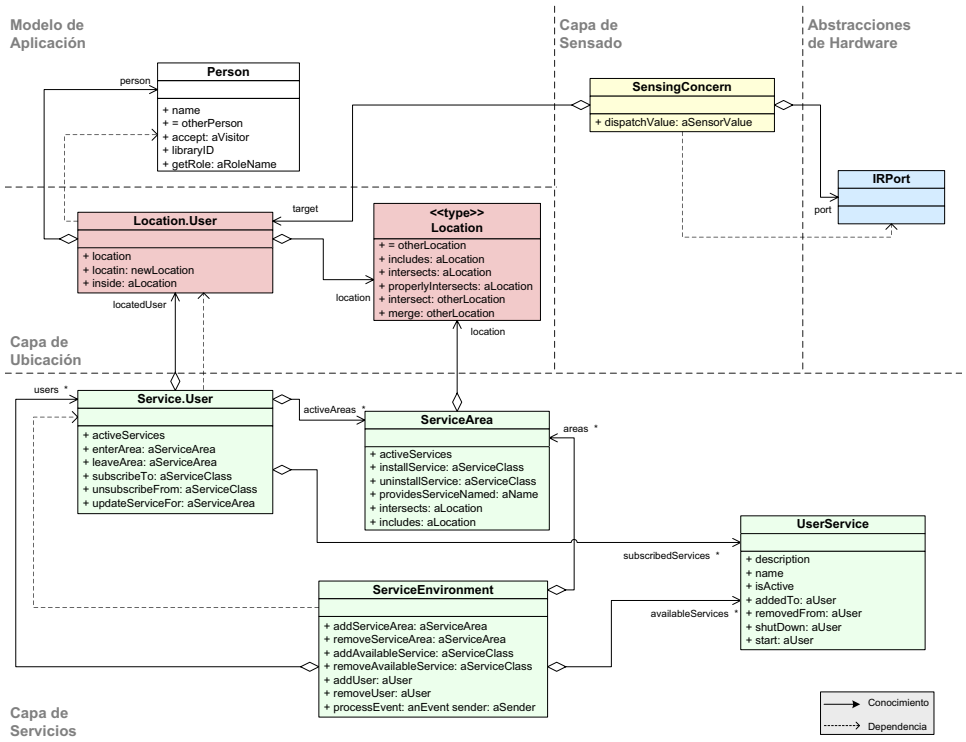

Figura 4.6: Principales clases de la arquitectura

En el momento que la capa de abstracciones de hardware detecta un cambio en el contexto (por medio de un sensor), notifica (utilizando el mecanismo de dependencias) a los aspectos de sensado que se han registrado como dependientes. Éstos, al recibir la notificación, convierten la información de bajo nivel de los sensores en objetos del dominio del aspecto que están adaptando y le envían al aspecto el mensaje con el que han sido configurados (por ejemplo, `#location:`). Al recibir este mensaje, un `Location.User` actualiza su posición e indica (nuevamente por el mecanismo de dependencias) que ha cambiado. Al recibir la notificación del cambio del `Location.User`, el `Service.User` también dispara un cambio, el cual es capturado por el `ServiceEnvironment` el cual se encarga de actualizar los servicios del usuario. Esta cadena de cambios es la que permite que, ante la aparición de un nuevo valor en un sensor, el sistema reaccione redefiniendo los servicios disponibles al usuario.

4.6 Capa de Ubicación

Como se indicó en la sección 4.5.1, la capa de ubicación se compone de dos modelos, uno genérico (modelo de posicionamiento) y otro dependiente de la aplicación (modelo de ubicación). El modelo de posicionamiento brinda un conjunto de abstracciones básicas para representar la posición de lugares y objetos. Su comportamiento se encuentra determinado por el tipo `Location`, lo que significa que cualquier objeto que implemente dicho tipo puede ser utilizado como modelo de posicionamiento. Sobre este modelo se construye un mapa de alto nivel, el cual extiende las abstracciones del modelo de aplicación con comportamiento geográfico. Los roles y responsabilidades de ambos modelos se detallan en las siguientes secciones.

4.6.1 Modelo de posicionamiento

Los sistemas *location-aware* necesitan un modelo que pueda representar la ubicación de objetos móviles y estáticos. Para resolver este problema, se utilizan dos grandes enfoques: representar al espacio como un sistema de coordenadas *n-dimensional* o bien como un conjunto de símbolos y sus relaciones. Al primero de estos modelos se lo denomina *geométrico* y al segundo *simbólico* (el lector interesado encontrará una explicación más detallada Apéndice A).

Con el objetivo de lograr una mayor flexibilidad en la implementación, el concepto de ubicación ha sido modelado por medio de un tipo [2] y no como una clase abstracta; de esta forma cualquier clase que implemente el tipo `Location` (y

[2]Idealmente un tipo no sólo indica el conjunto de mensajes a los que debe responder un objeto, sino también la semántica de cada uno de ellos. Dada la complejidad de especificar semántica en diagramas UML, se especificarán sólo nombres de los mensajes al referirnos al tipo `Location`.

respete su semántica) puede ser utilizada como un modelo de posicionamiento. El tipo en cuestión se muestra en la figura 4.7.

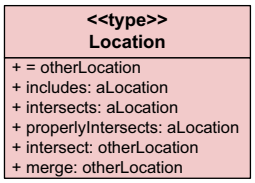

Figura 4.7: Tipo `Location`

Al trabajar con distintos modelos de posicionamiento y formas de sensado, muchas veces los desarrolladores se ven tentados a asociar unos con otros. Típicamente se asocia al modelo geométrico basado en pares ordenados *(latitud,longitud)* con un mecanismo de sensado GPS y a los modelos simbólicos con algún tipo de beacons (infrarrojos, RFID, etc.). A pesar que a primera vista esto pueda parecer natural, no sólo presenta un problema conceptual (ya que se confunden dos dominios independientes) sino que hace que los cambios en uno de los aspectos del sistema impacte directamente en el otro. Por este motivo, en la arquitectura planteada en el presente trabajo no existe ninguna relación entre modelo de posicionamiento y los mecanismos de sensado: se podría utilizar tanto GPS como beacons infrarrojos con cualquiera de los dos tipos de modelos. Para lograr esta independencia se utiliza la capa de sensado, la cual captura la información obtenida por los sensores, realiza un proceso de abstracción (convirtiendo datos en objetos) para luego enviarla al modelo de ubicación.

Hasta el momento todas las pruebas se han realizado en ambientes cerrados con pocas habitaciones, por lo que resultó más adecuado utilizar un modelo de posicionamiento simbólico; en particular se trabajó con el modelo jerárquico basado en dominios [40] (ver Apéndice A). A pesar de ésto, para probar la flexibilidad del sistema, se implementó también un prototipo de modelo geográfico, aunque sin el manejo de conversiones entre sistemas de coordenadas. Por último, uno de los trabajos a futuro es desarrollar un sistema híbrido que permita trabajar tanto con modelos geométricos como con simbólicos, encargándose de realizar las conversiones automáticamente.

4.6.2 Modelo de ubicación en objetos

Como se indicó anteriormente, el modelo de ubicación genera un mapa de alto nivel utilizando un modelo de posicionamiento en base a la especificación del tipo `Location`. Este mapa se basa en dos abstracciones fundamentales: las clases `LocatedModel` y `LocatedObject`. La primera clase extiende a un objeto del modelo de aplicación con comportamiento relacionado a su posición. La segunda

clase representa a un objeto que no tiene una representación en la capa de aplicación, pero que resulta importante a la hora de construir el mapa. En nuestro ejemplo del sistema de la universidad, un aula está representada en la capa de aplicación por una instancia de la clase `Room`, la cual es extendida en la capa de ubicación con una instancia de `Location.Room`. Dado que `Location.Room` representa en el modelo de ubicación, a un objeto del modelo de aplicación, ésta debe ser subclase de `LocatedModel`. Es importante notar que `Location.Room` no sólo hereda el comportamiento de `LocatedModel`, sino que agrega el comportamiento geográfico necesario para implementar los servicios de la aplicación. A modo de ejemplo, si queremos modelar un mapa que nos indique el camino entre dos aulas, sería necesario saber la disposición de las aulas entre sí y los pasillos que las conectan, así como el edificio que contiene a cada aula. En este caso, los pasillos y los edificios son conceptos requeridos en el modelo de ubicación que no tienen representación en el modelo de aplicación. Esto implica que clases como `Corridor` o `Building` serán subclases de `LocatedObject` y no de `LocatedModel`. Un ejemplo de esta situación puede verse en la figura 4.8; notar que la variable de instancia *`model`* de `LocatedModel`, en la clase `Location.Room` hará referencia a una instancia de la clase `Room`.

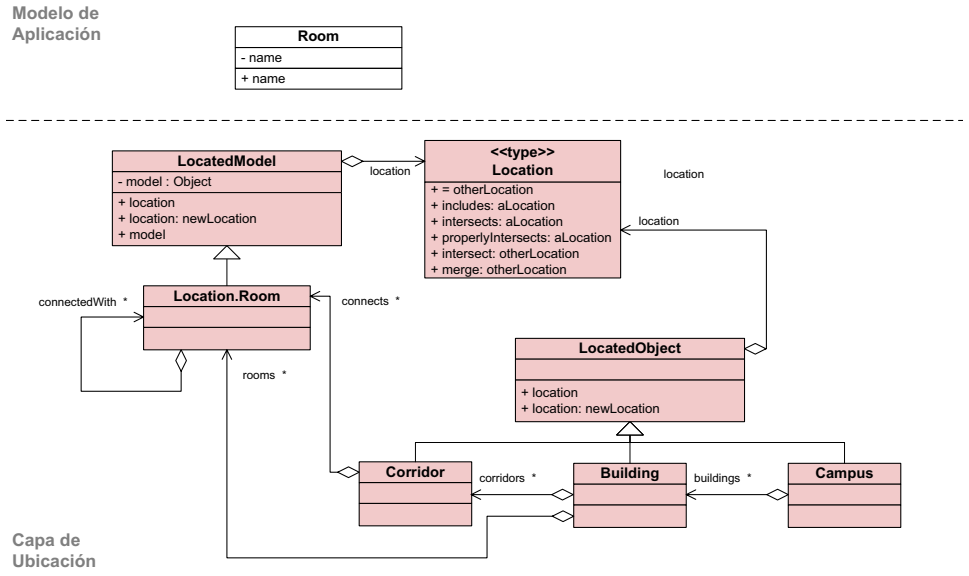

Figura 4.8: Modelo de ubicación

El comportamiento geográfico de los objetos de la capa de ubicación puede ser resuelto, en la mayoría de los casos, utilizando el modelo de posicionamiento subyacente. Por ejemplo, un aula se encuentra dentro de un edificio, si su colaborador *`location`* está incluído en el colaborador *`location`* del edificio (o sea,

si `aBuilding location includes: aRoom location` retorna `true`). A pesar de ésto, en algunos casos el comportamiento requerido no puede ser resuelto de esta manera. A modo de ejemplo, supongamos que queremos saber si dos aulas se conectan por medio de una abertura. Esto no puede ser determinado en base al mecanismo de posicionamiento, ya que no hay forma de representar la noción de *conexión por medio de una abertura* en un modelo de posicionamiento. Luego, esto debe ser modelado explícitamente en el mapa de alto nivel, estableciendo una relación entre un aula y aquellas a las que se encuentra conectada por medio de una abertura. De esto se desprende que los dos modelos son complementarios; el de posicionamiento brinda un conjunto de mensajes predefinido y no está atado a ningún dominio en particular, por lo que puede ser reutilizado en diversos desarrollos. Como consecuencia de ésto, se requiere de un modelo semánticamente más rico para representar las relaciones entre objetos físicos que no pueden resolverse mediante operaciones básicas como la inclusión o adyacencia.

4.7 Abstracciones de Hardware

En la capa de abstracciones de hardware se encuentran aquellas clases que representan los dispositivos de hardware utilizados para la adquisición de datos. Ejemplos de estas clases pueden ser `IRPort`, que representa el puerto infrarrojo de una PDA o `GPSReceiver`, que modela un receptor de GPS.

En esta capa se trabaja con un nivel muy bajo de abstracción, teniendo que interactuar generalmente con drivers y APIs provistas por los fabricantes de dispositivos. A nivel de comportamiento, las clases que representan a los sensores no realizan ningún proceso de abstracción, sino que se limitan a tomar los datos externos y hacerlos disponibles al resto del sistema en forma de objetos. Por ejemplo, en un sistema de reparto de encomiendas que utiliza un mecanismo GPS para establecer la posición del usuario, el sensor simplemente brindará al sistema un par *(latitud, longitud)*. En este caso, a pesar que la información brindada es útil, a la persona que hace el reparto no le es de gran ayuda saber su posición en términos de un par *(latitud, longitud)*, sino que lo que le interesa es saber la calle y la altura a la que se encuentra. De ésto se desprende que se debe realizar una conversión entre los dos sistemas; el punto crítico es establecer *quién* debe realizarla.

Una solución tentadora sería colocar la responsabilidad de realizar la conversión en el sensor mismo. Eso nos llevaría a modelar un sensor que posea la inteligencia necesaria para transformar un valor de entrada en formato *(latitud, longitud)* en una dirección, la cual podrá luego ser utilizada por la aplicación.

Esta opción, a primera vista correcta, plantea dos grandes problemas:

- El mecanismo de sensado debe conocer explícitamente al sistema de ubicación para interactuar con éste y lograr la conversión. Esto implica que los cambios en el modelo de ubicación impactan en la implementación del sensor.
- El sensor no puede ser reutilizado para otra aplicación que necesite otro tipo de conversión (por ejemplo, traducir pares *(latitud, longitud)* al número de kilómetros recorridos de una ruta.

Una alternativa para solucionar el problema de las conversiones es separar la adquisición de la interpretación en dos capas. En la capa de bajo nivel se encuentran los mecanismos de adquisición de datos, los cuales no poseen ningún conocimiento respecto de cómo van a ser utilizados. Sobre esta capa se coloca otra que se encarga de realizar las transformaciones pertinentes, lo que evita los dos problemas mencionados anteriormente, permitiendo reutilizar los sensores en diversas aplicaciones. En la arquitectura presentada, la capa de abstracciones de hardware es la que contiene a los mecanismos de bajo nivel y la capa de aspectos de sensado es lo que se encarga de la conexión con el modelo y de realizar las conversiones pertinentes.

4.8 Aspectos de Sensado

Como se indicó en la sección anterior, la adquisición de datos es realizada por medio de clases que representan a los sensores de hardware. Una vez que se ha capturado la información externa a la aplicación, ésta debe ser interpretada y convertida en una serie de envío de mensajes, los cuales son enviados a el(los) aspecto(s) de contexto correspondiente(s). A modo de ejemplo, supongamos que contamos con un mecanismo de posicionamiento basado en beacons infrarrojos, los cuales transmiten un identificador único a intervalos fijos de tiempo. Para capturar el identificador se utiliza el puerto infrarrojo de la PDA, el cual está representado en la capa de abstracciones de hardware por medio de la clase `IRPort`[3]. En el momento que el puerto físico captura un identificador, la instancia de `IRPort` avisa a sus dependientes del evento, indicando el valor leído. A partir de este evento el sistema debe:

1. Capturar la notificación disparada por la instancia de la clase `IRPort`.
2. Convertir el identificador leído en una ubicación. En la arquitectura propuesta esto significa encontrar al objeto que represente al área asociada al valor leído. Dicho objeto debe tener tipo `Location` .

[3]Dado que en una PDA existe un sólo puerto infrarrojo, la clase `IRPort` cumple la función de un Singleton [25].

3. Indicarle al usuario que ha ingresado en el área encontrada en el paso anterior. En el ejemplo del campus esto significa enviarle al objeto `Location.User` el mensaje *`#location:`*, pasándole como parámetro su nueva ubicación.

Este conjunto de acciones se encuentran encapsuladas en la clase `SensingConcern`, que es la encargada de mediar entre la capa de sensado y la de ubicación. Para llevar esto a cabo, la clase `SensingConcern` interactúa con dos clases:

`SensingPolicy`. Representa la política de interacción con el sensor, la cual puede ser *push* o *pull*. En el caso de trabajar con una política push (`PushSensingPolicy`), el sensor enviará un aviso al recibir un nuevo valor. En el caso de trabajar con una política pull (`PullSensingPolicy`), ésta tendrá la responsabilidad de consultar constantemente al sensor, para determinar si ha capturado algún valor nuevo.

`Dispatcher`. Es el encargado de determinar si un valor captado por un sensor debe ser despachado a su destino o no. En la mayoría de los casos se utiliza el dispatcher estándar (`ForwardDispatcher`), que redirecciona todos los valores que recibe. En caso que se requiera un tratamiento especial, el dispatcher puede ser reemplazado. Por ejemplo, se puede crear un dispatcher que descarte los valores sensados que tengan un error mayor al 15%.

En la figura 4.9 se muestra un diagrama que indica la relación entre las clases descriptas.

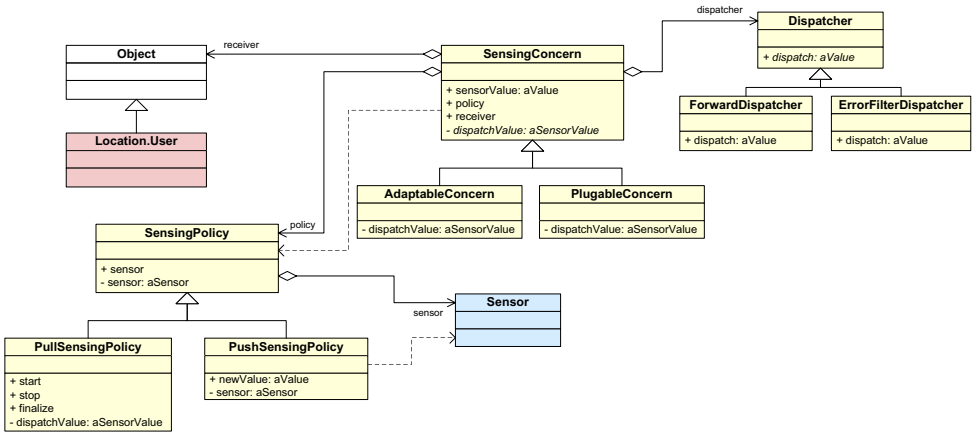

Figura 4.9: Diagrama de clases de la capa de sensado

Es importante aclarar que, con el objetivo de presentar un modelo sencillo, la figura 4.6 se muestra una relación de dependencia entre el sensor (`IRPort`) y el `SensingConcern`. Esta relación es conceptual, ya que como puede apreciarse en la figura 4.9, las relaciones de dependencia se generan entre la política de sensado y el `SensingConcern` y entre el sensor y su política (en el caso que ésta sea *push*).

El último paso que debe realizar un sensing concern es convertir el valor obtenido por el sensor en un objeto *ubicación* y enviar el mensaje `#location:`. En el caso de trabajar con beacons infrarrojos lo que debemos hacer es asociar identificadores con áreas, lo que puede hacerse usando un diccionario. A continuación se muestra un extracto de código que indica como instanciar un `SensingConcern` para que lleve a cabo esta tarea. Notar que, a pesar que a lo largo del texto se habla de la clase clase `SensingConcern`, ésta es en realidad abstracta. Como se ve en el diagrama de clases, las subclases concretas (como `AdaptableConcern` y `PluggableConcern`) son quienes efectivamente realizan la transformación entre la capa de sensado y el aspecto de contexto. Para clarificar la explicación, en la figura 4.10 se muestra un diagrama de interacción que indica cómo colaboran los objetos mencionados.

```
| port policy sc |

port:=IRPort instance.
policy:=PushSensingPolicy on: port.
sc:=PluggableConcern
             policy: policy
             receiver: user getLocationAspect
             block: [:usr :value | usr location: (idArea at: value)].
sc dispatcher: (ForwardDispatcher for: sc).
```

4.9 Capa de Servicios

En las secciones anteriores se explicó cómo construir las capas que modelan los aspectos de contexto (en particular la ubicación del usuario), la relación que tienen con el modelo de aplicación y cómo adquirir y abstraer la información capturada por los sensores de hardware. Sobre estas capas se construye la capa de servicios, que es la que contiene el comportamiento variable de las aplicaciones context-aware. Dicho comportamiento es modelado en el framework por medio de **servicios**.

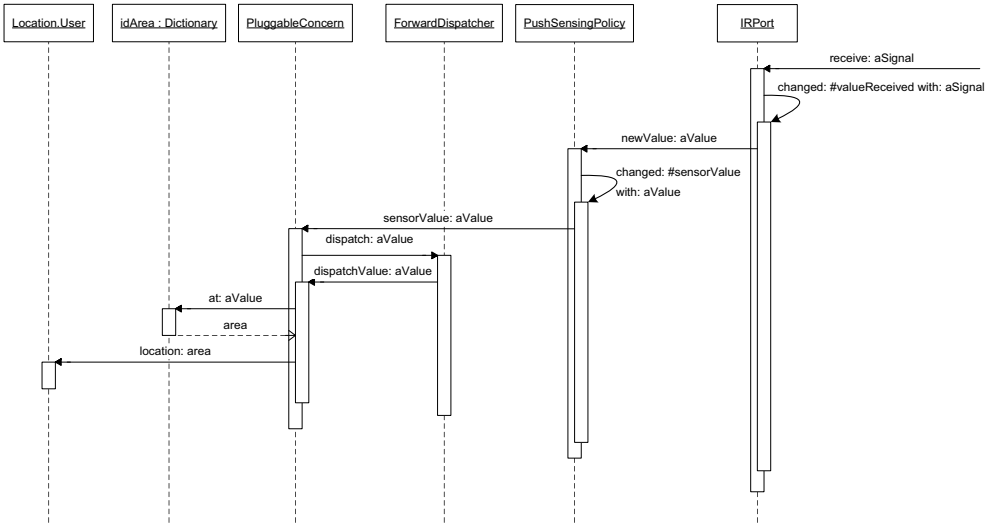

Figura 4.10: Envío de mensajes a partir de un cambio en un sensor

4.9.1 Modelo de Servicios

Los servicios se modelan en el framework como objetos de primer orden. Dado que los servicios deben implementar un conjunto de responsabilidades en común, existe una clase abstracta `UserService` de la cual todos los servicios que se agreguen a la aplicación deben ser subclases. La clase `UserService` contiene un conjunto de mensajes concretos (utilizados para el manejo interno del framework) así como un conjunto de mensajes abstractos (*hooks*) que deben ser implementados por las subclases concretas para definir el comportamiento de cada servicio. Gracias a la clase abstracta `UserService`, el framework puede tratar a todos los servicios de igual manera, simplificando su desarrollo y la adición de nuevos servicios.

A nivel conceptual existen dos tipos de servicios: aquellos que utilizan al modelo de aplicación y aquellos que son independientes de éste. A pesar que los últimos son importantes, los casos más complejos e interesantes surgen al trabajar con servicios que deben interactuar de alguna forma del modelo; desde esta perspectiva, los servicios son vistos como módulos que *extienden* la funcionalidad de la aplicación.

El diseño de los servicios generalmente es realizado sin importar que existan otros servicios, lo que implica que los servicios son independientes unos de otros y no son conscientes de la existencia de otros servicios. A pesar de ésto, en algunos casos es necesario que un servicio se encuentre activo para que otro servicio pueda funcionar (por ejemplo, un servicio de alarma puede depender del

servicio de agenda). Para ésto el framework brinda un mecanismo que permite especificar una relación de pre-requisitos entre los servicios: si un servicio S_1 es pre-requisito de S_2, S_2 no será presentado al usuario hasta tanto S_1 no se encuentre activo.

La capa de servicios no sólo especifica qué servicios existen, sino que también es responsable de saber qué servicios se encuentran activos para un determinado usuario. Para lograr ésto es necesario contar con una representación del usuario desde la óptica de los servicios, la cual se construye por medio de la clase `Service.User`. Esta clase representa un vista alternativa del usuario, cuyo foco está puesto en lo relativo a los servicios (servicios a los que el usuario se ha suscripto, servicios activos, etc). Asimismo un `Service.User` tiene la responsabilidad de construir una imagen de alto nivel del usuario, mediando entre todos los aspectos de contexto que sean relevantes. En el ejemplo que venimos tratando, el aspecto de contexto relevante es la ubicación del usuario, por lo que el `Service.User` debe conocer y ser dependiente del `Location.User`. Gracias a ésto, el `Service.User` puede reaccionar ante un cambio en la ubicación del usuario y así determinar qué servicios se encuentran disponibles.

Para completar el modelo es necesario enmarcar el ambiente de interacción entre los servicios y los usuarios. Para esto se crea la noción de un ambiente de servicios, el cual modela el entorno de servicios en que se encuentran inmersos los usuarios. Este entorno se representa por medio la clase `ServiceEnvironment`, siendo su responsabilidad llevar cuenta de los servicios disponibles, de los usuarios que se encuentran activos y de mediar entre los diversos componente del framework al producirse un cambio en el contexto de un usuario. Como veremos más adelante, el entorno tiene la responsabilidad de saber qué servicios se encuentran disponibles en una determinada área geográfica.

4.9.2 Creación de servicios

Para agregar un nuevo servicio se debe crear una subclase de `UserService`; cada subclase concreta de ésta cumple el rol de un Command [25]. El comportamiento particular de cada servicio viene dado por la implementación de uno o más mensajes abstractos, los cuales son invocados automáticamente por el framework. A modo de ejemplo, en el momento que un servicio es agregado a un usuario, éste recibe el mensaje `#addedTo:aUser`, cuyo comportamiento no está definido en `UserService`. Las subclases concretas de `UserService` pueden sobreescribir este mensaje y realizar aquellas acciones que correspondan.

Dado que no hay restricciones *a priori*, un ambiente puede contener una gran cantidad de servicios. A pesar de ésto, un usuario no tiene por qué estar interesado en utilizar todos esos servicios, por lo que es necesario contar con un mecanismo que permita discriminar entre los servicios que le interesan al usuario y los que no. Para ésto se define el concepto de subscripción: un servicio

será presentado a un usuario solo si éste se ha subscripto previamente a dicho servicio. Por este motivo, una vez que se ha creado la clase concreta del servicio, es necesario publicarla para que los usuarios se puedan suscribir.

Otra de las responsabilidades de la clase `ServiceEnvironment` es llevar cuenta de los servicios disponibles a los usuarios. Por ejemplo, para indicar que un servicio ha sido publicado en el ambiente se utiliza el mensaje *`#addAvailableService: aServiceClass`*. Una vez que se ha publicado un servicio en el ambiente, los usuarios pueden acceder por medio del ambiente y decidir si se quieren suscribir a alguno de ellos. Para llevar esto a cabo, la clase `Service.User` posee el mensaje *`#subscribeTo:aServiceClass`* (análogamente un usuario puede des-suscribirse utilizando el mensaje *`#unsubscribeFrom:aServiceClass`*).

De la misma forma que un usuario puede decidir qué servicios le interesan, un servicio puede indicar qué usuarios tienen permiso de suscribirse a ellos por medio del mensaje *`#canBeUsedBy:aUser`*. A modo de ejemplo, podemos pensar en un servicio que permite llenar las actas de los exámenes finales. Dado que a este servicio sólo pueden acceder los profesores, el servicio debe redefinir el mensaje *`#canBeUsedBy:aUser`* para verificar que el usuario cumpla el rol de profesor. Notar que esta validación es simplemente para establecer qué usuarios pueden *ver* al servicio disponible; luego es responsabilidad del servicio validar que un profesor sólo acceda a las actas de su materia.

4.9.3 Áreas de servicios

Hasta el momento se explicó cómo crear servicios y los mecanismos disponibles para configurar los servicios disponibles al usuario. El paso siguiente es indicar cómo relacionar servicios a determinadas áreas geográficas. Para realizar ésto se crean las áreas de servicio (`ServiceArea`). Al ingresar a un área de servicio el usuario recibe una notificación por medio del mensaje *`#enterArea:aServiceArea`*, lo cual hace que se agreguen aquellos servicios disponibles para el usuario en ese área de servicios.

A nivel de diseño, la arquitectura presenta un patrón de separación de responsabilidades recurrente. Al momento de incorporar el comportamiento geográfico al sistema, se optó por no modificar el modelo de aplicación. En lugar de ésto, se colocó una capa de ubicación que extiende en forma transparente al modelo de aplicación. La misma idea es aplicada para indicar los servicios disponibles en un área geográfica. En lugar de agregar los servicios a un elemento de la capa de ubicación, se creó una abstracción que representa un área geográfica en la cual se encuentran disponibles un conjunto de servicios: un *área de servicios*.

Un área de servicios se encuentra modelada por medio de la clase `ServiceArea`, la cual cumple el rol de un Mediator [25] entre una ubicación y un conjunto de servicios. Es importante notar que, dado que un área de servicios colabora con objetos `Location` y `UserService`, no sólo es independiente de los mecanismos de sensado subyacentes, sino que también lo es del mapa puntual que se esté utilizando. A modo de ejemplo, supongamos que queremos brindar un conjunto de servicios en un aula y en las cercanías de su entrada, como se indica en la figura 4.11. Las áreas rojas indican el área de cobertura de los beacons bluetooth colocados en el techo de edificio. Para generar el área de cobertura deseada se debe crear una instancia de `ServiceArea` cuyo colaborador *`location`* sea el resultado de realizar una unión entre el área cubierta por B_3 y la intersección entre el área cubierta por B_1 y B_2. De esta forma, cuando el `Location.User` reciba el mensaje *`#location:newLocation`* el sistema evaluará si se encuentra dentro del área de servicio definida; en caso de ser así se le agregarán los servicios correspondientes.

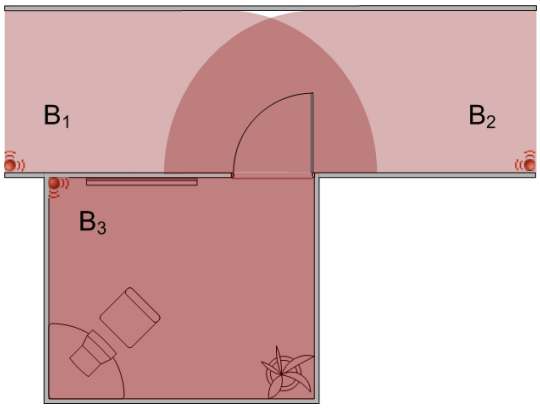

Figura 4.11: Ejemplo de área de servicio

Para brindar una idea completa de lo que sucede ante un cambio en la ubicación del usuario, en la figura 4.12 se muestra un diagrama de interacción. Desde el punto de vista de los servicios, un cambio en la capa de ubicación puede llevar a que se agreguen o quiten servicios a un usuario dependiendo de su posición anterior y actual. Para no complicar el diagrama, se muestra al cadena de mensajes a partir del cambio en la capa de ubicación. El lector interesado se puede referir al diagrama 4.10 para recordar como, a partir de un valor leído por un sensor, se le envía el mensaje *`#location:newLocation`* al `Location.User`.

Al recibir el mensaje *`#location:newLocation`*, el `Location.User` dispara un cambio, el cual es propagado a sus dependientes. En este momento el `Service.User` recibe la notificación del cambio y la redirecciona al `ServiceEnvironment`. Al enterarse del cambio, el ambiente determina las áreas

de servicio en las que se encontraba el usuario antes de moverse y las áreas de servicio actuales. En base a ésto, el ambiente le indica al usuario que ha ingresado o que debe abandonar un área de servicio por medio de los mensajes `#enterArea:aServiceArea` y `#leaveArea:aServiceArea`.

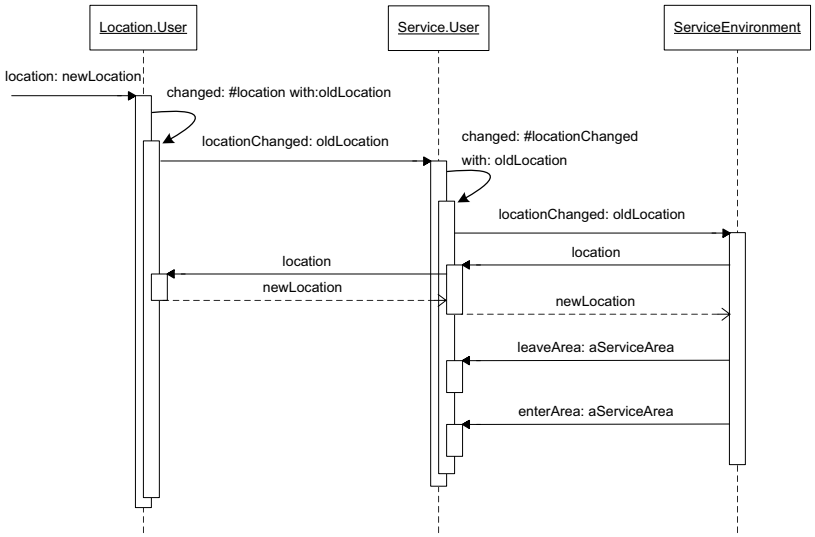

Figura 4.12: Cambio en el conjunto de servicios activos

4.10 Conclusiones

En este capítulo se planteó el alcance del trabajo, indicando qué aspectos de las aplicaciones context-aware se consideran relevantes y cuáles fueron dejadas de lado. Particularmente, el foco del trabajo se encuentra puesto en lograr una arquitectura flexible y concisa para el desarrollo de aplicaciones context-aware. Para lograr este tipo de arquitecturas una de las cosas necesarias es contar con un modelo de contexto que se adecue a su naturaleza dinámica. Por este motivo, en la sección 4.3 se describió un modelo de contexto distinto al utilizado en otras arquitecturas: en lugar de pensar en un *objeto contexto*, el contexto es modelado en base a sus aspectos. De esta forma, el contexto es el emergente de la interacción de sus aspectos, los cuales pueden agregarse y quitarse dinámicamente durante la ejecución de la aplicación.

En la sección 4.5 se presentó una arquitectura en capas, la cual consta de dos grandes concerns ortogonales: el de aplicación y el de sensado. Estos dos concerns se unen en la capa de aspectos de contexto (hasta el momento la

capa de ubicación), sobre la cual se construye la capa de servicios. En las sucesivas secciones se realizó un análisis más detallado de cada capa, indicando sus responsabilidades y la interacción entre las capas.

Capítulo 5

Una arquitectura para sistemas context-aware

And then, like the one who unchooses his own choice
And thinking again undoes what he has started
So I became: a nullifying unease.

The Inferno of Dante
Robert Pinsky

En el capítulo anterior se presentó una arquitectura basada en capas para diseñar aplicaciones context-aware. En particular se hizo hincapié en el diseño de aplicaciones location-aware. En este capítulo se describen las mejoras necesarias al framework desarrollado en este trabajo para construir cualquier tipo de aplicaciones context-aware. Para lograr ésto se utilizará un mecanismo de dependencias especializado para el trabajo con eventos de contexto. Luego se planteará un modelo genérico para modelar cualquier aspecto de contexto, finalizando con una generalización al modelo de servicios para extender la funcionalidad del propio framework. Por último se hará una breve comparación con otros desarrollos existentes.

5.1 Modelo basado en eventos para cambios de contexto

A pesar de que el mecanismo de dependencias permite establecer una independencia crucial para el desarrollo de la arquitectura en capas, no posee la flexibilidad necesaria para soportar un manejo basado en eventos. En la siguiente sección se enumerarán los problemas que surgen con el mecanismo de dependencias al intentar desacoplar los distintos aspectos de contexto. Luego se mostrará cómo desarrollar una capa intermedia, para llegar a un mecanismo basado eventos y manejadores.

5.1.1 Dificultades con el mecanismo de dependencias básico

En el capítulo anterior se mostró cómo, por medio del mecanismo de dependencias, un cambio en un aspecto de contexto se propaga hasta la capa de servicios. Para llevar esto a cabo, el aspecto de contexto (por ejemplo, el aspecto de ubicación modelado por `Location.User`) utiliza el sistema de dependencias estándar de Smalltalk, el cual requiere que se pase como parámetro al mensaje *`#changed:with:`* un símbolo que indique el motivo del cambio. En el caso del aspecto de ubicación, el envío del mensaje completo es *`self changed:#location with:oldLocation`*. La necesidad de pasar como parámetro un símbolo que indique el aspecto que cambió, se debe a que el `ServiceEnvironment` debe reaccionar en forma distinta ante los cambios en distintos aspectos de contexto. Por ejemplo, en el caso que cambie la ubicación, se debe calcular las áreas de servicio en las que se encontraba el usuario antes de moverse y las áreas de servicio en las que se encuentra luego; en base a ésto se determina si el usuario salió o entró en algún área de servicio. Supongamos ahora que el aspecto de contexto que nos interesa es la probabilidad de lluvias; en el caso que ésta sea mayor al 40%, se le muestra al usuario el pronóstico del tiempo. Si modelamos el aspecto de probabilidad de lluvias en forma similar en la que lo hicimos con la ubicación del usuario, al cambiar la probabilidad de lluvias el aspecto debería disparar un cambio ejecutando *`self changed:#probability`*. Al llegar dicho cambio al `ServiceEnvironment`, éste debería primero establecer qué tipo de cambio es (ubicación, porbabilidad de lluvias, tiempo, etc.), para luego decidir qué acción tomar.

Al intentar extender la aplicación con el agregado de tres o cuatro aspectos de contexto, resulta claro que el mecanismo planteado no escala. Uno de los grandes problemas de trabajar con este sistema, es que debemos modificar la implementación del `ServiceEnvironment` por cada nuevo aspecto de contexto que agreguemos. Como resultado de esto se genera una clase difícil de mantener y extender, ya que contiene una gran cantidad de métodos complejos. Para lograr un sistema que escale a medida que se agregan nuevos aspectos de contexto, es necesario colocar la responsabilidad del manejo de los cambios en un conjunto de objetos externos al ambiente de servicios. De hecho, sería ideal que al agregar un aspecto de contexto, se indique cómo manejar el cambio disparado por él sin tener que modificar el comportamiento central del framework.

5.1.2 Especialización del mecanismo de dependencias

El primer paso para llegar a un sistema con estas características es modelar a los cambios como objetos en lugar de ser simples símbolos. Esta idea no es nueva y se ha aplicado en muchos sistemas que requieren características similares, siendo los más populares los frameworks de manejo de interfaces gráficas. En estos fra-

meworks, los eventos son modelados como objetos de primer orden, siendo parte de una jerarquía cuya superclase representa un evento abstracto (`Event`). Como subclases encontramos taxonomías particulares, como la de eventos de mouse (`MouseEvent`), de teclado (`KeyboardEvent`) o de ventanas gráficas (`WindowEvent`), cada uno con sus subclases concretas. Tomando como punto de partida este modelo de eventos, el primer paso es lograr una abstracción que represente un evento context-aware, el cual tenga en cuenta el aspecto que cambió, el objeto que originó el cambio y los parámetros relevantes al cambio.

Para lograr una integración transparente con el ambiente, se extendió la clase `Object` con un sistema de dependencias refinado para eventos context-aware. Esta extensión comprende dos tipos de mensajes: los que se utilizan para disparar eventos y aquellos que indican cómo reaccionar ante un evento de contexto.

El disparo de un evento context-aware se realiza por medio del mensaje *`#contextEvent:anEventName`* o alguna de sus variantes. Esto hace que el sistema cree un objeto `ContextEvent` y lo configure con el nombre del evento, el objeto que disparó el cambio y los parámetros brindados por el emisor. Cuando un objeto dispara un evento de contexto, se avisa[1] a los objetos que previamente se habían registrado por medio del mensaje *`#onContextEvent:anEventName send:aSelector to:anObject`*.

5.1.3 Manejo de eventos de alto nivel

A pesar que el sistema planteado especializa el mecanismo de dependencia para eventos context-aware, todavía estamos limitados al manejo de símbolos y a modificar al `ServiceEnvironment` por cada aspecto que sea necesario agregar al sistema. Para solucionar ésto es necesario construir un sistema que trabaje con eventos y manejadores (*handlers*), de forma tal que cada evento pueda ser atendido en forma independiente y sin modificar al `ServiceEnvironment`. Para esto se crea un manejador por tipo de evento, el cual debe registrarse al `Service.User`. En el momento en que se produce un evento, el `Service.User` recibe la notificación, busca al manejador de dicho evento y notifica del cambio al `ServiceEnvironment`. Luego éste se encarga de activar al manejador indicándole el evento, el aspecto que lo generó y el ambiente de servicios en el que se está ejecutando.

Para dar una visión mas detallada del manejo de eventos, a continuación veremos como migrar la implementación presentada en el capítulo 4 a un sistema basado en eventos.

[1] Este aviso se logra por medio de un `ContextEventTransformer`, que es una variación del `DependencyTransformer`, especializado para eventos context-aware.

Al igual que en el sistema básico de eventos, un objeto indica un cambio utilizando uno de los siguientes mensajes:

```
#contextEvent:anEventName
#contextEvent:anEventName with:anArgument
#contextEvent:anEventName with:firstArgument with:secondArgument
#contextEvent:anEventName withArguments:anArray
```

Luego, en `Location.User` el método que modifica la posición del usuario debe realizar:

```
location: aLocation

    | oldLocation |

    oldLocation:=self location.
    self setLocation: aLocation.
    self contextEvent: #location with: (#oldLocation -> oldLocation).
```

Una vez disparado el cambio, es necesario crear un manejador. Para ésto se debe definir una subclase de `EventHandler`, la cual debe implementar el mensaje *#handle:aContextEvent from:aSender on:aServiceEnvironment*. En nuestro ejemplo, dicha clase se llamará `LocationEventHandler`. Una vez definido su comportamiento, el handler es registrado en el `Service.User` para manejar los eventos correspondientes. Dicha registración se realiza por medio del mensaje *#addEventHandler:anEventHandler*. Por ejemplo, si `user` es una instancia de `Service.User`, se debe evaluar:

```
user addEventHandler: LocationEventHandler new.
```

En el momento que el `Service.User` recibe la notificación de un evento, busca entre sus handlers para determinar cuál debe manejar dicho evento. Para determinar ésto, se busca en la colección de manejadores registrados a aquél que responda `true` al mensaje *#canHandle:anEvent*. Este mensaje tiene una implementación estándar en la clase `EventHandler`, que es chequear por el nombre del evento. En el caso de requerir un chequeo más complejo, se puede redefinir el mensaje en cualquier subclase de `EventHandler`, aunque en la práctica no ha sido necesario.

Los eventos tienen la particularidad que sus colaboradores son definidos dinámicamente al ser disparados por un aspecto de contexto. En particular, los parámetros que se pasan al disparar un mensaje son nombrados, lo que permite establecer una relación entre nombres y objetos. Por ejemplo, al disparar un cambio en la ubicación, el evento que se genera conoce a un objeto `Location` por medio del nombre `oldLocation` (por ese motivo el parámetro que se le pasa al evento es `(#oldLocation -> oldLocation)`).

En forma adicional, cuando un evento es configurado con un conjunto de colaboradores nombrados, automáticamente comprende los mensajes para acceder a éstos. En el caso de la ubicación, objeto evento responderá al mensaje *`#oldLocation`* retornando la posición previa del usuario.

Para completar el ejemplo, a continuación se presenta el código del mensaje *`#handle:aContextEvent from:aSender on: aServiceEnvironment`* de la clase `LocationEventHandler`.

```
handle: aContextEvent from: aSender on: aServiceEnvironment

  | old current removed added |

  current:=aServiceEnvironment getServicesAreaFor:
                              (aContextEvent originator location).
  old:=aServiceEnvironment getServicesAreaFor:
                              (aContextEvent oldLocation).
  removed:=old - current.
  added:=current - old.
  removed do:[:sa | aSender leaveArea: sa].
  added do:[:sa | aSender enterArea: sa].
  (removed isEmpty and:[added isEmpty])
                              ifTrue:[aSender updateServices].
```

5.2 Modelo de aspectos de contexto

Habiendo definido un modelo de cambios basado en eventos, el paso siguiente es generalizar el aspecto de ubicación (`Location.User`) a cualquier aspecto de contexto. Para lograr un manejo dinámico de los aspectos, el primer paso es modificar la estructura del `Service.User` para que trabaje con una colección de aspectos de contexto en lugar de simplemente la ubicación. Para ésto se reemplaza la relación de conocimiento *`locatedUser`* por una colección de aspectos llamada *`contextAspects`* y se define un protocolo de manejo de aspectos de contexto.

Dicho protocolo abarca los siguientes mensajes:

```
#addAspect:aContextAwareAspect
#findAspect:aBlock
#removeAspect:aContextAwareAspect
```

Una vez que el usuario cuenta con el protocolo de manejo de aspectos de contexto, el siguiente paso es convertir al `Location.User` en un aspecto de contexto, lo cual se logra creando la clase `LocationAspect` y migrando el código con las modificaciones anteriormente explicadas.

Es interesante notar la concordancia que hay entre el concepto subyacente de un aspecto de contexto y su implementación. Como se planteó en la sección 4.5, un aspecto de contexto *extiende* al modelo para darle características que no posee. Luego, a pesar que el contexto en sí mismo no está acotado, las responsabilidades de cada aspecto de contexto están perfectamente definidas. Es por esto que no resulta casual que la implementación de `LocationAspect` se reduzca a tan sólo los mensajes encargados de llevar cuenta de la posición del usuario:

```
LocationAspect>>location

                    ^location.

LocationAspect>>location: aLocation

                    | oldLocation |

                    oldLocation:=self location.
                    location:=aLocation.
                    self
                         contextEvent: #location
                         with: (#oldLocation -> oldLocation).
```

Al separar y generalizar el manejo de aspectos de contexto, no sólo logramos una estructura escalable sino que también soporta modificaciones en tiempo de ejecución. A pesar de ésto, el trabajo no está completamente terminado. En nuestro modelo de servicios, todavía queda un elemento dependiente de la ubicación que no ha sido generalizado aún: las áreas de servicio. Como se explicó en la sección 4.9.3, las áreas de servicio se utilizan para indicar en qué lugar geográfico se encuentran disponibles los servicios. En el momento que un usuario ingresa a un área de servicios, los servicios disponibles en ésta son agregados en la lista de servicios activos del usuario.

A pesar de ser un concepto correcto y a todas luces útil, las áreas de servicio sólo pueden utilizarse en el caso que el aspecto de contexto sea la ubicación de usuario. Por este motivo se debe lograr una generalización de este concepto y llevarlo a un conjunto de abstracciones acordes a cada aspecto de contexto que se pueda utilizar.

Desde el punto de vista de la ubicación, un área de servicios expresa un subconjunto de todas las posibles posiciones en las que un usuario se puede encontrar. Para saber si un usuario está dentro de un área de servicios se utiliza la noción de *inclusión*; un usuario está dentro de un área de servicios si su posición se encuentra dentro del rango expresado por la ubicación del área de servicios. De esto se puede extraer que, desde el punto de vista de la ubicación, un área de servicios denota una *restricción*.

Si ahora examinamos a las áreas de servicio desde el punto de vista de los servicios, resultará que podemos verlas como contenedores de servicios. Para el `ServiceEnvironment`, un área de servicios es una abstracción que contiene un conjunto de servicios disponibles para los usuarios. En el caso que un usuario se encuentre *dentro* de una de estas áreas, se le agregarán los servicios correspondientes. Es importante notar que, luego de haber independizado a los aspectos de contexto del `ServiceEnvironment`, éste ya nada sabe del manejo de la ubicación; ésto ahora es responsabilidad del `LocationEventHandler`. Luego, el `ServiceEnvironment` ve a las áreas de servicio como *proveedores de servicios*, tal que si un usuario cumple las restricciones impuestas por éste (por ejemplo, estar parado en una determinada zona), se le presentan efectivamente los servicios.

De lo dicho anteriormente se desprende que un área de servicio cumple dos roles: por un lado es un proveedor de servicios y por el otro denota una restricción en términos de la ubicación. Como puede apreciarse, uno de los roles pertenece a la capa de servicios, al tiempo que el otro es dependiente del aspecto de contexto que se esté modelando. Por este motivo, en la capa de servicios surge el concepto de `ServiceProvider`, el cual posee un conjunto de restricciones (`Constraints`). Ante un cambio en un aspecto de contexto del usuario, los distintos proveedores de servicios evalúan sus restricciones. En el caso que se cumplan, los servicios brindados por el proveedor pasan a estar activos para el usuario. En cuanto a las restricciones, éstas aparecen como subclase de la clase abstracta `Constraint`. Cada aspecto de contexto debe generar sus propias subclases concretas para indicar en qué casos los servicios pueden ser utilizados por el usuario. Por ejemplo, para lograr el equivalente a un área de servicios se debe evaluar `ServiceProvider on:(LocationConstraint on:aLocation)`. Luego, para configurar al `ServiceProvider` se utiliza el mensaje *`#installOn:aServiceProvider`*, definido en la clase `UserService`.

A continuación se muestra un pequeño ejemplo en el cual se instancia un usuario con un aspecto de ubicación y dos servicios (un mapa y una grilla de actividades):

```
"aPerson es parte del modelo de aplicación"
"map es el mapa del modelo de ubicación"

env:=ServiceEnvironment new.

user:=User model: aPerson.
user addAspect: (LocationAspect on: map).
user addEventHandler: LocationEventHandler new.

env addUser: user.
env addAvailableService: MapService.
env addAvailableService: SchedulerService.

sp:=ServiceProvider on:
         (LocationConstraint on:
                (map getLocationNamed: 'Aula 1')).

(env getServiceByName: 'Map Service') installOn: sp.
(env getServiceByName: 'Scheduler Service') installOn: sp.

env addServiceProvider: sp.
```

Habiendo planteado esta separación de clases y responsabilidades, estamos en condiciones de indicar las abstracciones que caracterizan a los aspectos de contexto. En la figura 5.1 se muestra un diagrama *conceptual* que describe el modelo de un aspecto de contexto. Este modelo se compone de cuatro abstracciones básicas:

Modelo de aspecto de contexto: Es el encargado de modelar el dominio del aspecto. En las aplicaciones location-aware, este rol lo cumple el modelo de ubicación, el cual está definido por el tipo `Location`.

Aspecto de contexto: Incorpora el aspecto de contexto al usuario. En las aplicaciones location-aware, este rol lo cumple la clase `Location.User`.

Manejador: Captura el cambio disparado por el modelo de aspecto de contexto y reacciona según corresponda. En las aplicaciones location-aware, este rol lo cumple la clase `LocationEventHandler`.

Restricción: Se utiliza para determinar si un usuario puede acceder a los servicios disponibles en un proveedor de servicios. En las aplicaciones location-aware, este rol lo cumple la clase `ServiceArea`.

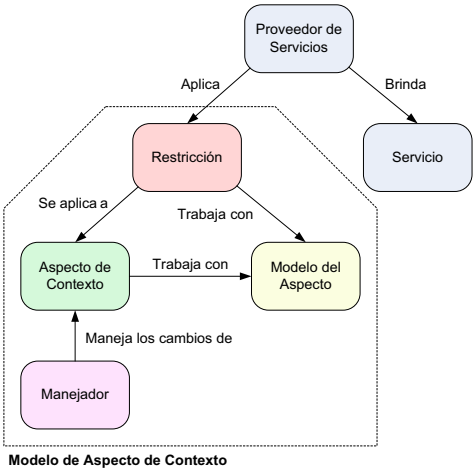

Figura 5.1: Esquema de un aspecto de contexto

A continuación, en la figura 5.2 se muestran cuatro ejemplos concretos del modelo para distintos aspectos de contexto.

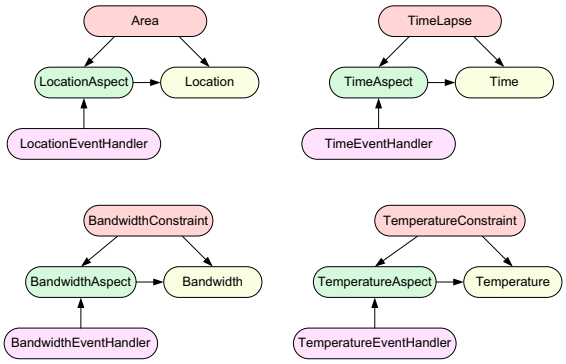

Figura 5.2: Ejemplos de contexto

Es importante notar que ahora la capa de servicios trabaja directamente con el modelo de aplicación y los aspectos funcionan como abstracciones que enriquecen la relación entre la capa de servicios y el modelo. En el caso que lo requiera, el aspecto de contexto puede conocer al modelo de aplicación, aunque esto no siempre es necesario (por ejemplo, el aspecto del ancho de banda de la conexión es independiente del modelo de aplicación). En la figura 5.3 se muestra

un diagrama de paquetes, indicando la relación entre los servicios, los aspectos de contexto y el modelo de aplicación. Luego, en la figura 5.4, se muestra un diagrama de clases indicando la relación entre el `ServiceProvider`, los servicios (`UserService`) y las restricciones (`Constraint`).

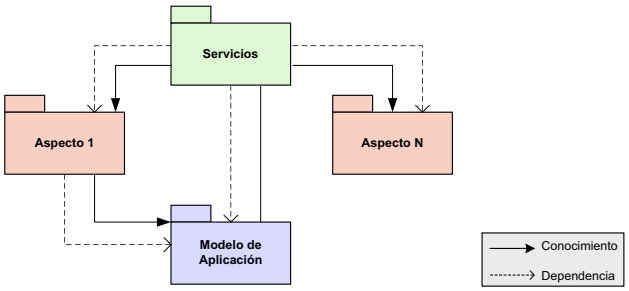

Figura 5.3: Relación entre los paquetes de servicio, los aspectos de contexto y el modelo de aplicación

5.3 Extensión a los Servicios

Generalmente los frameworks evolucionan de modelos de caja blanca (basados en herencia) a modelos de caja negra (basados en composición) [47]. El objetivo de este trabajo es llegar a un modelo de caja gris, donde la flexibilidad del comportamiento se logre por composición en la mayoría de los casos, pero con la posibilidad de extender el framework por subclasificación en el caso de ser necesario. Un buen ejemplo de este equilibrio está dado por el modelo de servicios. Desde el punto de vista del framework, los servicios son componentes que respetan un mismo protocolo y que pueden ser agregados o eliminados a un usuario. Desde el punto de vista del programador, cuando es necesario un nuevo servicio, se debe crear una subclase de `UserService` (caja blanca). Esta nueva clase luego es incorporada al framework en tiempo de ejecución por medio del mecanismo de composición (caja negra).

Como veremos en las próxima sección, a pesar que este modelo parece completo a primera vista, presenta un problema de escalabilidad al querer implementar diversos servicios. En las siguiente sección se presenta el problema y su solución, dejando para las secciones 5.3.2 y 5.3.3 una explicación detallada de la taxonomía de servicios.

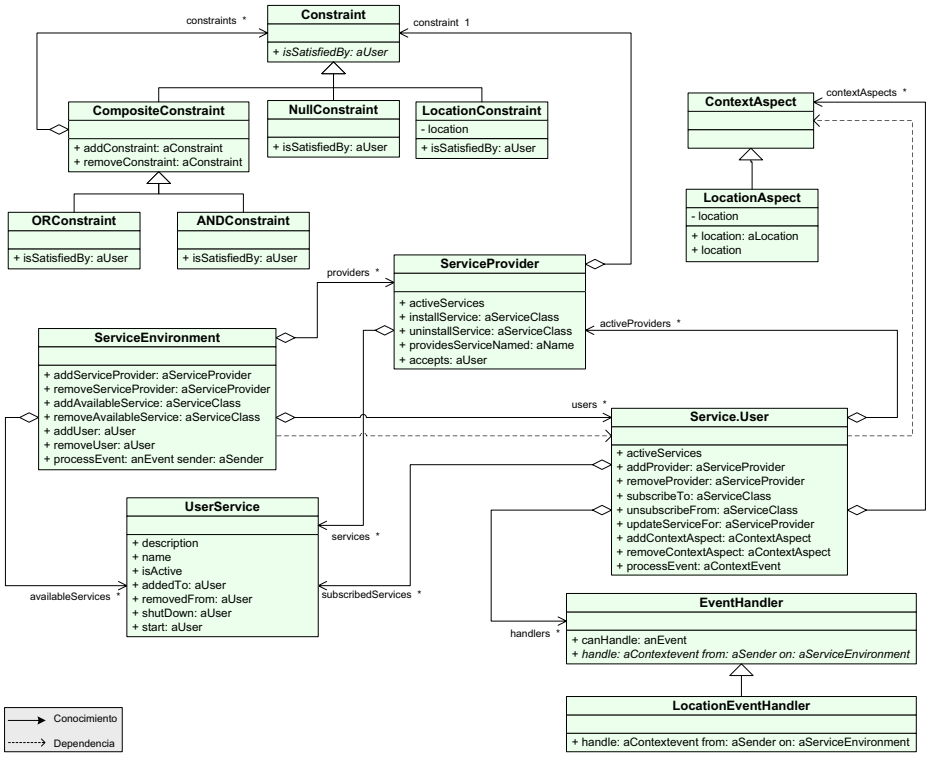

Figura 5.4: Diagrama de clases de la capa de servicios

5.3.1 Deficiencia en el modelo de servicios

Supongamos que queremos proveer un sistema de awareness para el usuario, de forma tal que al ingresar a un aula se muestren todos los usuarios que se encuentran dentro de ella. Asimismo es necesario que la lista de usuarios se actualice en el momento que un usuario entra o sale del aula. Para implementar un servicio de este estilo es necesario que un objeto lleve un registro de los usuarios que se encuentren en el aula en todo momento. Asimismo, dicho objeto debería avisar a todos los usuarios cada vez que otro usuario entre o salga del aula.

Una posible solución sería establecer un conocimiento directo entre los usuarios: cada usuario tendría una colección de todos aquellos usuarios que se encuentran en el ambiente de servicios. Una vez que se establece este conocimiento el usuario puede hacerse dependiente de todos los otros usuarios para enterarse cuando cambian su posición. De esta forma, un usuario podría determinar qué otros usuarios se encuentran en la misma aula que él.

Otra posible forma de solucionar este problema sería agregar una relación de conocimiento desde los proveedores de servicio hacia los usuarios. Con esta configuración habría que definir un proveedor de servicios por aula, tal que por medio del mecanismo de dependencia avise cuando pasa a estar activo para un usuario. Luego se debería definir un servicio que sea dependiente del proveedor de servicios y que actualice el listado de usuarios por cada cambio en el proveedor.

A pesar que ambas soluciones son sencillas de implementar y requieren cambios menores en el framework, contradicen al objetivo de llegar a un framework de caja gris. Así como es necesario agregar una relación de conocimiento y un par de métodos para este servicio en particular, muchos otros servicios requieren de "pequeñas modificaciones" en el comportamiento central del framewrok. Por este motivo es necesario encontrar un mecanismo que permita extender el comportamiento *del propio framework* sin que sea necesario modificar las clases que definen su comportamiento interno.

Para solucionar este problema se optó por seguir utilizando la metáfora del servicio, pero separado en dos tipos: servicios internos y servicios de usuario.

5.3.2 Servicios Internos

Un servicio interno representa, al igual que un servicio de usuario, un comportamiento que puede ser agregado o eliminado dinámicamente al framework. Desde un punto de vista arquitectural, un servicio interno cumple el rol de un Command [25]. A diferencia de un servicio de usuario, un servicio interno no es funcionalidad que se brinda al usuario, sino funcionalidad que se brinda al propio framework. De hecho, como veremos más adelante, este tipo de servicios permite realizar introspección sobre el propio framework.

Lo servicios internos son asociados a proveedores de servicios, para que puedan ser utilizados por otros servicios internos o por servicios de usuario. En el ejemplo del servicio de awareness, el primer paso consiste en crear un servicio interno que cumpla el rol de *servidor de awareness*. Este servicio (`AwarenessServer`) será el encargado de llevar cuenta de los usuarios que se encuentran en una determinada área y de avisar a los usuarios en el caso que alguien entre o salga del área en cuestión. Una vez que se ha implementado el servidor de awareness, es necesario instanciar un proveedor de servicios (`ServiceProvider`), el cual contenga al servicio interno y provea de servicios de usuario a cada usuario distinto (`AwarenessService`, una subclase de `UserService`). Por último es necesario indicar la restricción que tendrá el servicio, que será una instancia de `LocationConstraint` indicando el área de awareness. Con esta configuración, cuando el usuario ingrese dento del área definida por el `LocationConstraint` se le agregará el servicio de usuario que le permitirá visualizar a los otros usuarios. Asimismo, este servicio avisará al `AwarenessServer` que el usuario ha ingresado,

lo que le permitirá avisarle a los otros servicios de usuario (`AwarenessService`) para que actualicen su listado.

5.3.3 Servicios de Usuario

Como se explicó en la sección anterior, los servicios internos representan comportamiento activo dentro de un proveedor de servicios. A diferencia de éstos, los servicios de usuario no realizan ningún comportamiento activo dentro del proveedor de servicios, sino que son creados para cada usuario y "viven" con (y para) él. Esto lleva a que, desde el punto de vista del usuario, un proveedor de servicios sea un contenedor de servicios que tiene la responsabilidad de determinar qué servicios puede brindarle y cómo instanciarlos.

Desde el punto de vista de los servicios de usuario, el rol del proveedor es más importante, ya que no sólo debe instanciarlos, sino que también debe proveer aquellos servicios internos que pueda requerir el servicio de usuario para funcionar. En el ejemplo de awareness, el servicio de usuario *requiere* de un servidor para poder ser instanciado y coordinar el trabajo entre los usuarios.

Una primera solución para este problema es, como se planteó en la sección anterior, crear pares de servicio/servidor (`AwarenessService`/`AwarenessServer`) por cada proveedor de servicios. De esta forma, al crear el servicio de usuario nos aseguramos que exista un servidor de awareness que lo coordine. A pesar que esta solución funciona, resulta engorrosa ya que hay que configurar tantos proveedores de servicios con pares servicio/servidor como áreas en las que se quiera brindar el servicio de awareness. Asimismo no expresa la relación de requerimiento que hay entre el servicio de usuario y el servicio interno.

Una solución alternativa sería pensar en una dependencia entre los servicios internos y de usuario; en este caso, el servicio de awareness estará disponible en todos aquellos lugares en los que haya un servidor que lo coordine. Este concepto se puede llevar a la práctica utilizando el modelo de pre-requisitos ya existenes, extendido para el manejo de servicios internos y de usuario. Para esto, un servicio de usuario puede indicar qué servicios internos deben estar activos para que éste sea brindado al usuario. De ésta forma se evita la necesidad de configurar al ambiente con pares servicio/servidor, siendo necesario crear un único proveedor para el servicio de usuario y un proveedor con un servidor por área de awareness. En el momento que el usuario ingresa en el área en que se encuentra un servidor de awareness, se cumple el pre-requisito del servicio de usuario, por lo que éste es instanciado y mostrado al usuario.

En la figura 5.5 se muestra un diagrama de clases con la jerarquía de servicios completa.

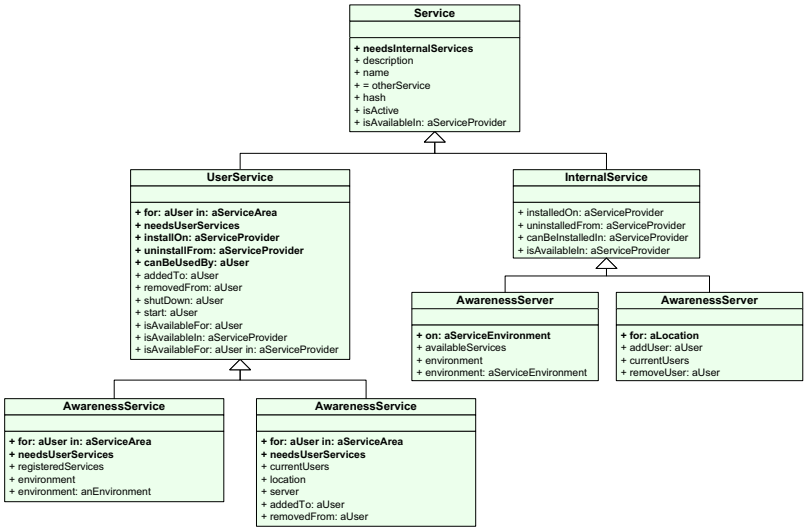

Figura 5.5: Jerarquía de servicios

5.4 Comparación con otros desarrollos

Desde el punto de vista conceptual, el modelo de contexto planteado resulta similar al presentado por Dourish [19] en sus trabajos y por lo tanto radicalmente distinto a la mayoría de los trabajos realizados en el área. Un caso particular dentro de éstos que merece ser mencionado es el trabajo realizado por Coutaz y Rey [11], ya que consideran al contexto dentro de la situación en la que se encuentra el usuario. Lamentablemente, a pesar que la idea de trabajar en base a una situación es correcta, la implementación propuesta toma al contexto como una *n-upla* de datos, lo que nos lleva nuevamente a una separación entre datos de contexto y comportamiento.

Desde un punto de vista arquitectural, el diseño presentado puede ser tomado como una evolución del Context Toolkit de Dey [48], que fue el primero en plantear una clara separación entre los mecanismos de sensado, interpretación y utilización de la información de contexto. A pesar de ésto, el trabajo de Dey continúa planteando un fuerte acoplamiento entre el contexto y la lógica de aplicación. Contrario a ésto, la arquitectura presentada en este trabajo plantea una clara separación entre el modelo de aplicación (libre de cualquier comportamiento contextual) y los diversos aspectos que hacen al contexto.

La separación en capas sobre la que se ha trabajado puede ser comparada con varios diseños previos [33, 21], ya que en general se reconocen al menos una capa de sensado y otra de aplicación. A pesar de ésto, hasta el momento no se ha presentado otra arquitectura que vea a la responsabilidad de adquirir datos

como un requerimiento ortogonal al sistema (*crosscutting concern*) y plantee una arquitectura que permita manejarlo en forma independiente.

En lo que respecta al trabajo de Bardram con el JCAF [1], éste ve al contexto como un conjunto de `ContextItem`s, lo cual se acerca a la noción de aspectos de contexto. Lamentablemente, los `ContextItem`s de Bardram funcionan básicamente como repositorio de datos y no como las unidades de comportamiento que deberían ser los objetos.

5.5 Conclusiones

A lo largo del capítulo se mostró cómo el framework presentado en el capítulo anterior puede ser adaptado para manejar cualquier aspecto de contexto. Para esto fue necesario extender el mecanismo de dependencias, generalizar las áreas de servicio y realizar modificaciones que permitan agregar funcionalidad al framework sin tener que modificar las clases más importantes de éste. A pesar de ésto, la arquitectura en capas presentada en el capítulo anterior se mantuvo sin modificación alguna.

Capítulo 6

Conclusiones y trabajo futuro

"You get what anyone gets - you get a lifetime" - Death

Death: The High Cost of Living - The Sandman
Neil Gaiman

En el presente trabajo de grado se ha diseñado una arquitectura flexible para el desarrollo de aplicaciones context-aware. Como prueba de concepto se ha implementado un framework y se ha tomado como ejemplo la construcción de aplicaciones location-aware. Por último, se indicó la forma general utilizada para modelar cualquier aspecto de contexto y las modificaciones necesarias al framework para la construcción de aplicaciones context-aware genéricas.

6.1 Resumen y conclusiones

En los últimos años ha habido un creciente interés por lograr que las aplicaciones adapten su comportamiento a diversos factores externos. En respuesta a esta motivación han surgido diversas áreas de investigación como personalización, hipermedia física, computación ubicua y aplicaciones sensibles al contexto.

Uno de los principales motores de la investigación en este tipo de aplicaciones es el avance de la tecnología en dispositivos de computación móviles (como PDAs y tablet PCs), así como la evolución de las redes inalámbricas (Wi-Fi, bluetooth, RF) y la gran variedad de sensores (GPS, Active Badge, iButtons) que permiten ingresar información de contexto a distintos dispositivos.

En el presente trabajo de grado se ha encarado el problema del desarrollo de aplicaciones context-aware desde un punto de vista arquitectural. Para ésto, el primer aporte del trabajo ha sido modelar al contexto de una forma que pueda representar su naturaleza dinámica. En dicho modelo, en lugar de ver al contexto como una entidad formal y acotada, se lo modeló en términos de un conjunto de aspectos de contexto, de cuya interacción emerge el contexto del usuario.

Una vez establecido el modelo de contexto se presentó una arquitectura para estructurar aplicaciones context-aware. Dicha arquitectura se basa en la separación de la aplicación en una serie de capas, cada una con una responsabilidad bien definida. Para lograr un bajo acoplamiento entre las capas, se utilizó un mecanismo de dependencias especializado para el desarrollo de aplicaciones context-aware. Asimismo, se ha hecho una separación de dos aspectos ortogonales de este tipo de sistemas, que son el de aplicación y el de sensado. Gracias a esta separación, se logra una independencia entre el modelo de contexto y la forma en la que se adquieren la información de los dispositivos externos.

Para verificar las suposiciones respecto del modelado del contexto y demostrar que la arquitectura se adapta satisfactoriamente a diversos requisitos, se construyó un framework denominado *Balloon*. A modo de ejemplo, en el capítulo 4 se mostró cómo instanciar el framework para desarrollar aplicaciones location-aware. Luego, en el capítulo 5 se indicó cómo generalizar los aspectos de contexto para que el framework soporte la construcción de cualquier tipo de aplicaciones context-aware.

6.2 Trabajo futuro

El área de context-awareness tiene un gran alcance, el cual no puede ser abarcado en su totalidad en este trabajo. A partir del desarrollo realizado, se ha identificado el siguiente trabajo futuro:

Ajustes de performance. Dado que el trabajo se ha centrado en el aspecto arquitectural y en lograr un modelo flexible y escalable, la performance del sistema ha sido relegada. Terminada la etapa de diseño e implementación, es ahora necesario realizar una serie de modificaciones para lograr sistemas context-aware con buena performance.

Realización de casos de prueba. A pesar que se han realizado una gran cantidad de pruebas de las aplicaciones location-aware construídas, es necesario testear al framework con diversas aplicaciones, especialmente con aquellas que impliquen trabajar con aspectos de contexto no pensados hasta el momento.

Tolerancia a fallas. Como se indicó en la sección 4.1, al implementar el framework se asumió que no habría fallas de comunicación ni de sensado. Claramente esta suposición es muy fuerte y debe ser modificada si se espera construir aplicaciones reales. Para ésto se debe construir una capa que soporte conexiones de red intermitentes y que maneje errores y pérdida de señal de los sensores.

Manejo de aspectos de contexto no deterministas. En el modelo de aspecto de contexto presentado, los proveedores de servicio se basan en restricciones para indicar si un usuario puede acceder a sus servicios. Para ésto, se asume un modelo sobre el cual se puede determinar si se cumple o no una determinada restricción. A pesar que esto se adecua bien a modelos deterministas, todavía no se ha evaluado su utilización con modelos no deterministas, como por ejemplo decidir (en base a su agenda, ubicación, cantidad de personas a su alrededor, etc.) si un usuario está o no en una reunión.

Interfaces gráficas y HCI. Un área todavía abierta es el diseño de interfaces gráficas para dispositivos con pantallas pequeñas. Asimismo, el cambio en la interacción hombre-máquina dado por los dispositivos de entrada utilizados en este tipo de dispositivos requiere de un replanteo en el diseño de aplicaciones. Por último hay un cambio radical en la atención que le brinda un usuario móvil a este tipo de aplicaciones.

Utilización de Web Services. Una extensión al framework presentado es la adición de *web services*, permitiendo presentar al usuario servicios brindados por proveedores distribuidos en la web.

Agregado de sensores lógicos. Hasta el momento, los sensores utilizados son objetos que pertenecen a la capa de abstracciones de hardware. A pesar que esto se corresponde con la idea de sensores físicos, existe la posibilidad de adquirir información por medio de sensores lógicos. A modo de ejemplo, se podría utilizar un sensor lógico que retornara el estado del tiempo de la zona en la que se encuentra el usuario. Notar que para ésto el sensor debería saber dónde se encuentra el usuario, lo que requeriría de una interacción con elementos de otras capas.

Apéndice A

Modelos de Posicionamiento

Los modelos de posicionamiento son una parte vital de los sistemas location-aware, ya que los cambios en la posición del usuario (y de los objetos alrededor del mismo) son los que disparan las modificaciones en el sistema. Hasta el momento se han planteado dos grandes modelos de posicionamiento, uno representar al espacio como un sistema de coordenadas n-dimensional y otro que lo ve como un conjunto de símbolos y sus relaciones. Al primer modelo se lo denomina geométrico y al segundo simbólico. Cada modelo presenta sus ventajas y desventajas, las que deben ser evaluadas antes de elegir cuál utilizar. Asimismo algunas aplicaciones necesitan utilizar ambos modelos para llevar a cabo su función; de hecho, en algunos casos los dos modelos deben ser utilizados al mismo tiempo.

Los modelos geométrico y simbólico son representaciones alternativas para modelar posicionamiento. Éstos pueden ser utilizados en forma independiente, pero uno no puede reemplazar al otro, por lo que el reto es lograr que ambos modelos puedan coexistir. Como veremos más adelante, este problema se ha encarado pensando en modelos de ubicación híbridos.

A.1 Modelos simbólicos

Los modelos simbólicos tratan a la ubicación de los objetos como símbolos; ejemplos de ésto pueden ser nombres como "Aula 102" o "Edificio de Matemática". Dado que tanto los lugares, como los objetos que se encuentran en éstos se representan por medio de símbolos, resulta natural pensar en los lugares geográficos como conjuntos y en los objetos (personas, medios de transporte, etc) como elementos de conjuntos. Usando este modelo decimos que un objeto se encuentra

en un determinado lugar si y solo si el símbolo que lo representa es miembro del conjunto que representa a dicho lugar.

El modelo simbólico permite establecer restricciones respecto de las ubicaciones: por ejemplo, en algunos casos, dos ubicaciones no pueden solaparse, lo que implica que un objeto puede estar, a lo sumo, en uno de los dos conjuntos. Profundizando sobre esta idea, el modelo podría establecer un orden parcial entre los símbolos, basado en la inclusión geográfica; dependiendo de si el modelo permite que las ubicaciones se solapen o no tendremos un modelo en forma de árbol o de grafo acíclico.

A pesar que existen muchos sub-modelos para trabajar dentro del modelo simbólico (como por ejemplo el basado en celdas o zonas [40]). A continuación se explicará el basado en dominios, ya que fue el elegido para realizar el trabajo de grado. En este modelo, cada dominio puede ser ordenado respecto de otros dominios. Para esto se establece un orden parcial utilizando la relación *contenido-en*, en la cual una ubicación está contenida dentro de otra si se respeta la inclusión a nivel geográfico. Notar que, como se puede ver en la figura A.1, las ubicaciones se puede solapar (el edificio "C" pertenece al campo de la universidad y a la facultad de informática).

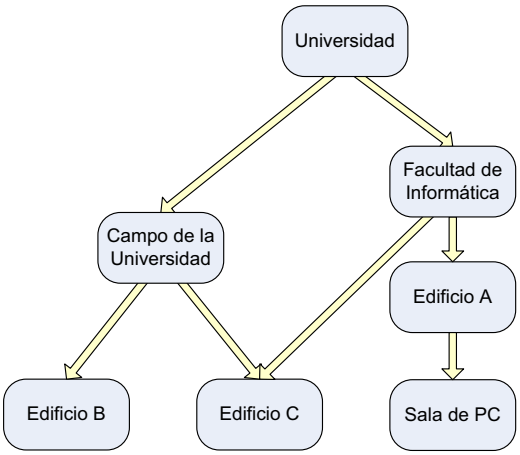

Figura A.1: Modelo de dominios con orden parcial de inclusión

Al utilizar este modelo se suele trabajar con un conjunto predefinido de dominios; los objetos se agregan y quitan de los conjuntos a medida que se mueven en el espacio geográfico. Notar que, si un objeto es parte de un dominio, también lo será de todos aquellos dominios que incluyan a éste (a estos dominios se los llama "padres"). Es importante observar que los dominios no tienen por qué ser fijos, lo que implica que la estructura de dominios estaría en constante actualización (aunque esto no es un caso común).

En este tipo de modelos no es necesario que exista una raíz que sea padre de todos los dominios, lo que implica que nuestro "mapa" puede ser un conjunto de árboles. A pesar de esta libertad, para hacer un manejo homogéneo (en el estilo del Composite [25]) se suele definir una raíz especial (*Anywhere*) que contiene a todos los árboles del mapa. Asimismo, definir un nodo *Anywhere* resulta muy práctico a la hora de indicar que el usuario ha salido del alcance del sistema, utilizándolo a nivel de implementación como un Null-Object [58].

Las principales ventajas de trabajar con un modelo simbólico son:

- Resulta sencillo para el trabajo con humanos, ya que nos referimos a conceptos conocidos (como por ejemplo, el "Edificio de Ciencias Exactas").
- El modelo jerárquico facilita el trabajo con múltiples resoluciones, brindando un mecanismo flexible para extender el sistema.
- La implementación del modelo y de las operaciones básicas (como inclusión o intersección) es sencilla.

Las desventajas de este modelo son:

- Los dominios del modelo dependen de la aplicación, lo que significa que en aplicaciones grandes con granularidad fina se debe tratar con una gran cantidad de dominios (lo cual se puede tornar inmanejable).
- En general los modelos simbólicos deben construirse a mano.
- Los modelos simbólicos restringen su resolución espacial a los datos que se hayan ingresado en el modelo explícitamente.
- No existe información respecto de la distancia entre las ubicaciones.

A.2 Modelos geométricos

Los modelos geométricos se basan en uno o más sistemas de coordenadas. Las ubicaciones se representan como puntos, áreas o volúmenes dentro de un sistema en particular, generalmente como *n-ulpas*. A diferencia del modelo simbólico, en este modelo no existe un diferencia entre lugares y los objetos que se encuentran dentro de éstos; todo se expresa por medio de coordenadas. Esta homogeneidad, a pesar de necesitar algoritmos más sofisticados, brinda un mayor potencial para desarrollar sistemas complejos. Asimismo, dentro de un mismo modelo geométrico pueden existir varios sistemas de coordenadas, lo cual permite realizar representaciones complejas. Por ejemplo, se puede modelar un colectivo en movimiento con un sistema de coordenadas (S_1)y a las personas dentro de éste con un sistema de referencia relativo al sistema del colectivo (S_2). Luego, por medio de transformaciones entre sistemas, se puede establecer la posición de un persona en el sistema S_2 respecto del sistema S_1.

En el modelo geométrico, la posición de los objetos suele estar dada por un punto en el plano en el caso de trabajar en 2 dimensiones. A pesar que esta representación ideal es útil en algunos casos, no se adecua bien a los sistemas que desean representar los objetos de la vida real. A modo de ejemplo, pensemos en modelar el lugar que ocupa una persona. Para ésto no sólo deberíamos distinguir el contexto de una persona vista desde arriba, sino que deberíamos adecuar el modelo a las diversas contexturas físicas. Dado que esto resulta molesto de implementar y en la mayoría de los casos, irrelevante, se opta por definir a los objetos como una posición y un área de cobertura. Obviamente, en algunos casos se pueden hacer mejores aproximaciones, como puede ser representar a un auto como un rectángulo.

El modelo geométrico presenta las siguientes ventajas:

- La precisión del modelo se mantiene independientemente de la forma en que haya sido cargado el modelo.
- Se pueden realizar una gran cantidad de operaciones geométricas sobre los objetos del modelo. Resulta más flexible que un modelo simbólico.
- En la gran mayoría de los casos los sistemas de referencia pueden ser reutilizados sin modificar el código.

Las desventajas del modelo son:

- Implementar las operaciones del modelo resulta complicado. Operaciones triviales en un sistema simbólico como puede ser la intersección resultan extremadamente complejos en sistemas de coordenadas en 3 dimensiones. Asimismo, el manejo de objetos en movimiento consume una gran cantidad de recursos.
- Es necesario contar con una estructura auxiliar que relacione los datos geométricos con los conceptos significativos para las aplicaciones.

A.3 Otros modelos

Los modelos de posicionamiento han sido estudiados durante mucho tiempo y se han planteado modelos alternativos y complementarios a los mencionados. Uno de éstos modelos plantea una extensión al modelo simbólico jerárquico para agregar información de distancia. En el caso de utilizar un modelo jerárquico, el árbol se complementaría con un grafo el cual uniría los nodos con aristas indicando la distancia entre las ubicaciones. Gracias a ésto podríamos saber, dependiendo de la resolución del modelo utilizado, a qué distancia se encuentran dos objetos.

Alternativamente Pradhan [45] presenta un modelo de ubicación semántico donde las ubicaciones vienen dadas por URLs de páginas web, en base a las que se genera una noción de espacio. En este modelo, cosas como la inclusión o intersección requieren una discusión que escapa al alcance de este trabajo de grado.

Por último existe la noción de modelos de posicionamiento híbridos en el cual se manejan dos representaciones (por ejemplo, simbólica y geográfica) de un mismo mapa. Estas representaciones se encuentran relacionadas para formar un modelo consistente y así obtener las ventajas de ambos modelos.

Apéndice B

Distribución de Objetos

En muchos casos, es necesario que los objetos que residen en máquinas distintas puedan comunicarse enviándose mensajes. Una forma de lograr esto es utilizar un servidor central al cual se conecten todas las máquinas que deseen interactuar con el modelo. Luego, por medio de RMI (Remote Method Invocation, una adaptación de RPC para objetos) los clientes pueden enviar sus peticiones. Una alternativa al modelo cliente-servidor es la distribución de objetos, en la cual el modelo de objetos se encuentra particionado (y eventualmente replicado) en todas las máquinas. En estos casos, todas las máquinas cumplen el rol de clientes y servidores, atendiendo peticiones de otras máquinas y enviando mensajes remotos a objetos que residen en otros hosts.

B.1 Conceptos básicos

En muchos casos, una aplicación debe ser accedida desde diversas máquinas en forma concurrente. Llevar este requerimiento a un ambiente de objetos como Smalltalk, implica que existen dos o mas imágenes cuyo objetos deben comunicarse. Generalmente estas imágenes suelen residir en máquinas distintas, distribuidas en una red y para conectarlas se utiliza un framework de distribución. El objetivo principal de un framework de distribución de objetos es lograr que los objetos puedan enviarse mensajes sin importar en qué imagen residan. Idealmente se busca una distribución *transparente*, donde el objeto que envía el mensaje no tiene forma de distinguir si el receptor es un objeto local o remoto.

Para la implementación del presente trabajo se utilizó el framework de distribución Opentalk [10]. En dicho framework, el manejo de bajo nivel de la red se realiza por medio de brokers: cada imagen posee un broker local, el cual se encarga de conectarse con los brokers de las otras imágenes de la red. De esta

forma, una imagen puede conectarse con cualquier otra imagen por medio de su broker local. A nivel de la red, la conexión se realiza por medio de sockets, pudiendo utilizar tanto el protocolo TCP/IP como el UDP. Notar que, como se indica en la figura B.1, no importa la forma física de conexión de la red (puede ser, por ejemplo, wireless), siempre y cuando haya conexión por sockets.

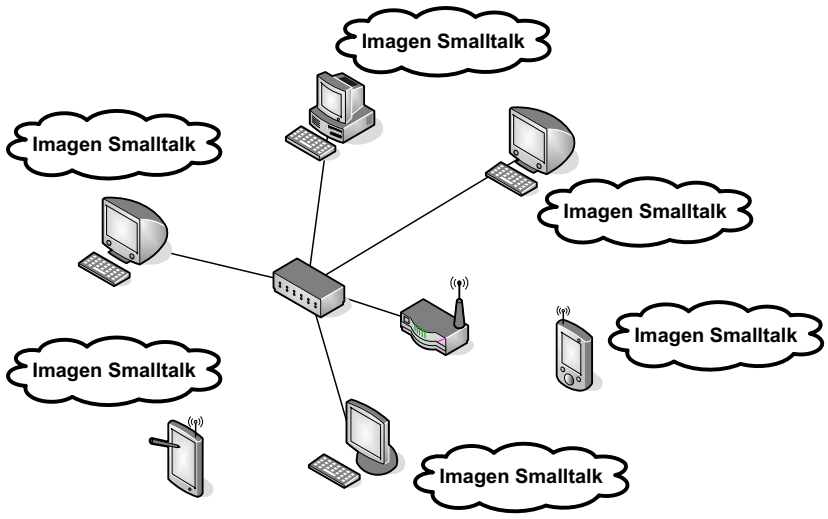

Figura B.1: Red de imágenes Smalltalk

B.2 Técnicas de distribución

Para distribuir objetos, Opentalk brinda dos estrategias básicas: distribución por *referencia* y distribución por *copia*.

La distribución por referencia se realiza utilizando proxies [25] a objetos remotos. Al utilizar esta técnica, el objeto "real" se encuentra viviendo en una imagen de Smalltalk, mientras que el resto de las imágenes sólo ven a un sustituto de éste. Al enviarle un mensaje al sustituto, éste se comunica mediante el broker local con el broker de la imagen en la que se encuentra el objeto real, redireccionándole el mensaje y esperando por una respuesta. El objeto real recibe el mensaje, activa el método asociado y retorna el objeto resultado, el cual viaja a través de la red hasta la imagen del sustituto. De esta forma, el objeto que envía el mensaje no tiene forma de distinguir si está trabajando con un sustituto o con el objeto real, ya que ambos responden a los mismos mensajes de la misma forma.

Una segunda forma de distribuir objetos es por copia. Al distribuir objetos por copia, en cada imagen reside una **copia** del objeto original, lo que implica que los mensajes enviados a ese objetos se resuelven en forma local. El problema que presenta esta técnica es que no hay garantía que un objeto y su copia se mantengan sincronizados. Al realizar una copia se crea un nuevo objeto, conteniendo sus colaboradores internos propios; en el caso que el objeto original cambie alguno de sus colaboradores el objeto copia quedará desactualizado. En forma análoga, si el resultado de enviarle un mensaje a un objeto copia es que éste reemplace un colaborador por otro, el objeto original y su copia conocerán colaboradores distintos, por lo que es de esperar que su comportamiento sea distinto.

B.3 Extensiones al framework

A pesar que Opentalk es un framework maduro y con años de desarrollo, no posee todas las características que necesitamos para nuestro sistema. Una funcionalidad que fue necesario agregarle al framework es la posibilidad de migrar objetos. Como se mencionó con anterioridad, en el caso de trabajar con objetos distribuidos por referencia, existe un objeto real y un conjunto de sustitutos que "miran" al objeto real. En estos casos puede darse que por algún motivo la imagen en la que residen objetos reales deba ser apagada. Para que el sistema siga funcionando, el framework debería "mover" los objetos reales a otra imagen y actualizar todas las referencias remotas. Para solucionar este problema se extendió el framework para que soporte la migración de objetos reales entre imágenes.

Notar que este agregado nos permite también soportar en forma transparente la movilidad del usuario: supongamos que un usuario se encuentra trabajando con una aplicación en su PC de escritorio y debe salir al aeropuerto a tomar una avión; por medio del mecanismo de migración puede trasladar la aplicación que está ejecutando a su PDA y seguir trabajando en ésta durante su viaje. De hecho, al llegar a su destino el usuario puede volver a migrar la aplicación a la computadora de escritorio (o notebook) que le asignen sin que el resto de los usuarios siquiera se percaten de las transiciones de hardware y sistemas operativos. Esta extensión ya ha sido implementada y está en etapa de prueba.

Otro de las falencias del framework viene dada por la falta de un mecanismo alternativo de distribución. La distribución por referencia tiene la ventaja que no existen problemas de sincronismo, ya que todos los mensajes son recibidos por el objeto real; en contrapartida, el envío de mensajes genera un gran tráfico de red, haciendo que la performance de la aplicación decaiga en varios órdenes de magnitud. En el caso de distribuir por copia no tenemos este problema, ya que todos los mensajes se resuelven en forma local. La desventaja que presenta es

que no hay forma de mantener a los objetos sincronizados. Para solucionar ésto, estamos trabajando en una extensión al framework que permita trabajar con copias sincronizadas automáticamente. Esta técnica alternativa de distribución se basa en la existencia de un objeto original y un conjunto de objetos copias (*mirrors*), que se mantienen sincronizados respecto del original. Esto significa que, en un ambiente distribuido, una imagen contendrá al objeto original y el resto, objetos mirror. El envío de un mensaje a un objeto se resuelve en forma local, ya que en todas las imágenes hay, o bien objetos reales, o bien mirrors. La única excepción a esta regla está dada por el caso en que el mensaje implique reemplazar un colaborador por otro (esto es, un cambio en el estado interno del objeto). En este caso, antes de realizar el cambio, el objeto original bloquea a todas las imágenes que contienen objetos mirror, realiza la modificación en el estado interno del objeto y luego libera a las imágenes del bloqueo para que continúen con su funcionamiento normal.

La estrategia de distribución por copia sincronizada resulta favorable si la cantidad de modificaciones es baja en comparación con el resto de los mensajes enviados. En el caso que no sea así, es mejor utilizar distribución por proxies. Es importante notar que este tipo de cosas no pueden saberse antes de realizar la aplicación y deben ser ajustadas en base a sus características particulares.

Apéndice C

Mecanismos de Sensado

Para detectar la ubicación de objetos móviles se debe utilizar algún tipo de dispositivo de posicionamiento. Estos mecanismos aún no se encuentran integrados a los dispositivos móviles como PDAs o laptops, lo que implica la adquisición de un sistema externo.

En los últimos tiempos se han desarrollado una importante gama de mecanismos para determinar la posición de objetos, que no sólo varían en el ambiente de utilización (posicionamiento en interiores o al aire libre) sino también en precisión, rango de cobertura, comodidad de instalación y costos. Dado que las pruebas se han realizado en ambientes cerrados, en el presente apéndice se muestra un breve resumen de los dispositivos disponibles para posicionamiento en interiores.

C.1 RFID

C.1.1 Descripción

Una alternativa muy utilizada es la identificación por Radio Frecuencia, empleada frecuentemente en grandes almacenes y fábricas para controlar el flujo de productos. Existen muchas variantes de esta tecnología, pero la idea básica es colocar etiquetas (tags) sobre los objetos que se quiere controlar. Estas etiquetas son detectadas por lectores, que se colocan principalmente en aberturas o lugares específicos por donde los productos transitan. Los tags RFID pueden ser de dos tipos: pasivos o activos. Los primeros contienen antenas que, al ser alcanzadas por las ondas emitidas por el lector RFID, generan un campo magnético con el que adquieren la energía necesaria para enviar su identificador único al lector. Al no requerir energía son considerablemente más baratos que

los activos, pero alcanzan distancias cortas (a lo sumo 3 metros). Los tags activos llevan su propia fuente de energía para transmitir su señal, lo que permite alcanzar distancias mucho mayores (más de 30 metros).

C.1.2 Instalación

En el caso de trabajar con tags pasivos, la instalación debe ser muy cuidadosa, ya que el rango de alcance se encuentra muy acotado. Generalmente los tags son colocados en los productos o en algún tipo de credencial para que las personas los transporten, al tiempo que los lectores se suelen colocar en aberturas o lugares por donde el sistema debe captar que el tag pase.

C.1.3 Costos

El costo de un tag pasivo se encuentra en el orden de los centavos de dólar, al tiempo que un tag activo cuesta alrededor de 50 dólares. Por su parte, los lectores varían mucho de precio, pero pueden conseguirse a partir de los 300 dólares. Por estos motivos, este tipo de mecanismo resulta adecuado en el caso de requerir uno o dos lectores y una gran cantidad de tags. Lamentablemente este no es el caso de las aplicaciones location-aware, ya que se deberían colocar lectores en todas las aberturas de cada habitación.

Entre las empresas que ofrecen tecnología RFID se pueden mencionar RF Code (www.rfcode.com) (que ofrece un sistema de tags activos con el que promete 500 metros de cobertura en campos abiertos), RightTag (www.righttag.com) y Brooks (www.ready4rfid.com) que ofrecen sistemas con tags pasivos.

C.2 Beacons

C.2.1 Descripción

Los beacons son dispositivos pequeños y sencillos que se encuentran emitiendo constantemente un identificador único. Para proveer información de posicionamiento a los dispositivos móviles, se pueden colocar beacons en cada área de interés. De esta forma, al recibir el identificador de un beacon, el sistema puede relacionarlo con un área geográfica y determinar en qué lugar se encuentra parado el usuario. De hecho, en el caso de recibir más de una señal, la posición del usuario será el resultado de realizar la intersección de las áreas de cobertura de cada beacon.

El tipo de señal a utilizar por los beacons depende de diferentes factores, como el área de alcance, características de los equipos móviles receptores, costos y preferencias de instalación.

Los tres tipos de beacons más utilizados son:

Infrarrojo. Generalmente, los dispositivos móviles (PDAs, laptops, smartphones) llevan instalado un lector de señales infrarrojo. Por lo tanto, una opción viable es utilizar un beacon que emita en la gama del infrarrojo, ya que no es necesario adquirir un lector por separado. A pesar de su conveniencia en términos de tamaño y costo, trabajar con infrarrojo tiene algunas desventajas: en primer lugar las señales infrarrojas a baja frecuencia son débiles y fáciles de obstaculizar. Si bien esto puede ser útil en algunas ocasiones (por ejemplo si no queremos que traspasen paredes para sensar ámbitos cerrados), representa un problema, ya que al colocar cualquier elemento entre el emisor y el receptor se pierde la conexión. El segundo problema es que el área de cobertura es muy limitada y exige que los lectores sean apuntados directamente al beacon. Esto implica que, para lograr una lectura satisfactoria, el usuario debería apuntar explícitamente su dispositivo al beacon, situación que resulta por demás incómoda.

Bluetooth. La tecnología Bluetooth es un estándar cada vez más utilizado. Cualquier dispositivo móvil nuevo, ya sea computadora de mano o teléfono celular, viene equipado con esta tecnología. Asimismo, presenta un conjunto de ventajas importantes respecto del infrarrojo: la señal que emite un dispositivo Bluetooth no es interrumpida fácilmente por la presencia de obstáculos. Esto no sólo significa que el usuario no debe apuntar su dispositivo al beacon, sino que puede llevarlo guardado en un bolsillo que igual captará la señal. Asimismo, si bien fue creada para conexiones a distancias cortas, pueden conseguirse dispositivos que lleguen a los 100 metros de alcance. A pesar que los beacons bluetooth son difíciles de conseguir, se pueden utilizar unos dispositivos llamados Bluetooth Dongles para reemplazarlos. Estos dispositivos, al conectarse al puerto USB de una PC, la extienden para proveer conectividad bluetooth. Esto implica que la PC estará emitiendo, a intervalos de tiempo regular, una baliza con la dirección MAC de la placa de red de la PC. Esta característica puede ser aprovechada para emplearlos como beacons y así determinar la posición de la persona.

Radiofrecuencia. Brindan las mismas ventajas de la tecnología Bluetooth e incluso logran un mayor rango de alcance. La principal desventaja de utilizar este tipo de beacons es que los dispositivos móviles no suelen tener lectores de radiofrecuencia, lo que implica que hay que adquirir lectores adicionales para cada equipo.

C.2.2 Instalación

La instalación de beacons es relativamente sencilla; el único punto crítico es asegurarse que el radio de alcance del beacon abarque el área deseada. En el caso de utilizar radiofrecuencia, se debe tener en cuenta la instalación del lector en cada dispositivo móvil.

C.2.3 Costos

Un beacon infrarrojo básico (sólo emisión) puede costar entre 10 y 30 dólares y puede ser fabricado en serie en la Argentina por un costo menor. Los beacons bluetooth son difíciles de conseguir en el mercado. La empresa BlipSystems (www.blipsytsems.com) ofrece un diseño a 50 euros y BlueLon (www.bluelon.com) ofrece un beacon receptor/emisor, por 99 euros. Como se indicó anteriormente, la alternativa de utilizar Bluetooth Dongles es mucho más viable. De hecho, se pueden adquirir dos clases de dongles según los requerimientos de alcance:

- Clase 1: alcance de 100 metros aproximadamente en campo abierto. Cuestan alrededor de 40 dólares.
- Clase 2: alcance de 20 metros aproximadamente en campo abierto. Cuestan alrededor de 25 dólares.

C.3 RSSI - Triangulación

C.3.1 Descripción

Esta técnica se basa en establecer la localización del dispositivo móvil en base a su cercanía con los Access Points (APs) de una WLAN. Para determinar esa distancia se observa la potencia de la señal (RSSI: Received Signal Strength Indicator) que el dispositivo recibe de los APs cercanos y se estima su posición aplicando un proceso de triangulación. A pesar que la idea es atractiva, un problema importante surge al asumir que la potencia de la señal recibida es directamente proporcional a la distancia entre el AP y el dispositivo. Muchas veces la potencia de la señal disminuye por factores como obstáculos (paredes, tabiques, etc) o la orientación de la antena del emisor. Para corregir este problema se desarrolló un método llamado *fingerprinting*, que consiste en dividir el espacio sensado en celdas y observar el índice RSSI en cada una de ellas. De esta manera son tenidas en cuenta las atenuaciones de señal provocadas por factores no deseados. Existen empresas como Airespace (www.airespace.com) o Aruba Networks (www.arubanetworks.com) que ofrecen este servicio.

C.3.2 Instalación

Para lograr la triangulación se requiere que los dispositivos móviles siempre reciban señal de al menos 3 APs. Para esto se requiere colocar los APs a una distancia de aproximadamente 20 metros (60 pies) entre sí. En el caso que se desee mejorar la precisión mediante fingerprinting u otra técnica similar, hay que tener en cuenta el trabajo estadístico previo al uso del sistema y el ajuste fino del sistema.

C.3.3 Costos

Una de los puntos importantes que que la mayoría de los dispositivos modernos están equipados con placas WiFi, por lo que el costo es simplemente el de los Access Points (o adaptadores de red que funcionen como tales) que se requieran. El problema es que un AP cuesta en promedio 200 dólares, lo que implica que al menos se debe realizar una inversión de 600 dólares.

Bibliografía

[1] Jakob E. Bardram. The java context awareness framework (jcaf) - a service infrastructure and programming framework for context-aware applications. In *Pervasive*, pages 98–115, 2005.

[2] Louise Barkhuus and Paul Dourish. Everyday encounters with context-aware computing in a campus environment. In *Ubicomp*, pages 232–249, 2004.

[3] L. W. Barsalou. The content and organization of autobiographical memories. In *Remembering reconsidered: Ecological and traditional approaches to the study of memory.*, pages 193–243, 1988.

[4] Benjamin B. Bederson. Audio augmented reality: a prototype automated tour guide. In *CHI 95 Conference Companion*, pages 210–211, 1995.

[5] M. Brown. Supporting user mobility. In *IFIP World Conference on Mobile Communications*, pages 69–77, 1996.

[6] P. J. Brown. The Stick-e document: A framework for creating context-aware applications. *j-EPODD*, 8(2/3):259–272, June/September 1995.

[7] P. J. Brown, J. D. Bovey, and X. Chen. Context-aware applications: from the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[9] Deborah Caswell and Philippe Debaty. Creating web representations for places. In *HUC*, pages 114–126, 2000.

[10] Cincom. Opentalk protocol layer developer's guide, 2003.

[11] Joëlle Coutaz and Gaëtan Rey. Foundations for a theory of contextors. In *CADUI*, pages 13–34, 2002.

[12] N. Davies, K. Mitchell, K. Cheverst, and G. Blair. Developing a context sensitive tourist guide, 1998.

[13] A. Dey, G. Abowd, and D. Salber. A context-based infrastructure for smart environments, 1999.

[14] A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications, 2001.

[15] Anind K. Dey. Context-aware computing: The cyberdesk project. In *AAAI 1998 Spring Symposium on Intelligent Environments (AAAI Technical Report SS-98-02)*, pages 23–25, 1998.

[16] Anind K. Dey, Gregory D. Abowd, and Andrew Wood. Cyberdesk: A framework for providing self-integrating context-aware services. In *Intelligent User Interfaces*, pages 47–54, 1998.

[17] Anind K. Dey, Daniel Salber, Gregory D. Abowd, and Masayasu Futakawa. The conference assistant: Combining context-awareness with wearable computing. In *ISWC*, pages 21–28, 1999.

[18] Anind Kumar Dey. *Providing architectural support for building context-aware applications.* PhD thesis, Georgia Institute of Technology, 2000. Director-Gregory D. Abowd.

[19] Paul Dourish. What we talk about when we talk about context. *Personal and Ubiquitous Computing*, 8(1):19–30, 2004.

[20] Richard Draves, BrianÑ. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *SOSP*, pages 122–136, 1991.

[21] P. Fahy and S. Clarke. Cass - middleware for mobile context-aware applications, Mobisys 2004, Workshop on Context Awareness, 2004.

[22] Steven Feiner, Blair MacIntyre, Tobias Hollerer, and Anthony Webster. A touring machine: Prototyping 3d mobile augmented reality systems for exploring the urban environment. In *ISWC '97: Proceedings of the 1st IEEE International Symposium on Wearable Computers*, page 74, Washington, DC, USA, 1997. IEEE Computer Society.

[23] A. Fitzpatrick, G. Biegel, S. Clarke, and V. Cahill. Towards a sentient object model, 2002.

[24] D. Franklin and J. Flachsbart. All gadget and no representation makes jack a dull environment. In *Proceedings of AAAI 1998 Spring Symposium on Intelligent Environments*, 1998. AAAI TR SS-98-02.

[25] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[26] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[27] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowl. Acquis.*, 5(2):199–220, 1993.

[28] Tao Gu, H. C. Qian, J. K. Yao, and H. K. Pung. An architecture for flexible service discovery in octopus. In *12th International Conference on Computer Communications and Networks (ICCCN)*, Dallas, Texas, 2003.

[29] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. A middleware for context-aware mobile services. *IEEE Vehicular Technology Conference*, 2004.

[30] Jennifer Healey and Rosalind W. Picard. Startlecam: A cybernetic wearable camera. In *ISWC*, pages 42–49, 1998.

[31] Karen Henricksen and Jadwiga Indulska. A software engineering framework for context-aware pervasive computing. In *PerCom*, pages 77–86, 2004.

[32] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Pervasive*, pages 167–180, 2002.

[33] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger, Josef Altmann, and Werner Retschitzegger. Context-awareness on mobile devices - the hydrogen approach. In *HICSS*, page 292, 2003.

[34] Scott E. Hudson and Ian E. Smith. Electronic mail previews using non-speech audio. In *CHI Conference Companion*, pages 237–238, 1996.

[35] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. Ip-based protocols for mobile internetworking. In *SIGCOMM*, pages 235–245, 1991.

[36] Hiroshi Ishii and Brygg Ullmer. Tangible bits: Towards seamless interfaces between people, bits and atoms. In *CHI*, pages 234–241, 1997.

[37] Panu Korpipää, Jani Mäntyjärvi, Juha Kela, Heikki Keränen, and Esko-Juhani Malm. Managing context information in mobile devices. *IEEE Pervasive Computing*, 2(3):42–51, 2003.

[38] Gerd Kortuem, Zary Segall, and Martin Bauer. Context-aware, adaptive wearable computers as remote interfaces to 'intelligent' environments. In *ISWC*, pages 58–65, 1998.

[39] M. Lamming and M. Flynn. Forget-me-not: intimate computing in support of human memory. In *Proceedings FRIEND21 Symposium on Next Generation Human Interfaces*, 1994.

[40] U. Leonhardt. *Supporting Location-Awareness in Open Distributed Systems.* PhD thesis, Dept. of Computing, Imperial College, 1998.

[41] Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid prototyping of mobile context-aware applications: The cyberguide case study. In *MOBICOM*, pages 97–107, 1996.

[42] Elizabeth D. Mynatt, Maribeth Back, Roy Want, Michael Baer, and Jason B. Ellis. Designing audio aura. In *CHI*, pages 566–573, 1998.

[43] Jason Pascoe. Adding generic contextual capabilities to wearable computers. In *ISWC*, pages 92–99, 1998.

[44] Pinpoint 3d-id.

[45] Salil Pradhan. Semantic location. *Personal and Ubiquitous Computing*, 4(4):213–216, 2000.

[46] Bradley J. Rhodes. The wearable remembrance agent: A system for augmented memory. In *Proceedings of The First International Symposium on Wearable Computers (ISWC '97)*, pages 123–128, Cambridge, Mass., USA, 1997.

[47] Don Roberts and Ralph E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3.* Addison Wesley, 1997.

[48] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[49] Bill Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, US, 1994.

[50] Bill Schilit and M. Theimer. Disseminating active map information to mobile hosts. *IEEE Network*, 8(5):22–32, 1994.

[51] BillÑ. Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, 1995.

[52] BillÑ. Schilit, Norman Adams, Rich Gold, Michael M. Tso, and Roy Want. The parctab mobile computing system. In *Workshop on Workstation Operating Systems*, pages 34–39, 1993.

[53] Yasuyuki Sumi, Tameyuki Etani, Sidney Fels, Nicolas Simonet, Kaoru Kobayashi, and Kenji Mase. C-map: Building a context-aware mobile assistant for exhibition tours. In *Community Computing and Support Systems*, pages 137–154, 1998.

[54] A. Ward, A. Jones, and A. Hopper. A new location technique for the active office, 1997.

[55] Mark Weiser. The computer for the 21st century. *Human-computer interaction: toward the year 2000*, pages 933–940, 1995.

[56] Mark Weiser and John Seely Brown. The coming age of calm technolgy. *Beyond calculation: the next fifty years*, pages 75–85, 1997.

[57] Tim O'Shea William W. Gaver, Raoul N. Smith. Effective sounds in complex systems: The arkola simulation. In Judith S. Olson Scott P. Robertson, Gary M. Olson, editor, *Proceedings of the ACM CHI 91 Human Factors in Computing Systems Conference*, pages 85–90. ACM Press, 1991.

[58] Bobby Woolf. Null object. *Pattern languages of program design 3*, pages 5–18, 1997.

[59] Jie Yang, Weiyi Yang, Matthias Denecke, and Alex Waibel. Smart sight: A tourist assistant system. In *ISWC '99: Proceedings of the 3rd IEEE International Symposium on Wearable Computers*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.