View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

Arquitectura Orientada a Objetos para Aplicaciones Sensibles al Contexto

Tesis presentada para obtener el grado de Magíster en Ingeniería de Software

Facultad de Informática
Universidad Nacional de La Plata

A.C. Javier Bazzocco
javier.bazzocco@lifia.info.unlp.edu.ar

Director: Dra. Silvia Gordillo

La Plata - Buenos Aires
Argentina
Agosto de 2005

Dedicatoria

Este trabajo está dedicado a

- Mis padres Dorita y Chiqui, los cuales representan mis raíces, que han sabido transmitirme sus valores, amor y respeto por el trabajo.
- Mi presente, constituido por Gisela , a quien agradezco profundamente su amor, constante apoyo y confianza; a Silvia Gordillo, Gustavo Rossi y Máximo Prieto por haberme inculcado las bases y los valores profesionales que rigen mi trabajo diariamente y a todos los que de una manera u otra han colaborado en mi formación profesional.
- Mi futuro: Sofía, quien es el motor que me impulsa constantemente a ser un mejor profesional y por sobre todo, mejor persona.

Índice

Dedicatoria ... 3
Índice ... 5
Índice de figuras ... 7
Capítulo I: Introducción ... 9
Capítulo II: Trabajos previos ... 17
2.1 Introducción ... 17
2.2 Conclusiones ... 24
Capítulo III: Arquitectura para aplicaciones móviles multi-capas ... 27
3.1 Introducción a los dispositivos móviles ... 27
3.2 Las aplicaciones ... 31
3.3 Requerimientos generales ... 34
3.4 Capas de la arquitectura ... 38
3.5 Diagrama de clases ... 43
3.5.1 Diseño de clases de los mensajes ... 43
3.5.2 Diseño de clases de los codificadores de mensajes ... 45
3.5.3 Diseño de clases de los decodificadores de mensajes ... 47
3.5.4 Diseño de la capa de comunicaciones del cliente ... 48
3.5.5 Diseño de la capa de comunicación del servidor ... 52
3.6 Conclusiones ... 55
Capítulo IV: Framework O.O. para soportar Context-Awareness ... 59
4.1 Introducción ... 59
4.2 Especificación de Contextos ... 59
4.3 Framework para aplicaciones "context-aware" ... 64
4.4 Diagrama de colaboración ... 69
4.5 Diagrama de clases ... 70
4.6 Diagrama de Secuencia ... 71
4.7 Interacción entre los elementos del framework y las aplicaciones ... 72
4.7.1 Registración de las aplicaciones ... 72
4.8 Conclusiones ... 74
Capítulo V: Implementación de un prototipo: CAWI ... 75
5.1 Introducción ... 75
5.2 ITeM ... 76
5.3 Servidor CAWI ... 80
5.4 Simulador ... 86
5.5 Aplicación móvil ... 89
5.6 Conclusiones ... 99
Capítulo VI: Beneficios de la integración de context-awareness a la Ingeniería de Software ... 101
6.1 Introducción ... 101
6.2 Áreas de aplicación ... 102
6.3 Conclusiones ... 105
Capítulo VII: Trabajo futuro ... 107

7.1 Introducción 107
7.2 Violación de la privacidad 107
7.3 Pruebas de campo "reales" 108
7.4 Interoperabilidad 109
7.5 Robustez 110
7.6 Performance y optimización 110
Capítulo VIII: Conclusiones 113
Bibliografía 117
Anexo I: Patrones de diseño 121
Abstract Factory 121
Composite 121
Command 121
Decorator 121
Facade 122
Mediator 122
Observer 122
Proxy 122
Singleton 122
Strategy 122
Template Method 123
Anexo II: Tecnologías del Prototipo desarrollado 125
Http (HyperText Transfer Protocol) 125
Java 125
Hibernate 126
Jakarta Tomcat 127
MySQL 127
Xml (eXtensible Markup Language) 127
Servlets 128

Índice de figuras

Figura 1: Discontinuidad en los servicios accedidos. 29
Figura 2: Dispositivos móviles. 30
Figura 3: Otros tipos de dispositivos móviles. 31
Figura 4: Categorización de aplicaciones móviles "context-aware". 34
Figura 5: Diagrama de paquetes de la arquitectura 38
Figura 6: Principales componentes de la arquitectura. 39
Figura 7: Diagrama de clases de los mensajes. 43
Figura 8: Diagrama de clases de los codificadores de mensajes. 45
Figura 9: Diagrama de clases de los decodificadores de mensajes. 47
Figura 10: Diagrama de clases de la capa de comunicación en el cliente. 49
Figura 11: Diagrama de clases de la capa de comunicación del servidor. 53
Figura 12: Jerarquía de clases de contextos. 61
Figura 13: Principales elementos del framework. 65
Figura 14: Diagrama de colaboración del framework. 69
Figura 15: Diagrama de clases del framework. 70
Figura 16: Diagrama de secuencia de una notificación. 72
Figura 17: Visualización de la lista de requerimientos en el ITeM. 78
Figura 18: Modelo del servidor CAWI. 82
Figura 19: Interfase gráfica del simulador del prototipo. 88
Figura 20: Interfase gráfica para dar de alta habitaciones. 88
Figura 21: Aplicación móvil preparada para iniciar su ejecución. 90
Figura 22: Listado de proyectos de un usuario 91
Figura 23: Pantalla de resumen del estado de un proyecto. 92
Figura 24: Diagrama de clases del modelo de la aplicación móvil. 93
Figura 25: Diagrama de clases de la capa de comunicaciones móvil. 95
Figura 26: Diagrama de clases de las interfases gráficas y sus comandos. 95
Figura 27: Diagrama de clases de la aplicación móvil. 98

Capítulo I: Introducción

Desde los inicios de la informática como ciencia, los lenguajes de programación que rigen y establecen la forma de comunicación entre los seres humanos y las computadoras se han basado en un esquema en el cual se debe definir de manera explícita lo que se desea que las computadores realicen para evitar ambigüedades. Si bien este hecho es reconocido como sumamente beneficioso, este esquema evita que las aplicaciones puedan utilizar información implícita del contexto en que se desenvuelven como lo hacen habitualmente los humanos.

En los últimos años se ha comenzado a estudiar y analizar nuevas herramientas para llevar a la interfase hombre-máquina a un nuevo nivel. Entre dichas herramientas se destaca entre otras posibilidades, la capacidad por parte de las aplicaciones de hacer uso de información de carácter implícito, es decir no brindada explícitamente (o directamente) por el usuario, para proveer un mejor servicio o adecuarse a los recursos existentes en un momento determinado.

Este tipo de aplicaciones en general se basa en sensores (posiblemente remotos) y otros dispositivos electrónicos para capturar información relevante del medio ambiente en donde se están ejecutando. Una vez capturada, analizada y establecida qué información resulta relevante, la aplicación se adapta de acuerdo a reglas preestablecidas, con el fin de aumentar su utilidad para el usuario. Consideremos algunos ejemplos simples para establecer claramente a qué tipos de aplicaciones nos referimos:

- Usted se encuentra sentado en una sala junto a otros altos ejecutivos en medio de una importante reunión. De repente su teléfono celular comienza a sonar. Presumiblemente debido a los

nervios, Ud. no encuentra los botones para desactivar rápidamente el sonido y termina apagando el dispositivo, sin conocer realmente la importancia del mensaje. ¿No sería acaso mucho mejor que el teléfono celular (o alguna aplicación que esté siendo ejecutada en el mismo) pudiese reconocer el lugar en el que Ud. se encuentra y en caso de no tratarse de un llamado importante cambiar la melodía por un mensaje con un simple "bip" o incluso alguna indicación de carácter gráfica que tiene un mensaje sin leer?

- Supongamos ahora que Ud. es una persona que debe llevar ciertos documentos impresos en forma personal a una oficina distante. Imagínese ahora poder ir revisando los documentos en su PDA[1] (Personal Digital Assistant), despreocupado de cualquier cuestión técnica, seguro de que cuando se esté aproximando a la sala de reunión, su PDA lo alertará de la presencia de impresoras listas para imprimir su trabajo, teniendo además en cuenta su prisa a modo de criterio de priorización (ante la posibilidad de la existencia de múltiples trabajos encolados para su impresión), de modo de tener su trabajo listo en forma inmediata apenas Ud. es detectado en las inmediaciones de la sala de reuniones.

- Generalmente cuando uno se encuentra de visita en una ciudad que no conoce, es sumamente fácil desorientarse y uno inevitablemente termina pidiendo ayuda a algún transeúnte. Ahora imagínese contar con una PDA cargada con una aplicación capaz de recomendarle el mejor restaurante de la zona teniendo en cuenta elementos tan disímiles (a primera vista) como :

[1] Dispositivo móvil con capacidades reducidas de procesamiento y presentación. Como ejemplo de este tipo de dispositivos considérese una Palm Pilot. Para más referencias dirigirse a http://www.palm.com.

- Su ubicación relativa.
- El horario local y las costumbres gastronómicas del lugar.
- El rango de precio o calidad del restaurante.
- La cercanía respecto de otros ítems de su recorrido (por ejemplo: museos o lugares turísticos para los cuales Ud. ya tiene una reserva hecha).
- La cantidad de colectivos, taxis u otros medios que debe tomar para llegar a destino (pudiendo variar los criterios utilizados en la selección para considerar aspectos relacionados con el tiempo de viaje, costo de los boletos, etc.).

En cada uno de los ejemplos anteriores, se han citado algunas de las características que definen claramente a este tipo de aplicaciones, que de una manera directa (Ud. define el rango de precios para la consulta de restaurantes) u indirecta (mediante la utilización de un GPS [2] el dispositivo logra conocer su ubicación en términos de latitud, longitud y altura) se adaptan al medio ambiente para poder proveerle un servicio con un valor agregado importante.

Un aspecto importante a considerar es "cuanta" noción de la información necesaria para la aplicación tiene el usuario final. En otras palabras, el usuario debería tener la menor conciencia posible de las necesidades de información acerca del contexto que las aplicaciones requieren, razón por la cual las aplicaciones deberían contar con medios lo más automatizados posible de reconocimiento de información contextual.

Hoy en día resulta poco común encontrar una definición a nivel arquitectónico a la cual apegarse mientras se desarrollan aplicaciones móviles,

[2] GPS: Global Positioning System, sistema mediante el cual se puede ubicar una entidad sobre la superficie de la tierra. Para esto se hace uso de 3 o 4 satélites, los cuales establecen la posición de la entidad en coordenadas de latitud, longitud y altura. Estos dispositivos son utilizados con asiduidad en el área de Sistemas de Información Geográfica.

razón por la cual esta tesis aporta soluciones de diseño y arquitectura claras y precisas a fin de facilitar la tarea a los desarrolladores y diseñadores. Asimismo, esta tesis presenta un refinamiento posterior para soportar la definición y construcción de aplicaciones que también pueden ser clasificadas como sensibles al contexto. Dicho refinamiento toma la forma de framework orientado a objetos, haciendo hincapié en las particularidades de las aplicaciones sensibles al contexto. Este es un aporte importante de este trabajo en esta área, ya que no es habitual encontrar soluciones orientadas a objetos para este tipo de aplicaciones.

Es notorio el poco interés que ha despertado en la comunidad científica hasta el momento el área de la Ingeniería de Software como área de aplicación de la sensibilidad al contexto. Este trabajo entonces introduce dicha área como área de aplicación de la sensibilidad al contexto, enfocándose en particular en las actividades relacionadas con la administración de los requerimientos y pedidos de cambios que se dan durante el ciclo de vida de un proyecto de software.

Finalmente se presenta como parte de este trabajo un prototipo desarrollado con tecnologías que actualmente se están utilizando en el mercado para demostrar las ventajas que posee este framework al ser utilizado como guía y/o base para la implementación de aplicaciones móviles sensibles al contexto.

El **Capítulo 2** presenta un breve relevamiento de los trabajos previos existentes en el área de computación "sensible al contexto". Se han seleccionado tres trabajos, cada uno de los cuales ataca un aspecto en particular del tema desde un enfoque tal, que los ha transformado en lectura obligada a lo largo de los años.

Esta tesis abarca dos conceptos importantes: por un lado el relacionado con los dispositivos móviles, los cuales son básicamente dispositivos con ciertas capacidades de cómputo y conexión, que pueden ser llevados por los usuarios más allá de sus estaciones de trabajo; y por otro lado, se hallan las aplicaciones que aumentan sus prestaciones al incorporar información acerca del "contexto" o "ambiente" en el cual se están ejecutando.

Existe un subconjunto de aplicaciones móviles que requieren de acceso a los recursos provistos por un servidor de aplicaciones para brindar su funcionalidad. En estos casos es necesario contar con una arquitectura definida y a la vez lo suficientemente flexible como para acomodar los sucesivos cambios. El **Capítulo 3** presenta una arquitectura que puede ser utilizada efectivamente para el desarrollo de aplicaciones móviles bajo el estándar J2ME [J2ME 05]. El capítulo inicia la discusión mediante la presentación de las características principales de los dispositivos móviles que serán considerados. Luego se discuten los aspectos esenciales de las aplicaciones que son ejecutadas en estos dispositivos para luego sí entrar de lleno en la especificación de la arquitectura conceptual. La arquitectura planteada en este capítulo será tomada como punto de partida para el próximo capítulo, en el cual se presentará la definición de un framework orientada a objetos para soportar aplicaciones móviles que además son sensibles al contexto.

El **Capítulo 4** contiene una descripción detallada de un framework orientado a objetos que da soporte a las necesidades que habitualmente se encuentran al construir aplicaciones "context-aware", que son ejecutadas en dispositivos móviles (aumentando de esta manera las fuentes de incorporación de información contextual) y que se apoyan en un servidor para brindar su funcionalidad. Más adelante se define explícitamente el concepto de "contexto" con el fin de introducir finalmente los detalles del framework propiamente dicho. Nuevamente es necesario destacar que este capítulo se basa fuertemente

en las características básicas presentes en la arquitectura definida en el capítulo anterior.

El **Capítulo 5** abarca una detallada explicación de las diferentes partes componentes de un prototipo desarrollado a fin de obtener experiencia real en el campo de las aplicaciones sensibles al contexto.

En el **Capítulo 6** se presentan los resultados iniciales obtenidos al integrar el concepto de "context-awareness" (y su correspondiente implementación) con una herramienta de seguimiento y control de requerimientos para proyectos de desarrollo de software. La intención de dicha integración es incrementar en primera medida la usabilidad general de sistemas de tales características; y al mismo tiempo realizar una experiencia "real" que demuestre los beneficios y falencias del framework desarrollado.

El **Capítulo 7** presenta algunos temas no resueltos en el presente trabajo y que son de relevancia para los sistemas del tipo "context-awareness". Entre ellos se encuentran los temas de seguridad y privacidad. Este último tema está cobrando particularmente gran auge ya que plantea interrogantes éticos muy importantes respecto de cuanta información es prudente que las aplicaciones adquieran automáticamente sin que el usuario participe directamente.

Las conclusiones de esta tesis son presentadas en el **Capítulo 8**. Éstas abarcan los aspectos teóricos planteados en los capítulos anteriores así como los resultados prácticos obtenidos a partir de la implementación de un prototipo.

En el **Anexo I** se discuten aspectos tecnológicos del prototipo, y se detallan los criterios utilizados en la selección de cada una de las tecnologías utilizadas en el desarrollo de cada capa de la arquitectura.

En el **Anexo II** se presentan brevemente los patrones de diseño utilizados a lo largo de los diferentes capítulos de este trabajo a fin de tener una referencia de sus intenciones.

Capítulo II: Trabajos previos

Diversos autores han trabajado a partir de la década del '90 en el establecimiento de una definición y un consenso general sobre el significado de los "contextos"[3] y de las características fundamentales de las aplicaciones sensibles al mismo.

A continuación se presenta una descripción de los principales trabajos presentados, así como los beneficios, falencias y problemas no solucionados de cada uno de los enfoques propuestos.

2.1 Introducción

Los primeros trabajos en este tema se remontan al año 1994 cuando Schilit [Schilit, B et al. 94] presenta la primera definición reconocida del término "contexto" como: "la ubicación, identidades de las personas y objetos cercanos, así como los cambios a dichos objetos". Dicha definición ha sido considerada como seminal en esta área de estudio, y ha sido tomada como base para otras definiciones más recientes.

Otro autor reconocido, Brown [Brown et. al 97] define al "contexto" de la siguiente manera: "ubicación, identidades de las personas alrededor del usuario, la hora del día, estación del año, temperatura ambiente, etc.". Nótese que esta definición es similar a la definición anterior; sin embargo, ahora también se consideran aspectos del medio ambiente físico (como la temperatura ambiente) y otros tipos de información como por ejemplo la estación del año.

[3] Los términos "sensibles al contexto", "context-aware" y "sensibles al medio ambiente" serán utilizados en forma indistinta a partir de este momento; considerándolas como sinónimos. Sin embargo, es preciso aclarar que en la bibliografía suele considerarse al medio ambiente como un tipo particular de contexto.

Notablemente, algunos autores como Dey [Dey 98] y [Dey 00] incluyen bajo la palabra contexto incluso el estado emocional del usuario al momento de utilizar la aplicación, la cual mediante sensores específicos es capaz de detectar cambios en la retina u otros parámetros biomédicos, identificando de esta manera el estado emocional del usuario. Por ejemplo, este tipo de aplicaciones es capaz de ir reconfigurando su interfase gráfica para ir simplificando la información presentada a medida que el usuario se va fatigando debido a las horas de trabajo.

En el trabajo *"Disseminating Active Map Information To Mobile Hosts"* [Schilit. B et al. 94], publicado el año 1994, si bien no se atacan de manera directa los aspectos críticos y fundamentales de las aplicaciones sensibles al contexto, se presentan por primera vez algunas características que deberían aparecer en toda aplicación que mejore su funcionamiento basándose en detalles del contexto en el cual se está ejecutando.

El objetivo principal del trabajo es presentar las conclusiones a las que arribaron los autores al trabajar con diferentes estrategias para implementar lo que hoy se conoce como "mapas activos". Un mapa activo no es más que un servicio que publica información relevante acerca de los objetos "localizables" en una región determinada. Es importante notar el hecho de que el usuario es quién decide generalmente qué objeto debe ser considerado "localizable". Basándonos en la definición anterior, podemos establecer que un "mapa activo" es un tipo especial de aplicación sensible al contexto, en el cual el contexto de mayor importancia es el contexto dado por la ubicación.

Los autores presentan la siguiente definición de una aplicación "context-aware":

> *"La computación sensible al contexto es la habilidad de una aplicación utilizada por un usuario móvil, de descubrir y reaccionar a los cambios en el ambiente en el cual se encuentra".*

Un aspecto importante tratado en dicho artículo tiene que ver con la manera en la cual las aplicaciones toman conciencia de los cambios del contexto en el que se encuentran. Existen básicamente dos maneras:

- Las aplicaciones contienen uno o más subsistemas especializados en el monitoreo constante de aquellos contextos que presentan o contienen información relevante.

- Las aplicaciones se manejan en un esquema "Publisher & Subscriber" [Cugola et al. 02] en el cual cada aplicación debe registrarse como oyente o "listener" de aquellos contextos de los cuales está interesada en recibir las novedades. Cada contexto es responsable de notificar a todos sus oyentes a medida que se van dando los cambios. El esquema de "Publisher & Subscriber" establece algunas restricciones que deben cumplir los oyentes para que los objetos que actúan de fuente de eventos tengan una manera estándar y bien conocida de notificar los cambios.

A continuación se presentan algunas razones por las cuales las aplicaciones deberían o quisieran suscribirse como oyentes o bien monitorear en forma constante:

- Proveer una representación gráfica de los objetos "localizables".
- Para poder mantener un registro de las personas y otros objetos ya encontrados, de modo de poder soportar aplicaciones basadas en

modelos de recuperación de información que toman en cuenta el tiempo en el cual una tarea fue realizada.

- Para detectar información específica de una ubicación, por ejemplo mensajes electrónicos dejados por un usuario en un espacio determinado para que otro usuario al pasar por dicha ubicación reciba los mensajes.
- Para mantener un listado de los recursos cercanos que podrían ser utilizados en caso de ser necesario.

En definitiva, a modo de conclusión, para Schilit [Schilit. B et. al 94], lo importante es donde uno se encuentra, quienes lo acompañan y que recursos se encuentran cerca.

En el año 1997, P. Brown, J. Bovey y X. Chen publicaron el trabajo "*Context Aware Applications: From the Laboratory to the MarketPlace*" [Brown et al. 97], el cual es considerado como uno de los trabajos fundamentales en el tema de "sensibilidad al contexto o medio ambiente".

Según estos autores, las aplicaciones sensibles al contexto son aquellas que cambian su comportamiento de acuerdo al contexto del usuario. En forma más general, la información puede ser adaptada dependiendo de múltiples aspectos del medio ambiente en el que se encuentra el usuario, por ejemplo su ubicación, hora del día, estación del año, temperatura, y así siguiendo. Resulta evidente que este trabajo extiende la definición anterior, ya que agrega aspectos del medio ambiente físico (como la temperatura ambiental) no considerados en la definición de Schilit [Schilit. B et al. 94].

El adjetivo "sensible al contexto" se utiliza principalmente en aplicaciones motivadas por el contexto. En general este tipo de aplicaciones suelen ser de carácter móvil, debido a que la movilidad tiene como

consecuencia directa un cambio constante y rápido, haciendo que el comportamiento "sensible" sea realmente útil.

Los autores presentan una división de las aplicaciones sensibles al contexto (de una manera al menos "borrosa" o "no estricta"):

- Continuas: en este tipo de aplicaciones, la información presentada al usuario cambia todo el tiempo. Como ejemplo se puede considerar una PDA dotada de un sensor de posición sumamente preciso y también con un compás digital; una aplicación simple podría mostrar una flecha con el sentido del movimiento que debería seguir el usuario para alcanzar un punto determinado (por ejemplo un punto señalado de una tubería en donde se está produciendo una pérdida).

  Es necesario notar que aún hoy en día es sumamente difícil de modelar y de implementar. Por fortuna, la mayoría de las aplicaciones caen dentro de la siguiente categoría.

- Discretas: las aplicaciones que caen bajo esta categoría manejan piezas separadas de información que van unidas a contextos separados (habitaciones, rangos de tiempo o temperaturas, grupos de personas, etc.) y que son mostradas según se ingresa a cada uno de estos contextos.

Luego de definir de manera difusa las cualidades de las aplicaciones sensibles al contexto, los autores brindaron una suerte de definición de un lenguaje (basado en SGML [SGML 95]) que permitiera a las aplicaciones intercambiar información contextual de manera estándar. Dicha definición se

valida mediante un DTD[4], el cual es utilizado principalmente como medio de validación sintáctica de la estructura de un documento de transferencia de información.

En conclusión, los autores se enfocaron en la descripción de un mecanismo estándar de comunicación y representación de la información contextual (estableciendo brevemente las características de las aplicaciones "context-aware") con el objetivo de que la gente que diseñe e implemente este tipo de aplicaciones no necesariamente tenga que tener un "background" tecnológico importante.

En el año 1998, se publica el trabajo "*Context-Aware Computing: The CyberDesk Project*" [Dey 98], que contiene algunos aspectos interesantes no cubiertos en los trabajos anteriores como por ejemplo el estado emocional del usuario como información relevante de contexto para poder alterar el funcionamiento de las aplicaciones. A diferencia de los trabajos anteriormente presentados en esta tesis, Dey se enfoca en la construcción de una arquitectura de software que dinámicamente descubre e integra servicios en base a módulos. Dicha integración es llevada a cabo teniendo en cuenta el contexto del usuario, el cual incluye ambientes tan variados y disímiles como el físico, el social, el emocional e incluso el mental. El trabajo inicial presentado por Dey en [Dey 98] ha sido extendido años más tarde para incorporar especificaciones básicas a nivel de arquitecturas de software para la construcción de este tipo de aplicaciones [Dey 00].

Este autor incluye dentro de la definición de "contexto", información como el estado emocional del usuario, su foco de atención en un momento determinado, ubicación física y orientación, fecha y hora del día, así como el

[4] DTD: Document Type Description. El propósito del DTD es definir los bloques constructivos legales de un documento XML. Define asimismo la estructura de dicho documento con una lista de los elementos considerados "legales". Para más información referirse a [XML].

conjunto de objetos y personas que de manera directa o indirecta están interactuando con el usuario en su medio ambiente.

Además de extender las definiciones previas existentes, se agregan nuevos aspectos tales como la idea de utilizar la información de contexto para predecir las necesidades de ciertos servicios por parte del cliente, lo cual supone un cambio importante en el esquema que se maneja habitualmente en el cual es el usuario mismo el que debe buscar los servicios que desea; según este nuevo esquema, la información contextual es utilizada para inferir cuales serán las necesidades del usuario y en consecuencia alterar el comportamiento de la aplicación para satisfacerlas de una manera más adecuada y oportuna, sin que ello signifique una participación conciente o activa por parte el usuario.

Un punto importante que debe ser recalcado, es que este trabajo se concentró casi exclusivamente en los aspectos relacionados con la construcción del software, apenas nombrando los detalles del hardware necesario para soportar todo el sistema; este hecho se opone a una característica habitualmente hallada en la mayoría de la bibliografía existente sobre este tema, que generalmente se enfoca en las posibilidades tecnológicas del hardware, relegando a un segundo plano el diseño del software.

A continuación se citan brevemente algunas de las capacidades o funcionalidades que provee el sistema desarrollado, y que es capaz de aprovechar la información contextual (mayormente la ubicación del usuario) para mejorar la calidad del servicio:

- Cuando el usuario ingresa a una cocina:
  - Se listan los elementos congelados en el freezer;

  - A la hora de la comida, se crean y muestran las recetas de las comidas que se pueden cocinar teniendo en cuenta los elementos disponibles en la heladera;
  - Generar pedidos electrónicos para reponer los elementos faltantes.

- Cuando el usuario entra en una habitación (el living por ejemplo):
  - Se listan los titulares más importantes del día;
  - Se muestran los principales resultados deportivos;
  - Se filtran y presentan los emails personales.

- Finalmente, cuando el usuario ingresa a su oficina:
  - Se cargan los documentos que se han revisado últimamente;
  - Se verifican los recordatorios relacionados con el trabajo;
  - Se cargan las notas de las últimas reuniones.

Como conclusión, es importante resaltar a la versatilidad conseguida en la arquitectura propuesta en el trabajo analizado y la importancia de enfocarse en los aspectos de diseño del software más que en las características de hardware necesarias para soportar las funcionalidades "extendidas" de las aplicaciones "sensibles al contexto".

2.2 Conclusiones

En los tres principales trabajos analizados se destaca el estudio de las aplicaciones "context-aware" desde el punto de vista de la introducción de una nueva tecnología, e incluso desde lo social y filosófico, pero es evidente que se carece en todo momento de una visión profunda desde el punto de vista del diseño y posterior desarrollo de sistemas en forma estándar y reconocida.

En mayor o menor medida, cada uno de los trabajos presentados se ha enfocado más en los aspectos relacionados con el hardware (PDAs, Teléfonos celulares, Redes inalámbricas, etc.) que en las características del software en sí mismo.

En particular el punto de vista del paradigma orientado a objeto no ha sido casi explorado hasta fines de la década del '90, razón por la cual es entendible que no aparezca como tema importante en los trabajos analizados. Debido a que hoy por hoy estamos "casi" acostumbrados a la constante revolución (más que evolución) de los dispositivos electrónicos, el foco de interés de la comunidad actual es encontrar diferentes maneras de desarrollar este tipo de aplicaciones con todo el "know-how" adquirido en las áreas de ingeniería de software, desarrollo de aplicaciones Web y tecnología de objetos.

La siguiente tabla permite observar las diferencias entre los tres trabajos analizados, estableciendo un conjunto de criterios que serán también utilizados más tarde para comparar esta tesis.

Criterios/Trabajos	Schilit	Brown	Dey
Año	1994	1997	1998
Utiliza el paradigma O.O.	No	No	No
Incluye aspectos de hardware	Sí	Sí	No
Incluye aspectos de software	No	Sí	Sí (principalmente arquitectura)
Basada en estándares	No	No	No
Soporte de Java	No	No	No
Utiliza XML	-	Sí	-
Dispositivos	PDAs	PDAs	PCs
Soporte de procesamiento distribuido	Sí	Sí	Sí
Soporte de múltiples fuentes de información contextual	Sí	Sí	Sí

Tabla 1: Comparación entre los 3 trabajos analizados

Capítulo III: Arquitectura para aplicaciones móviles multi-capas

Una aplicación móvil en general carecerá de la posibilidad de manejar en forma local grandes cantidades de información, razón por la cual será necesario contar con una forma de comunicación con un servidor para resolver las funciones que requieran de grandes cantidades de datos. A continuación se describe una arquitectura para trabajar con este tipo de aplicaciones.

3.1 Introducción a los dispositivos móviles

Como se mencionó en el capítulo anterior, esta tesis apunta a establecer un framework que dé soluciones a los problemas habitualmente hallados en el ámbito de la computación móvil, y en particular aquellos problemas relacionados con las aplicaciones "sensibles al contexto". A raíz de esto, resulta conveniente presentar un breve detalle acerca de los llamados "dispositivos móviles", los cuales son dispositivos con cierto poder de cálculo y capacidades de conexión (a redes de información vía cable o radio), que pueden ser llevados por un usuario mientras se mueve de un lugar a otro. Estos dispositivos pueden pertenecer a diferentes "familias", desde teléfonos celulares, pasando por agendas hasta los denominados PDAs.

Para tomar rápidamente conciencia de la difusión actual que tienen los dispositivos móviles y que son capaces de ejecutar aplicaciones escritas en Java basta visitar [Java-M 05], [Micro 05] o [Wireless 05]. En esto sitios se presentan extensas listas (increíblemente, algunas con más de 130 ítems[5]) que son

[5] Las listas podrían ser aún más largas si se consideran los dispositivos tipo PDAs (Personal Digital Assistants) y Pocket Pcs. Para más información ver [Palm] y [iPAQ]

actualizadas regularmente. En dichas listas además se presentan las capacidades de algunos dispositivos, y cómo puede apreciarse, éstos no están muy lejos del día en que equiparen el poder de cálculo y la capacidad de memoria de una máquina de escritorio.

Sin embargo, a pesar de que este tipo de dispositivos puedan en algún momento ejecutar grandes aplicaciones, es también cierto que están sujetos a ciertas restricciones que dificultan el desarrollo de aplicaciones dirigidas a ellos. Quizás las tres restricciones más importantes tienen que ver con la capacidad de almacenamiento, la calidad de la interfase gráfica y el tipo de conexión con que usualmente trabajan.

La primera restricción citada está relacionada directamente con la cantidad y calidad de las aplicaciones que pueden ejecutarse en estos dispositivos. Al considerar cierto tipo de aplicaciones muy demandantes de información para su operatoria (los clásicos LBS[6]), es obvio que para estos dispositivos no sería viable manejar semejante cantidad de información (tanto por el tiempo que demandaría el envío de la información, como por el tamaño del almacenamiento requerido en el dispositivo).

Si ahora consideramos la segunda restricción, queda claro que estos dispositivos no serán capaces (en el mediano plazo) de presentar tanta información al usuario como sería deseable. Esto se debe simplemente al tamaño que la mayoría de los dispositivos tiene. Si nos referimos nuevamente a las listas anteriormente citadas, el dispositivo que más capacidad gráfica ostenta (aunque impresionante para un dispositivo de su tipo), apenas tiene una resolución de 640 x 200 pixels y una resolución de 12 bits de profundidad. Para tener una magnitud real de cuanto significa esto, 640 x 200 pixels representan

[6] LBS (Location Based Services): servicios que toman en cuenta la ubicación del usuario al momento de ser ejecutados. En general se relacionan con grandes volúmenes de información o sistemas de información geográfica.

un área compuesta de 128000 puntos, mientras que un monitor de 17 pulgadas con una resolución estándar de 1280 x 1024 puntos roza los 1310720 puntos (en otras palabras, el dispositivo móvil cuenta con el 10% de la capacidad gráfica de una máquina de escritorio convencional).

Finalmente la tercera restricción es quizás la más crítica de todas, ya que establece un esquema de programación radicalmente distinto pues a diferencia de las computadoras conectadas a redes locales, estos dispositivos sufren continuamente desconexiones y cambios de proveedores de comunicación. Felizmente, la velocidad de estas conexiones no representa un problema, pero como se citó más arriba, su fiabilidad y robustez sí lo son. Esta última restricción tiene impacto mucho más allá de las aplicaciones que corren en los dispositivos, ya que incluso termina impactando en el modo en el cual los servidores deben interactuar con los clientes móviles. Esto es así ya que los servidores ahora no pueden fiarse de la recepción o no de los datos solicitados por un usuario móvil (con lo cual deben manejar esquemas de transacciones mucho más elaborados), e incluso puede darse el caso de que un servidor deba terminar una tarea que ha sido disparada por un usuario en otro servidor, pero que por cuestiones de movilidad, ahora son responsabilidad del nuevo servidor.

La figura 1 ayuda a comprender más claramente esta situación.

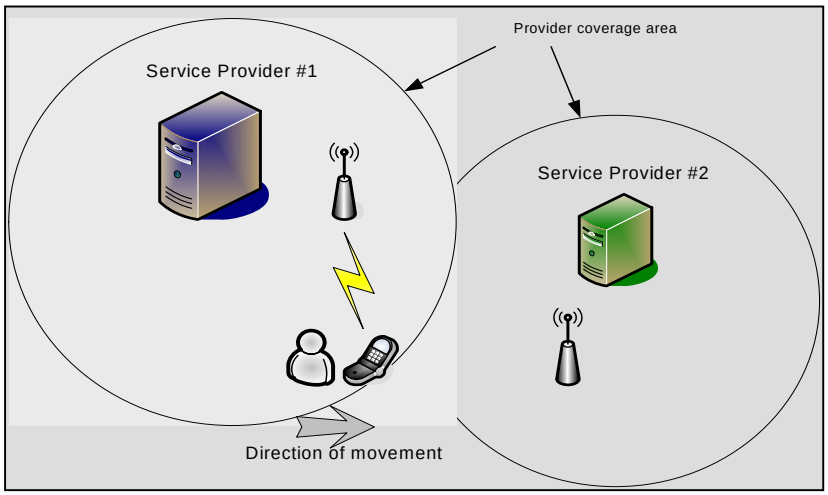

Figura 1: Discontinuidad en los servicios accedidos.

Los problemas hallados habitualmente en el desarrollo de aplicaciones móviles sensibles al contexto no sólo están relacionados con los aspectos físicos (hardware y conexión a redes), existen además problemas relacionados con el diseñó mismo del software. Resulta deseable contar con una manera accesible, mantenible, flexible y estándar para representar la información contextual al momento de desarrollar una nueva aplicación (o posiblemente al migrar a otro dispositivo más moderno). Cuestiones básicas como cuál es la mejor manera de representar al contexto y posteriormente administrarlo (o interactuar con éste) no están aún resueltas del todo. Tampoco existen definiciones completas respecto de la arquitectura de software que debería utilizarse para el desarrollo de este tipo de aplicaciones.

Todo lo citado anteriormente no hace más que exponer un subconjunto de los problemas que habitualmente se enfrentan, y seguirán enfrentando, los desarrolladores de aplicaciones móviles en relación a los recursos disponibles.

A modo de muestra la figura 2 presenta algunos de los dispositivos nombrados en [Java-M 05], [Micro 05] o [Wireless 05].

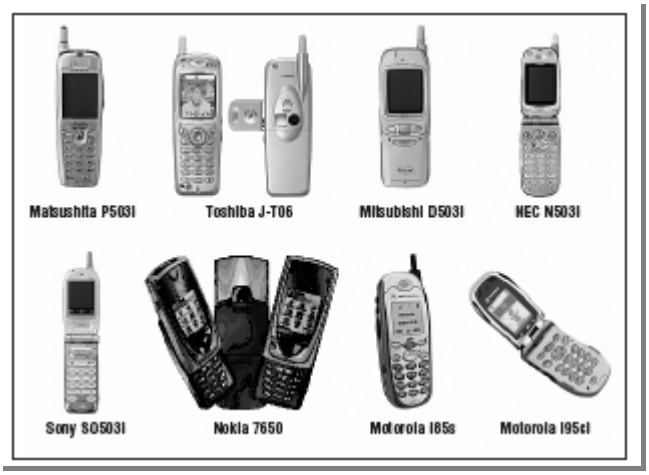

Figura 2: Dispositivos móviles.

La figura 3 por otro lado presenta otra familia de dispositivos, como las PDAs y una Pocket PC (iPAQ).

Figura 3: Otros tipos de dispositivos móviles.

3.2 Las aplicaciones

Al concebir o diseñar una arquitectura, es importante tener en mente el tipo de aplicaciones que serán soportadas. Con ese fin, aquí se presenta una posible categorización de las aplicaciones. Esta categorización se basa en la distinción entre las aplicaciones "autosuficientes" y las que necesitan la colaboración de otros elementos. Es por esto que inicialmente tenemos la siguiente clasificación:

- Aplicaciones "stand-alone": estas aplicaciones constituyen el tipo más común y simple de aplicaciones que se ejecutan en un dispositivo móvil integrando diversas fuentes de información para proveer un valor agregado al usuario. Estas aplicaciones no tienen más requerimientos que aquellos establecidos por sus necesidades de memoria. Son totalmente independientes de grandes cantidades de información. Generalmente requieren la entrada de información mínima. Supongamos por ejemplo una aplicación que dependiendo de la persona que está usando un dispositivo móvil aplica un determinado esquema de colores y resolución de pantalla previamente configurado por el usuario. En este caso, la única información necesaria para la ejecución de la aplicación es la identidad del usuario y su esquema de colores seleccionados. También es necesario resaltar que las

desconexiones frecuentes no son un factor importante en este caso, ya que toda la información es almacenada localmente en el dispositivo.

- Aplicaciones "clientes": existen cierto tipo de aplicaciones que para ser ejecutadas necesitan un gran conjunto de datos. Supongamos por ejemplo un escenario en el cual un usuario móvil se encuentra caminando por una zona desconocida de la ciudad y en un momento dado decide almorzar. Afortunadamente dicho usuario cuenta con un servicio "pre-pago" de su empresa de telefonía la cual le brinda un servicio a través del cual puede consultar desde cualquier área de cobertura la lista de los restaurantes que cumplen con algún criterio en particular (costo, cercanía, tipo de comida, etc.). Resulta claro que el usuario deberá acceder a dicha funcionalidad a través de una aplicación instalada en su teléfono celular. Sin embargo, lo que no resulta claro es dónde residen los datos sobre los que se ejecutará la consulta. De la discusión del inciso anterior se puede sospechar que no es viable el mantenimiento de la información en el dispositivo móvil, básicamente por tres motivos: primero, por el desperdicio de espacio de almacenamiento en el teléfono; en segundo lugar, por el tiempo que hubiera requerido enviar toda la información necesaria (que podría minimizarse dándole al usuario la posibilidad de conectar su dispositivo a un ordenador para descargar todos los datos necesarios); y tercero y más importante: ya no se estaría frente a una aplicación "móvil", ya que por más que el usuario se encuentre en lugares muy diferentes, éste solamente podrá consultar lo que se haya almacenado en su dispositivo. De lo precedente se podría concluir que para aplicaciones similares a la hipotética aplicación de restaurantes, es útil contar con algún recurso externo identificado al cual

requerirle de alguna manera preestablecida la ejecución de ciertos servicios. Evidentemente, al requerirse "ayuda" externa, el problema de las desconexiones frecuentes cobra aquí una real dimensión.

A pesar que la categorización anterior tiene como fin la separación en diferentes grupos a las aplicaciones del universo "móvil", aún existe un factor que las une: ambos tipos de aplicaciones requieren de información contextual para su ejecución. En el primer caso para establecer que esquema visual debe aplicarse (en este caso la información contextual está dada por la identidad del usuario), mientras que en el segundo caso para establecer los restaurantes más cercanos (en este caso la información contextual está representada por la ubicación del usuario en algún sistema de coordenadas).

Continuando con la caracterización de las aplicaciones "móviles"[7], otro aspecto que resalta de dichas aplicaciones es su alta optimización en la utilización de los recursos disponibles. Esto se hace evidente sobre todo cuando se cuentan con aplicaciones capaces de administrar elementos como cámaras o placas de red para evitar la descarga prematura de la batería.

Además de las caracterizaciones citadas anteriormente, es todavía posible introducir una nueva categorización que ayudará al lector a comprender más claramente el conjunto de aplicaciones a las cuales se apunta en esta tesis.

La figura 4 presenta un diagrama al respecto (en el centro se muestran las aplicaciones consideradas en este trabajo).

[7] Los términos "aplicación móvil" y "aplicaciones de dispositivos móviles" se utilizarán indistintamente, aún cuando el primero no es semánticamente correcto ya que la aplicación nunca es móvil (se ejecuta siempre donde ha sido instalada) sino el dispositivo donde se encuentra instalada.

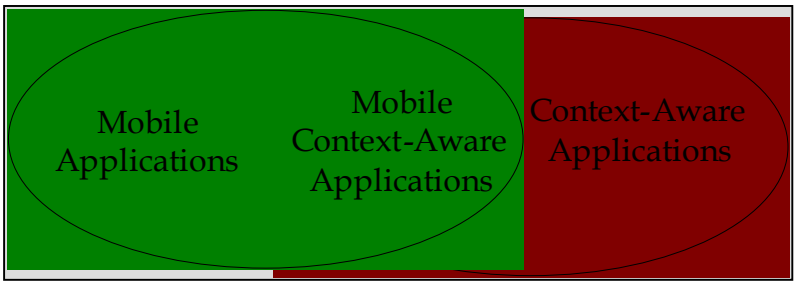

Figura 4: Categorización de aplicaciones móviles "context-aware".

Como ejemplo de cada uno de los tipos de aplicaciones expuestos en la figura 4, se pueden considerar:

- Aplicación móvil: un usuario lleva en su computadora portátil un software que le permite consultar su agenda de reuniones.
- Aplicación sensible al contexto: cualquier aplicación que obtenga información sobre el medio ambiente automáticamente; por ejemplo una aplicación que se reconfigure dependiendo del usuario que la está ejecutando (considerando el perfil del usuario).
- Aplicación móvil sensible al contexto: una aplicación de citas que automáticamente presenta una lista con las reuniones que tiene el usuario dentro del edificio en el que se encuentra actualmente (considerando además la presencia en el mismo edificio de las otras personas involucradas en las reuniones).

3.3 Requerimientos generales

Como ya se comentó en los capítulos anteriores, una de las mayores restricciones a las que están sujetas las aplicaciones para dispositivos móviles es la restricción en cuanto a tamaño. Dicha restricción afecta no sólo al tamaño de la aplicación misma (que trae como consecuencia una restricción en cuanto a la cantidad de funcionalidad que puede ser provista por la misma), sino que también afecta de manera directa la cantidad de información que puede ser manejada en forma local.

De la discusión precedente se deduce que toda aplicación middlet[8] que requiera de grandes cantidades de información para su funcionamiento deberá contar con un "par" (eventualmente podrían incorporarse más servidores) al cual solicitarle dicha información. Más adelante se verá que también dicho "par" o compañero puede ser utilizado para ejecutar parte de la computación necesaria para resolver un requerimiento. Esto plantea una vuelta al esquema "cliente-servidor" conocido. Sin embargo, este esquema es demasiado "rígido" como para acomodar los cambios que se dan habitualmente en este tipo de aplicaciones.

En este capítulo se presenta una arquitectura para aplicaciones móviles que requieran de algún tipo de "computación" externa para cumplir con su funcionalidad. Bajo este esquema, básicamente se contará con los siguientes actores:

- Aplicación middlet: esta aplicación representa al "cliente". Aquí conviene pensar en un middlet como un cliente de tipo semi-pesado (ya que posee características que le permiten manipular objetos, definir interfases gráficas, etc., pero al mismo tiempo tiene restricciones importantes de recursos disponibles).
- Servidor: esta aplicación será la encargada de proveer el acceso a diversas fuentes de información y de realizar aquellos cálculos o funciones que requieran del acceso a tales fuentes de información.
- Capa de comunicación: esta capa será responsable de comunicar las dos puntas de la arquitectura, es decir al cliente (middlet) con el servidor.

A continuación se detallan los criterios de calidad que serán utilizados para evaluar la arquitectura para aplicaciones móviles propuesta en este

[8] Middlet: tipo de aplicación definida en el estándar J2ME que se ejecuta en los dispositivos móviles. Todas las aplicaciones ejecutadas en un ambiente J2ME deben ser middlets. Para más detalles ver [J2ME].

capítulo. Algunos de ellos han sido tomados de los requerimientos generales planteados en la tesis doctoral de Dey [Dey 00].

- Robustez: esta cualidad está referida a la capacidad de la arquitectura de soportar condiciones de operación para las cuales no fue pensada originalmente. Por ejemplo se pueden considerar situaciones en las cuales es tan bajo el ancho de banda disponible que resulta sumamente difícil el intercambio de información entre el cliente y el servidor. Es tales circunstancias la arquitectura deberá acomodar fácilmente soluciones que permitan cambiar los niveles de detalle de la información que se intercambia, de modo de poder comunicarse incluso con un acceso a la red deficiente.
- Flexibilidad: este criterio establece que el agregado de nueva funcionalidad debe tener el menor impacto posible en la implementación ya existente. Por lo tanto, el agregar una nueva función sólo debe significar "diseñar" y "codificar" dicha función, siguiendo las guías propuestas por la arquitectura, sin requerir alteraciones las funciones ya existentes y que no están relacionadas directamente con la nueva funcionalidad.
- Bajo acoplamiento: tanto el cliente como el servidor deben tener el menor conocimiento posible de las particularidades de la implementación de su "contraparte". Al conseguir esta característica es posible cambiar todo el modelo existente en el servidor sin siquiera alterar en lo más mínimo la parte del cliente (obviamente manteniendo las interfases de comunicación entre ambas partes).
- Performance: esta característica es probablemente la más "visible" desde el punto de vista del usuario final y por lo tanto no debe ser descuidada. En todo momento debe buscarse la forma óptima de

integrar cada una de las capas a fin de conseguir un sistema óptimo en cuanto a la ejecución de la funcionalidad.

- Baja utilización del medio de comunicación: al reducir la cantidad de comunicación necesaria entre el cliente y el servidor, de manera indirecta se está reduciendo la posibilidad de caídas del sistema entero (situación común en los ambientes móviles debido a la baja calidad del servicio de conexión de estos dispositivos).
- Tamaño reducido: la arquitectura debe ser ligera en la utilización de los recursos existentes en el dispositivo, debido principalmente a la carencia de grandes recursos. Si la arquitectura fuera muy "pesada", no habría casi espacio para albergar la aplicación en sí misma.
- Mantenibilidad: la arquitectura debe definir claramente las responsabilidades de cada una de sus capas, de modo de hacer fácil y simple la detección y solución de fallas.
- Soporte para desconexiones: el cliente debería poder seguir trabajando, aún en aquellos casos en los cuales no se cuenta con una conexión establecida con un servidor. Resulta evidente que esta característica depende fuertemente del tipo de aplicación, pero incluso las que dependan en gran medida de un par remoto deberían reaccionar de manera predecible antes los períodos de desconexión.
- Basada en estándares del mercado: al utilizar estándares del mercado, las diferentes capas de la arquitectura no dependen de las particularidades de uno o varios productos particulares; pudiendo de esta manera cambiar a diferentes implementaciones que cumplan con un estándar determinado frente a fallas o problemas de performance.

3.4 Capas de la arquitectura

La arquitectura propuesta tiene su base fundamental en el patrón de diseño *"Command"*[9] [Gamma et al. 94]. Dicho patrón establece la idea de modelar los diferentes métodos de una clase como objetos de primera clase en sí mismos. A partir de dicha idea, puede implementarse cada función del sistema como un "command"; es decir que el cliente al solicitar la ejecución de alguna función en el servidor, en última instancia estará pidiendo la ejecución de un comando[10].

La siguiente figura presenta un diagrama de paquetes que establece de manera clara las relaciones entre los principales componentes de la arquitectura.

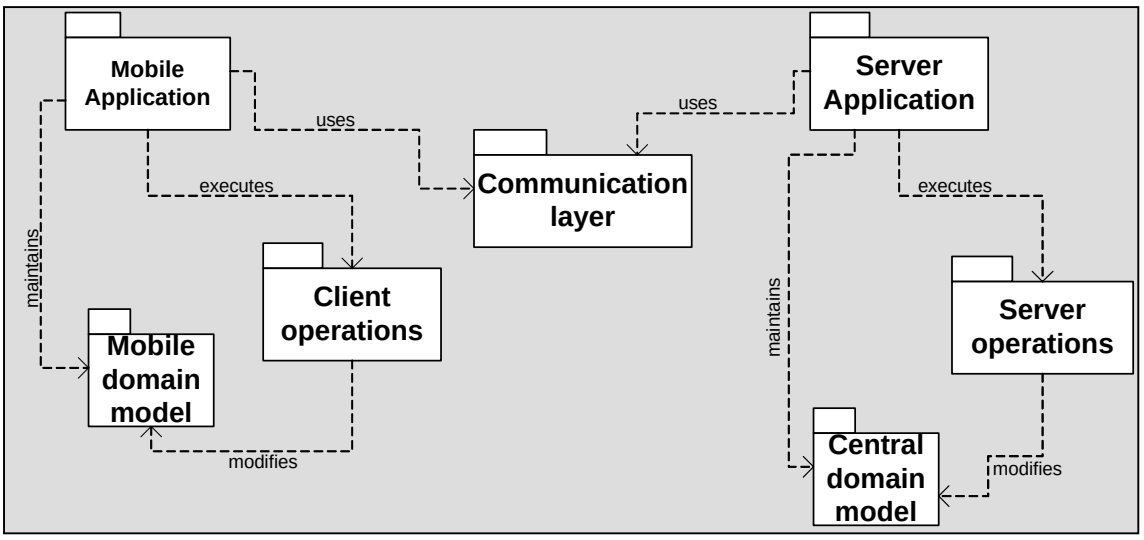

Figura 5: Diagrama de paquetes de la arquitectura

[9] El patrón de diseño Command establece que las diferentes operaciones que realiza un objeto (modeladas como métodos generalmente) pueden ser re-implementadas como instancias de clases polimórficas, cada una de las cuales realiza una función en particular.

[10] Los términos "command" y "comando" serán usados en forma indistinta y ambos se refieren a una instancia de alguna clase que implemente el patrón de diseño Command.

A continuación se muestra un diagrama informal de los principales componentes de la arquitectura y sus interacciones. Luego se brinda una explicación detallada de cada uno de estos componentes.

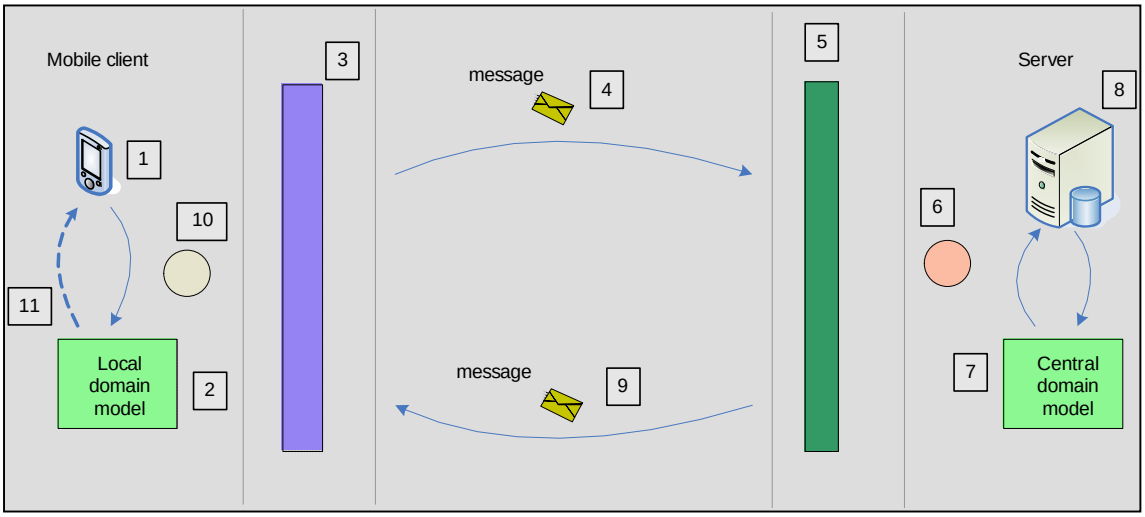

Figura 6: Principales componentes de la arquitectura.

1. Dispositivo móvil (PDA o Teléfono Celular): es el dispositivo sobre el cual se está ejecutando una aplicación middlet.
2. La aplicación móvil está soportada por un modelo orientado a objetos, el cual representa los principales conceptos del dominio real. Debido a las restricciones de tamaño antes citadas, no es posible contener en forma local un modelo grande, razón por la cual en general la mayoría de las clases de este modelo serán implementaciones del patrón de diseño *"Proxy"*[11] [Gamma et al. 94], lo cual trae como consecuencia directa que la aplicación sea independiente de los detalles concernientes a la distribución. De hecho, la aplicación no "conoce" en ningún momento si los requerimientos se están resolviendo en forma local o remota. Es necesario considerar la característica deseada de bajo acceso a la

[11] El patrón de diseño Proxy establece la creación de objetos que representen en forma local a otros objetos que son mantenidos en otros espacios de memoria. Esto permite que los clientes de dichos objetos sean independientes de los detalles de conexión y otros aspectos relacionados con la distribución.

red, con lo cual se debe establecer un esquema bajo el cual la información que ya ha sido recibida desde el servidor se mantiene en forma local (teniendo en cuenta las restricciones de espacio) a fin de minimizar las llamadas remotas. Una implementación de este esquema establecería un período de caducidad de la información mantenida en forma local; una vez caducado dicho período, la información volvería a ser representada por un proxy capaz de obtenerla nuevamente a través de una llamada remota. Este esquema se torna necesario en ambientes en los cuales hay más de un cliente conectado, alterando la información del servidor (tener en cuenta que en general el servidor no podrá comunicar cada cambio de la información a todos los clientes) de modo que cuando un dispositivo cualquiera quiera enviar información puede darse el caso de dicha información no sea válida. El esquema de caducidad de información tiende a minimizar dichos casos de información inválida.

3. Los proxies del modelo utilizan una capa de comunicación para enviar y recibir la información. Dicha capa se encarga de encapsular todo el conocimiento necesario para establecer una comunicación fiable y segura con el servidor. Incluso, al aplicar el patrón de diseño *"Strategy"*[12] [Gamma et al. 94], puede generarse una compleja capa de comunicaciones con diferentes implementaciones para el envío y recepción de información desde el servidor. Al integrar esta capacidad con el desarrollo de prototipos, resulta claro que se podría desarrollar íntegramente la capa del cliente aún antes de comenzar el desarrollo del servidor (ya que el cliente se comunica siempre a través de los proxies, se pueden desarrollar en primer lugar una serie de proxies que

[12] El patrón de diseño Strategy establece la creación de clases que representan una misma operación conceptual, pero variando la forma en la cual se implementa tal operación. Esto permite variar la implementación sin que varíen los resultados obtenidos.

devuelven datos ficticios, a fin de poder desarrollar en forma totalmente independiente al cliente del servidor, y más tarde integrar los proxies que realmente se conectan a través de la capa de comunicación con el servidor).

Esta capa de comunicación se basa en el patrón *"Strategy"* nuevamente para implementar la manera en la cual se transforman los objetos para viajar a través del medio físico (por ejemplo en formato CSV[13] o XML [XML 03]).

4. La capa de comunicación se comunica con su contraparte del servidor mediante el envío de mensajes, los cuales contienen una indicación de la tarea que se está solicitando, además de toda la información necesaria para ejecutar tal tarea. Es aquí en donde se logra la característica deseada de bajo acoplamiento, debido a varias razones:
    a. El cliente solamente envía un nombre simbólico que representa la operación que se solicita (en realidad el nombre simbólico está relacionado con un comando que será ejecutado en el servidor).
    b. El cliente envía la información necesaria, pero no conoce en detalle la forma en la que se utiliza dicha información.
    c. El cliente no conoce el formato con el cual se debe enviar la información, razón por la cual se utiliza un protocolo que no se basa en estructura, sino en contenido como lo es XML.
5. El servidor cuenta también con una capa de comunicación que utiliza para interpretar los mensajes recibidos desde el cliente conteniendo información útil para ejecutar alguna operación.

[13] CSV (Comma Separated Values): formato de texto en el cual los valores se envían separados por comas. La estructura del archivo debe ser acordada por las partes que estén participando del intercambio de información.

6. Las operaciones o servicios provistos por el servidor están implementados como comandos, razón por la cual el mensaje que debe ser enviado por el cliente debe contener alguna indicación (simbólica) del comando que se desea ejecutar. El cliente debe utilizar un alias simbólico del comando del servidor para mantener lo más bajo posible el acoplamiento entre el cliente y el servidor.
7. El servidor cuenta con un modelo completo del dominio del problema ya que no tiene restricciones de tamaño como el cliente. A raíz de la ejecución de los comandos, este modelo irá alterándose y también creciendo a medida que se integran diversas fuentes externas de información. El modelo presente en el cliente (basado en proxies) es un subconjunto del modelo presente en el servidor.
8. El servidor de aplicaciones presta servicios a clientes mediante la publicación de nombres simbólicos que pueden ser ejecutados en forma remota por los clientes. Es probable que el servidor mantenga un modelo de objetos en algún repositorio de persistencia utilizando por ejemplo Hibernate [Hibernate 02], razón por la cual este servidor es responsable de manejar todo el acceso transaccional a dicho modelo.
9. Una vez que se han ejecutado satisfactoriamente los comandos, el servidor notifica los resultados al cliente en forma de mensajes nuevamente. Dichos mensajes contienen una indicación simbólica (nombre del comando en el cliente) de la tarea que debe ejecutar el cliente y toda la información necesaria para ejecutar la tarea.
10. El cliente tiene desarrollado un conjunto de comandos, los cuales contienen la lógica de la aplicación. Estos comandos son ejecutados como consecuencia de la recepción de mensajes desde el servidor.

11. Mediante la implementación de un esquema de notificaciones genérico, el modelo local en el cliente notifica a las diferentes vistas para que se actualicen. Dicho esquema genérico es el propuesto por el patrón de diseño "*Observer*"[14] [Gamma et al. 94].

3.5 Diagrama de clases

A continuación se presenta un diagrama de clases que soporta el esquema conceptual planteado en la figura anterior.

Por una cuestión de claridad y legibilidad, se presentarán varios diagramas de clases, cada uno con su correspondiente explicación.

3.5.1 Diseño de clases de los mensajes[15]

Este diseño contiene la definición común que representa los mensajes que son intercambiados entre el cliente y el servidor, acorde con los puntos 4 y 9 de la descripción informal de la arquitectura (figura 6). A continuación se describen brevemente los principales elementos de este paquete.

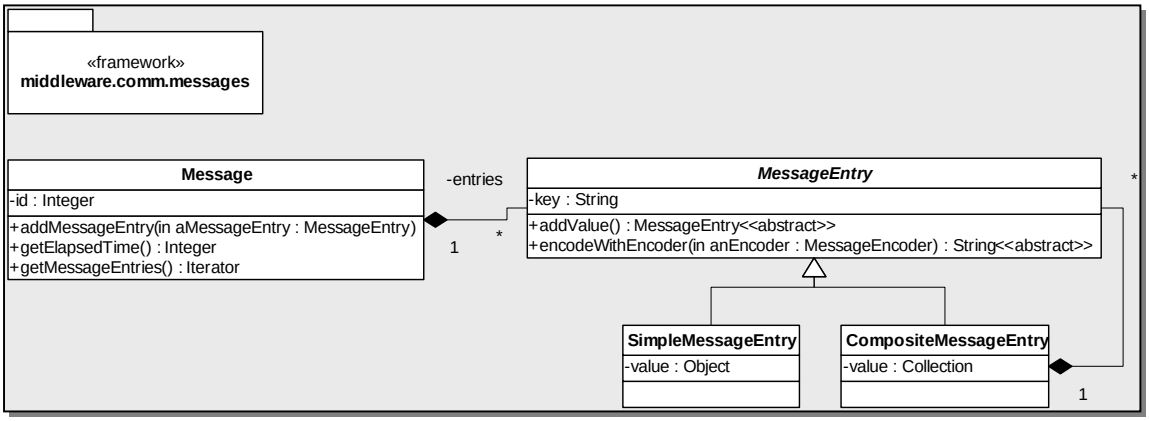

Figura 7: Diagrama de clases de los mensajes.

[14] El patrón de diseño Observer plantea un esquema de notificaciones genérico para que la parte que notifica un cambio no deba conocer los detalles de la parte a ser notificada. Esto permite la definición de múltiples vistas sin tener que alterar de manera alguna a la parte que notifica sus cambios.
[15] Al utilizar el lenguaje de programación Java, estas clases serán agrupadas lógicamente en paquetes.

- Clase *Message*: representa un mensaje que es intercambiado entre las diferentes capas de la arquitectura. Contiene entradas con un nombre y valores.

- Clase *MessageEntry*: es el tope de la jerarquía de las clases que representan una entrada en un mensaje. Define el protocolo que deberá ser implementado por sus subclases. Define el mensaje *"encodeWithEncoder()"* que es enviado por un codificador de mensajes que forma parte de una implementación de la técnica de "double-dispatching"[16]. El siguiente código ilustra esta técnica (código en negritas).

```
public abstract class Encoder {

  public String encodeMessageEntry(MessageEntry
anEntry) {

  anEntry.encodeWithEncoder(this);

  }

}
```

```
public class SimpleMessageEntry {

  public String
encodeWithEncode(MessageEncoder anEncoder)
  {


anEncoder.encodeSimpleMessageEntry(this);

  }
```

Además esta clase representa la base para la implementación del patrón de diseño *"Composite"*[17] [Gamma et al. 94].

- Clase *SimpleMessageEntry*: esta clase representa la "hoja" simple del patrón del diseño *"Composite"*. En otras palabras, las instancias de esta clase se utilizan para almacenar datos atómicos dentro de un mensaje.

[16] Double-dispatching es una técnica de diseño que establece la forma de interacción entre dos objetos, de manera que ninguna deba asumir nada en particular de la otra instancia. Para más detalles ver [Double Dispatching].

[17] El patrón de diseño Composite brinda una solución para modelar situaciones en donde ciertos objetos pueden estar compuestos de objetos que pertenecen a su misma jerarquía, de forma recursiva.

- Clase *CompositeMessageEntry*: esta clase representa la agregación de elementos de la jerarquía de entradas de mensajes (corresponde a la agregación del mismo patrón anterior). Las instancias de esta clase se utilizan cuando es necesario almacenar dentro de un mensaje un conjunto de datos lógicamente asociados. Los elementos que componen esta entrada compuesta son instancias de la clase *SimpleMessageEntry*.

3.5.2 Diseño de clases de los codificadores de mensajes

Aquí se presenta la definición de las clases que son utilizadas para codificar los mensajes que son intercambiados entre el cliente y el servidor (corresponde a los puntos 3 y 5 de la figura 6). La figura 8 muestra el diagrama de clases correspondiente, y luego se discuten las características de las principales clases de cada elemento.

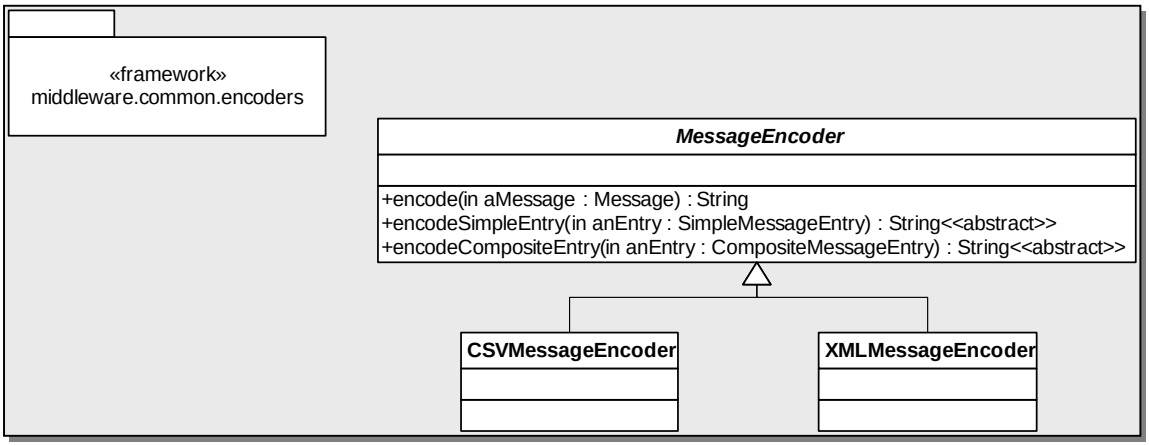

Figura 8: Diagrama de clases de los codificadores de mensajes.

- Clase *MessageEncoder*: esta clase representa el tope de la jerarquía de las clases que son utilizadas para codificar los mensajes que se intercambian entre el cliente y el servidor. Esta clase es una

implementación del patrón de diseño *"Strategy"* [Gamma et al. 94], ya que establece en conjunto con sus subclases diferentes alternativas de codificación de mensajes que pueden ser intercambiadas en forma dinámica.

Es importante recalcar que esta clase también define un método cuya firma es "*public String encode(Message aMessage)*", el cual sigue los lineamientos planteados por el patrón de diseño *"Template Method"*[18] [Gamma et al. 94] para establecer un algoritmo genérico de codificación de mensajes que luego es implementado en forma concreta por cada una de las subclases.

Los métodos cuya firmas son "*public String encodeSimpleEntry(SimpleMessageEntry anEntry)*" y "*public String encodeCompositeEntry(CompositeMessageEntry anEntry)*" son implementaciones de la técnica de double dispatching y son utilizados en conjunto con las clases de que modelan los mensajes en sí mismos.

- Clase *CSVMessageEncoder*: esta clase representa una implementación concreta de codificación. Las instancias de esta clase al recibir un mensaje para su codificación retornan un string que contiene la información codificada en formato CSV.

- Clase *XMLMessageEncoder*: esta clase representa una implementación concreta de codificación. Las instancias de esta clase al recibir un mensaje para su codificación retornan un string que contiene la información codificada en formato XML.

[18] El patrón de diseño Template Method establece la definición de un algoritmo genérico en la superclase, de manera de asegurar que cada una de las subclases siga un mismo esquema de trabajo. Las particularidades de la implementación son libradas a las subclases concretas.

3.5.3 Diseño de clases de los decodificadores de mensajes

Esta sección contiene definición de las clases que son utilizadas para decodificar los mensajes que son intercambiados entre el cliente y el servidor (acorde con la descripción informal de la arquitectura presentada en la figura 6, etiquetas 3 y 5 respectivamente).

A continuación se presenta el diagrama de clases correspondiente (figura 9) para luego comentar las particularidades de cada clase importante.

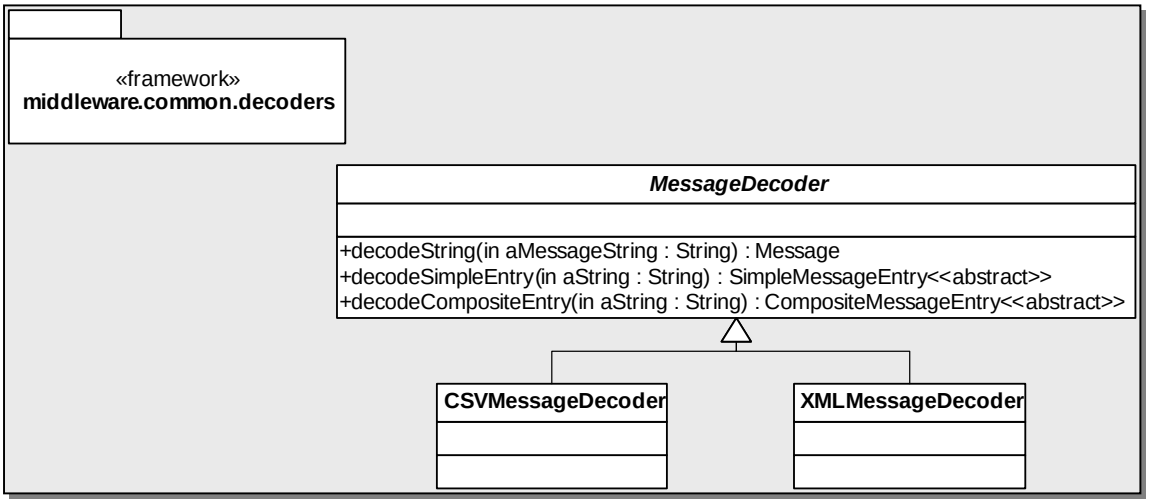

Figura 9: Diagrama de clases de los decodificadores de mensajes.

- Clase *MessageDecoder*: Esta clase representa el tope de la jerarquía de las clases que son utilizadas para decodificar los mensajes que se intercambian entre el cliente y el servidor. Esta clase es una implementación del patrón de diseño "*Strategy*" [Gamma et al. 94], ya que establece en conjunto con sus subclases diferentes alternativas de decodificación de mensajes que pueden ser intercambiadas en forma dinámica.

Es preciso resaltar que esta clase también define un método cuya firma es "*public Message decodeString(String aMessageString)*", el cual sigue los lineamientos planteados por el patrón de diseño *"Template Method"* [Gamma et al. 94] para establecer un algoritmo genérico de decodificación de mensajes que luego es implementado en forma concreta por cada una de las subclases.
Los métodos cuya firmas son "*public String decodeSimpleEntry(String aString)*" y "*public String decodeCompositeEntry(String aString)*" son implementaciones de la técnica de double dispatching y son utilizados en conjunto con las clases del paquete middleware.comm.messages.

- Clase *CSVMessageDecoder*: esta clase es una implementación concreta que permite decodificar un string que contiene información en formato CSV para retornar una instancia de la clase *Message* apropiadamente cargada.

- Clase XMLMessageDecoder: esta clase representa otra implementación concreta de decodificación que permite interpretar la información contenida en un string con formato XML para retornar una instancia de la clase *Message* configurada apropiadamente.

3.5.4 Diseño de la capa de comunicaciones del cliente

Esta sección presenta las definiciones de las clases que son utilizadas en la capa del cliente para entablar una comunicación con el servidor (corresponde a la etiqueta 3 de la descripción informal de la figura 6).

La siguiente figura presenta el diagrama de clases correspondiente y a continuación de la misma se brindan detalles de las clases más importantes.

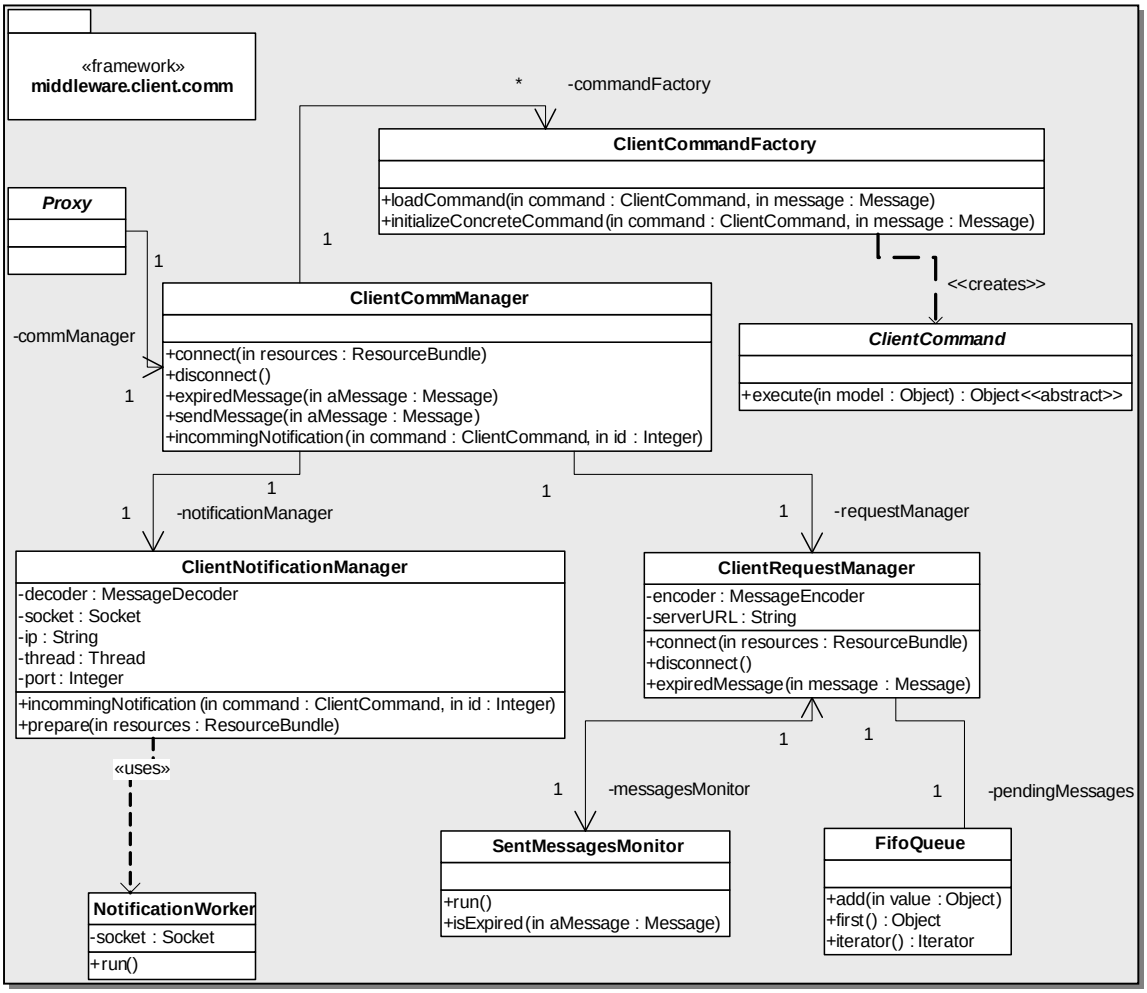

Figura 10: Diagrama de clases de la capa de comunicación en el cliente.

- Clase *Proxy*: en la sección anterior "Capas de la Arquitectura" se citó que la implementación del modelo en el cliente podría estar guiada por la aplicación del patrón de diseño "*Proxy*" [Gamma et al. 94]. Teniendo esto en cuenta, este paquete cuenta con una clase abstracta *Proxy*, la cual será la base que deberán extender cada una de las clases que representen conceptos del dominio en el cliente. En principio esta clase colabora con un administrador de comunicaciones para llevar a cabo su labor de comunicarse con el

servidor, a la vez que se simula la resolución de un requerimiento en forma local.

- Clase *ClientCommandFactory*: esta clase es una implementación del patrón de diseño *"Abstract Factory"*[19] [Gamma et al. 94] y permite la creación de los comandos que son ejecutados sobre el modelo local del cliente. La principal ventaja de su utilización es que la aplicación se ve totalmente independizada de los detalles de creación de cada comando. Incluso el proceso que debe seguirse a partir de la recepción de un mensaje (instancia de la clase *Message*) hasta la creación y carga de un comando apropiado es encapsulado por esta clase.

- Clase *ClientCommand*: esta clase abstracta representa el tope de la jerarquía de comandos que más tarde deberán ser implementados por el desarrollador. Establece el protocolo básico que todo comando debe implementar. Un comentario aparte merece el método cuya firma es *"public Object execute(Object model)"*; en esta firma, "model" representa a un objeto del modelo del cliente a partir del cual se puede alcanzar a todas las demás instancias (patrón de diseño *"Root Object"*[20] [Loomis 95]).

- Clase *ClientCommManager*: las instancias de esta clase representan el punto de acceso a la capa de comunicación y se utilizan para enviar los mensajes al servidor. Con el fin de asegurarse que en todo momento exista una única instancia de esta clase en todo el

[19] El patrón de diseño Abstract Factory ofrece una solución a los problemas que se encuentran cuando se deben crear familias de objetos relacionados (en este caso comandos).
[20] El patrón de diseño Root Object establece la creación de una clase que representa al sistema en sí que se está modelando (ej. la clase Banco para una aplicación bancaria). Dicha clase provee acceso fácil a todos los demás elementos del modelo.

espacio de memoria de la aplicación se ha implementado el patrón de diseño *"Singleton"* [Gamma et al. 94].

La única instancia de esta clase es utilizada tanto para enviar los mensajes (método *"sendMessage(Message aMessage)"*) como para recibir las notificaciones que el servidor pudiera estar enviando (método *"incommingNotification(Command aCommand, int id)"*). A los efectos de las aplicaciones que utilicen este administrador de comunicaciones, se puede considerar esta clase como un *"Facade"*[21] [Gamma et al. 94], ya que las aplicaciones se ven abstraídas de los detalles complejos de la comunicación. Esto además permite en el futuro cambiar el protocolo de comunicación subyacente, sin que este hecho tenga un impacto importante en el resto del sistema.

- Clase *NotificationWorker*: esta clase representa un hilo de ejecución separado. Las instancias de esta clase son utilizadas para recibir en forma asincrónica el contenido de un mensaje enviado por el servidor. Una vez que terminan de recibir el contenido del mensaje detienen su ejecución y son eliminadas del sistema.

- Clase *SentMessagesMonitor*: el subsistema de comunicaciones utiliza una instancia de esta clase con el fin de permitir un control de los mensajes que son enviados al servidor y de los cuales se obtienen respuestas. Cada vez que es enviado un mensaje al servidor, se le asigna un "timestamp"[22] y se lo almacena en una cola. Cada mensaje enviado al servidor genera una notificación de éste (que puede contener información o representar simplemente

[21] El patrón de diseño Facade establece la creación de una o varias clases que encapsulan la complejidad de un subsistema, de modo que los objetos clientes se vean aislados de los cambios internos, además de poder operar con una interface simplificada.

[22] Timestamp es un atributo asignado automáticamente por el sistema que refleja la fecha y hora en que se produjo un evento.

un ACK[23]). Al recibirse una confirmación de recepción de un mensaje, la instancia de esta clase busca el mensaje correspondiente en la cola de mensajes enviados y lo elimina. En caso de que expire una cantidad de tiempo determinada, se informa al administrador de comunicaciones de este hecho con el fin de alertar a la aplicación de posibles fallas en la comunicación.
La implementación de esta clase sigue los principios establecidos por el patrón de diseño *"Singleton"* [Gamma et al. 94].

- Clase *ClientRequestManager*: esta clase es responsable de administrar todos los detalles relativos a los mensajes enviados hacia el servidor. El administrador de comunicaciones delega toda la responsabilidad del manejo de los mensajes enviados, como así también el manejo de los mensajes expirados a esta clase. La implementación de esta clase también sigue los lineamientos planteados por el patrón de diseño *"Singleton"* [Gamma et al. 94] a fin de garantizar la existencia de una única instancia en todo el sistema. Cuando se envía un mensaje al servidor, este administrador almacena una copia local en una cola, a fin de monitorear los mensajes que no han sido respondidos en un tiempo prudencial.

3.5.5 Diseño de la capa de comunicación del servidor

Aquí se presenta la definición de una implementación minimalista de una capa de comunicación para el servidor de aplicaciones (que está representada por la etiqueta 5 de la descripción informal de la arquitectura presentada en la figura 6). Dicha capa podría ser utilizada en conjunto con

[23] ACK es un término técnico que corresponde a una notificación de recepción de un mensaje (ACK es un acrónimo de "Acknoledgement").

Servlets[24] para armar un servidor que brinde servicios a los dispositivos móviles sobre un protocolo estándar como HTTP [HTTP 96].

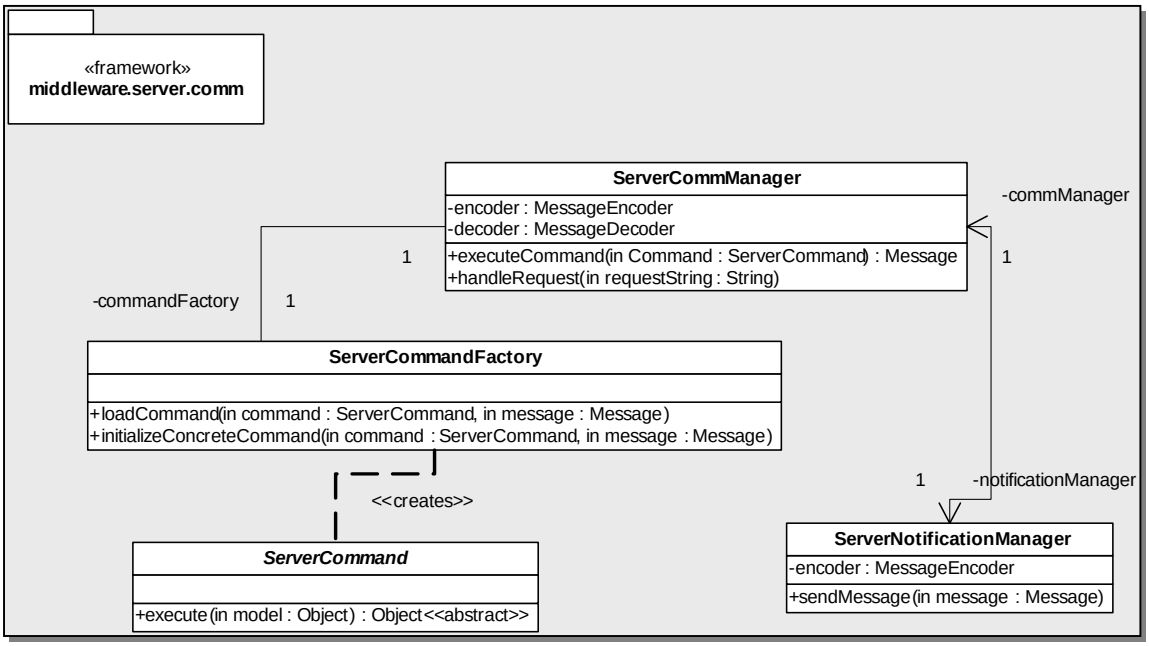

Figura 11: Diagrama de clases de la capa de comunicación del servidor.

- Clase *ServerCommManager*: esta clase administra la comunicación en el servidor de aplicaciones. Al utilizarse en conjunto con los servlets [Servlets 05] es posible generar una arquitectura en la cual éstos últimos reciben los pedidos sobre HTTP [HTTP 96] y luego direccionan el pedido hacia la única instancia de esta clase (patrón de diseño *"Singleton"* [Gamma et al. 94]) para que sea ejecutado finalmente un comando del lado del servidor. A los fines prácticos, para las aplicaciones mantenidas en el servidor, esta instancia actúa como una "fachada" detrás de la cual se pueden cambiar los mecanismos de comunicación sin impactar de alguna manera en las clases de las aplicaciones (patrón de diseño "Facade" [Gamma et al. 94]).

[24] Servlets: tecnología perteneciente al estándar J2EE para la definición de aplicaciones que corren en un servidor dentro de un contenedor que le brinda servicios básicos como gerenciamiento del ciclo de vida. Para más detalles ver [Servlets 05].

- Clase *ServerCommandFactory*: esta clase tiene la responsabilidad de encapsular todos los detalles concernientes a la creación de los comandos que se ejecutan sobre el modelo mantenido en el servidor. Representa una implementación del patrón de diseño *"AbstractFactory"* [Gamma et al. 94]) ya que permite la creación de familias de objetos relacionados, en este caso comandos, los cuales pueden ser implementados de diferentes maneras. Sin importar la forma de implementación, las clases que los utilizan no se ven impactadas por los cambios en los comandos, ya que siempre utilizan este "factory" para crearlos, sin necesidad de utilizar siquiera el nombre de la clase del comando involucrado.

- Clase *ServerNotificationManager*: este objeto es responsable de administrar las notificaciones que son enviadas (en formato codificado) a las aplicaciones móviles clientes. A diferencia de lo que sucede en el cliente, este objeto no mantiene un registro de los mensajes enviados (ya que por defecto un servidor no es capaz de iniciar una conversación con un cliente).

- Clase *ServerCommand*: esta clase abstracta representa la base que deberá ser extendida por cada uno de los comandos que se ejecutarán sobre el modelo mantenido en el servidor. De esta manera puede ser considerada como una implementación del patrón de diseño *"Command"* [Gamma et al. 94]. Es necesario también que esta clase se basa en el patrón de diseño *"Root Object"* [Loomis 95], ya que define un método *"public Object execute (Object model)"* que establece como parámetro de entrada el objeto raíz del modelo local del servidor a partir del cual se puede acceder al resto de las instancias.

3.6 Conclusiones

En este capítulo se ha presentado una arquitectura orientada a objetos que puede ser utilizada para el desarrollo de aplicaciones móviles que requieran de la presencia de un servidor como soporte para su funcionalidad.

Dicha arquitectura se basa en técnicas y patrones de diseño conocidos y aceptados, como *"Facade"*, *"Singleton"*, *"AbstractFactory"*, *"Command"*, etc. Cada uno de estos patrones ha sido aplicado con la idea de ganar en aspectos de calidad, mantenibilidad y flexibilidad, sin descuidar aspectos importantes como performance o robustez. Es en particular el último patrón citado más arriba el que brinda la base conceptual sobre la que se construye esta arquitectura, ya que la misma representa ni más ni menos que un *"Command"* distribuido.

A continuación se discute brevemente como la arquitectura planteada en este capítulo cumple los requerimientos planteados en la sección 3.3:

- Robustez: la posibilidad de desarrollar cada capa en forma separada de las demás, trae en forma indirecta la posibilidad de ejecutar pruebas exhaustivas sobre cada componente del sistema por separado, lo que incrementa la robustez general de la arquitectura.
- Flexibilidad: la incorporación de nueva funcionalidad, estrategia de codificación/decodificación, protocolo de envío de los mensajes y otros elementos de la arquitectura es sumamente fácil ya que la misma tiene las responsabilidades bien delimitadas, haciendo que el sistema se vuelva suficientemente flexible para acomodar sin mayores inconvenientes o cambios las evoluciones que se requieran.
- Bajo acoplamiento: la utilización de una capa de comunicación que utiliza mensaje codificados en formato XML tiene como consecuencia que el cliente y el servidor se vean totalmente

independizados de representaciones propietarias, permitiendo a cada una de las partes variar mientras se respete la interfase de intercambio de información establecida.

- Performance: la aplicación de los patrones de diseño *"Command"* y *"Proxy"* trae como beneficio la posibilidad de ejecutar las operaciones en forma asincrónica, lo cual conlleva una mejora sustancial en la performance de la aplicación cliente que no debe esperar las respuestas del servidor a cada pedido para poder realizar el siguiente pedido.
- Baja utilización del medio de comunicación: la utilización de proxies "inteligentes" que almacenan localmente los datos accedidos recientemente reduce drásticamente la necesidad de entablar comunicación con el servidor.
- Tamaño reducido: ninguna de las soluciones utilizadas en esta arquitectura impone una gran utilización de recursos.
- Mantenibilidad: la separación de las responsabilidades lograda en la arquitectura planteada incremente en gran medida la mantenibilidad del sistema, ya que por ejemplo al detectarse problemas con una funcionalidad determinada (o comando), sólo se debe alterar dicha funcionalidad dejando intacto el resto del sistema.
- Soporte para desconexiones: nuevamente la utilización de proxies "inteligentes" hace que sea posible almacenar localmente (en el dispositivo móvil) los cambios que se deben realizar en el modelo central del servidor hasta que sea posible enviarlos efectivamente al mismo para su ejecución.
- Basada en estándares del mercado: en la definición de esta arquitectura se han utilizado tecnologías y estándares del mercado reconocidos, como Servlets, XML, J2ME; lo cual confiere un soporte real del mercado para la implementación del prototipo.

La arquitectura de software (con sus múltiples capas) planteada en este capítulo será tomada a continuación como base para la especificación de un framework orientado a objetos que especifique y proporcione guías claras para el diseño de aplicaciones móviles y que a su vez son "sensibles al contexto".

Capítulo IV: Framework O.O. para soportar Context-Awareness

A más de tres décadas de su presentación, la ley de Moore[25] continúa verificándose, sobre todo en el ámbito de los equipos electrónicos relacionados con la "movilidad" de los usuarios. Sin embargo, el sólo hecho de contar con más poder de cálculo y recursos no es suficiente para asegurar el correcto diseño y la posterior implementación de aplicaciones móviles.

4.1 Introducción

En el capítulo anterior se ha presentado una arquitectura de software especialmente pensada para cubrir los requerimientos de las aplicaciones móviles. Un conjunto importante de este tipo de aplicaciones son las que son capaces de responder a estímulos del medio ambiente. Resulta entonces interesante plantear un framework orientado a objetos que presente guías de diseño y soluciones generales a los tipos de problemas usualmente hallados al construir aplicaciones context-aware. Dicho framework se basará en las características principales de la arquitectura del capítulo anterior.

4.2 Especificación de Contextos

Existen diferentes definiciones de lo que un "contexto" representa en el ámbito de la computación móvil. Entre ellas tenemos:

[25] Ley de Moore: ley presentada en 1965 por Gordon Moore que establece que la potencia de cálculo de los dispositivos se duplica cada 18 meses. Actualmente esta velocidad de duplicación de la capacidad ha sido sobrepasada. Para más información referirse a [Moore 00].

- "*Ubicación del usuario, identidades de objetos y personas cercanas, así como los cambios a dichos objetos*" [Schilit B. et al. 94].
- "*Ubicación del usuario, identidades de las personas alrededor del usuario, la hora del día, estación del año, temperatura, etc.*" [Brown et al. 97].
- "*Ubicación del usuario, medio ambiente, identidad y tiempo*" [Ryan et al. 97].
- "*Estado emocional del usuario, foco de atención, ubicación y orientación, fecha y hora, objetos y gente en el medio ambiente del usuario*" [Dey 98].

Resulta evidente a partir de las definiciones citadas más arriba, que cualquier definición nueva que se intente dar, debería contemplar mínimamente los aspectos relacionados con la ubicación del usuario, su identidad y perfil asociado, medio ambiente (el cual incluye objetos, personas y recursos circundantes), y su situación temporal.

Aquí resulta interesante exponer una breve definición propia a modo de introducir al lector a la jerarquía de clases tendientes a modelar los contextos y sus características. Un contexto es *toda aquella información que resulta útil (de manera directa o indirecta) para personalizar y adaptar la operatoria de un software para un usuario en particular*. Dentro de dicha información podemos incluir información acerca del medio ambiente físico y social del usuario (lugar, temperatura, usuarios en las cercanías, humedad, etc.), intención del usuario y su foco de atención, estado emocional del usuario, medio ambiente lógico (permisos para ejecutar alguna actividad), recursos disponibles (conectividad, poder de procesamiento, ancho de banda, recursos para impresión, etc.).

Considerando la definición anterior, se plantea la siguiente jerarquía inicial para modelar contextos en la figura 12.

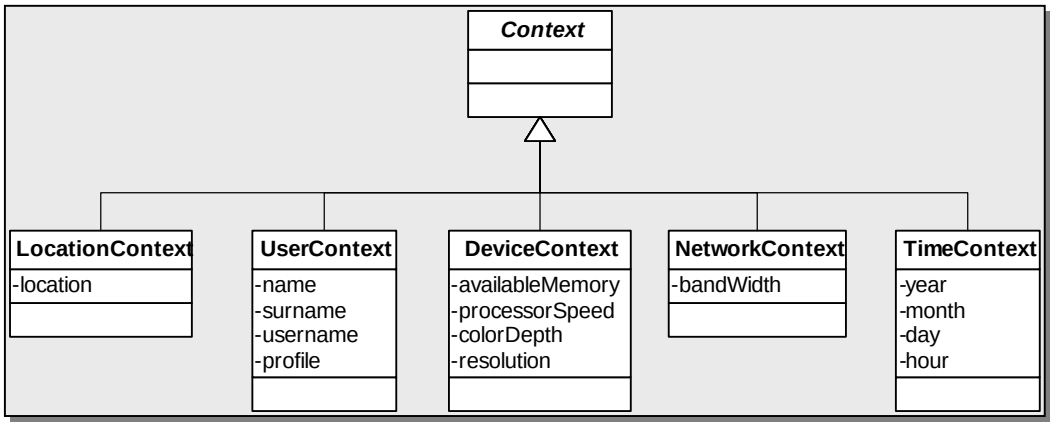

Figura 12: Jerarquía de clases de contextos.

A continuación se brinda una reseña de las características y responsabilidades generales de cada una de las clases participantes:

- Clase *Context*: es la raíz de la jerarquía de contextos. Establece el protocolo que deben implementar todos los diferentes "tipos" de contextos y brinda comportamiento en común a todas las subclases. Es el punto de partida para el establecimiento de una jerarquía polimórfica que no imponga conocimiento en particular de cada subclase concreta a los objetos clientes.

- Clase *LocationContext*: es la subclase responsable de modelar el comportamiento relacionado con la ubicación del usuario móvil. Contiene información acerca de la ubicación del usuario. Esta posición será derivada a partir de la colaboración con algún tipo de dispositivo físico de ubicación como celulares o GPS.

- Clase *UserContext*: esta subclase representa toda la información personal relevante del usuario móvil. Cada vez que se requieran datos acerca del usuario, como su nombre, edad, sexo; serán las instancias de esta clase las que proveerán acceso a dicha información. Hay que destacar que los atributos mostrados en la figura son solamente la "punta del iceberg", ya que incluso se

podrían introducir conceptos tales como privacidad, estado de ánimo o incluso aspectos de personalización de preferencias (ver [UWA 02]).

- Clase *TimeContext*: las instancias de esta subclase serán las encargadas de mantener la información relacionada con el tiempo. Aquí pueden contemplarse múltiples significados de la palabra "tiempo":
  - Momento en el que se ejecuta la consulta por parte de la aplicación cliente.
  - Restricción de tiempo de respuesta en la ejecución del servicio solicitado.
  - Período durante el cual se debe ejecutar el servicio solicitado.
  - Momento para el cual debería estar lista la respuesta.
  - Momento del día a partir del cual se definen los servicios disponibles para el cliente.
  - Momento del día que se considera para armar un posible itinerario de recorrido por distintos lugares (para el caso de una aplicación turística).

- Clase *NetworkContext*: esta clase es responsable de manejar todos los aspectos de la información en relación al estado de la red, a través de la cual el usuario envía y recibe datos. Entre los eventos más comunes que suelen encontrarse se pueden citar: variaciones en el ancho de banda; desconexión y posterior conexión; cambio de proveedor de servicio de red.

- Clase *DeviceContext*: aquí se almacenarán todos los datos concernientes al dispositivo que el usuario móvil esté utilizando al

momento de ejecutar las aplicaciones. Actualmente se contemplan datos como cantidad de memoria disponible, resolución y profundidad de colores de la pantalla. En el futuro serán incorporadas capacidades multimedia, velocidad del procesador, etc.

Es evidente que el diseño presentado en la figura 12 es solamente la base de la solución requerida. Si nos remitimos a los contextos tal cual han sido expresados en la figura, en el estado actual son meros repositorios de información. Esto se debe a la falta de pautas de diseño y de utilización que brinden soluciones y guías de trabajo a los desarrolladores de aplicaciones basadas en estos contextos. Resulta entonces más clara la necesidad de contar con una nueva capa que administre la información contextual representada en la figura 12.

La jerarquía de contextos presentada más arriba merece un comentario extra relacionado con el aparente "solapamiento" de modelos que existe cuando un diseñador modela una entidad del mundo real dentro de su aplicación (por ejemplo una persona) que también existe como concepto en la jerarquía de contextos presentada recientemente (en particular la clase *UserContext*). En este punto es necesario establecer que bajo ningún punto de vista se intenta suprimir la necesidad por parte del diseñador de la aplicación sensible al contexto de modelar los elementos del dominio. Como se verá más adelante, la jerarquía planteada no es más que un componente interno que será utilizado por el framework de contextos para notificar de cambios en la información contextual. Será responsabilidad exclusiva de las aplicaciones definir la forma en la que se mapean las instancias de la jerarquía de contextos con los objetos propios del dominio de la aplicación. Como ejemplo final de esta discusión se puede pensar que el atributo denominado "location" en la clase *LocationContext*

puede derivar en la integración en la aplicación de todo el framework para GIS desarrollado en [Gordillo et al. 97].

4.3 Framework para aplicaciones "context-aware"

En el inciso anterior se estableció una jerarquía de clases de contextos, la cual es la base sobre la cual puede construirse un framework que brinde servicios de más alto nivel y valor agregado a las distintas aplicaciones que utilicen información contextual para su funcionamiento. El principal objetivo es de esta manera liberar a los desarrolladores de realizar una y otra vez tareas de detección y manipulación de información contextual en forma "artesanal" en favor de una manera más "estándar", flexible y potente.

La figura 13 muestra un esquema conceptual de los diferentes elementos que componen dicho framework. Aquí resulta importante aclarar que este framework es utilizado por las aplicaciones en forma "local" (es decir que existe en forma de librería que debe incorporarse en el desarrollo para poder hacer uso de sus funciones). En otras palabras, teniendo en cuenta la arquitectura multi-capas presentada en el capítulo anterior, este framework de contextos se ejecuta en el ambiente provisto por un dispositivo móvil.

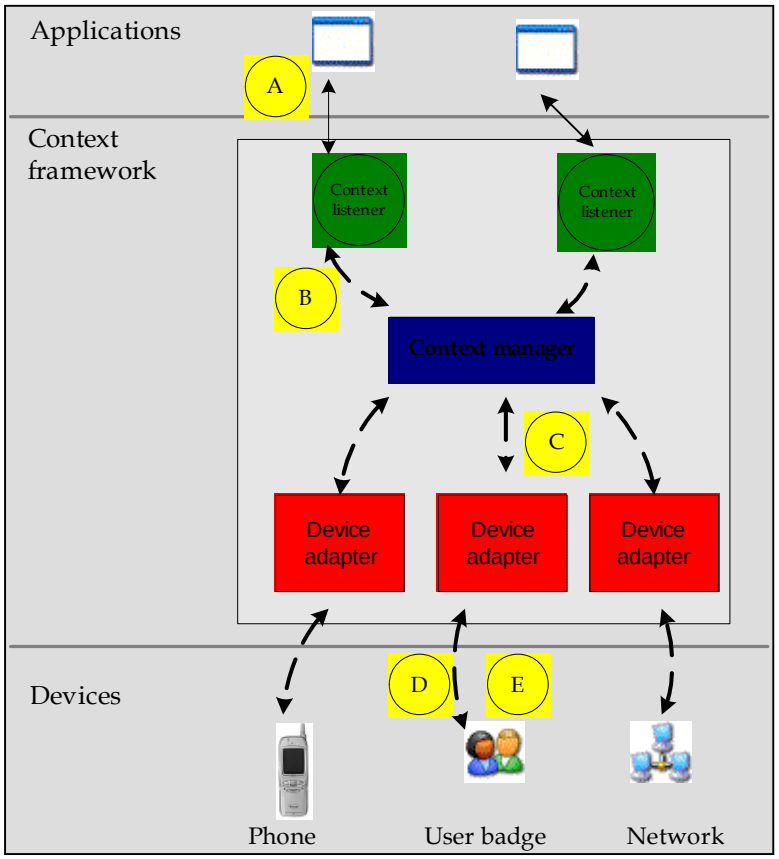

Figura 13: Principales elementos del framework.

A continuación se presenta una breve discusión de los aspectos principales de cada elemento del framework.

- ContextManager: este elemento es responsable del mantenimiento de los diferentes contextos requeridos por las múltiples aplicaciones que se pueden estar ejecutando en el dispositivo móvil. Cada vez que se da un cambio en algunos de los contextos, arbitra los medios necesarios para notificar a las aplicaciones que correspondan. Su implementación se basa en el patrón de diseño *"Singleton"*[26] [Gamma et al. 94] ya que es conveniente asegurarse que en todo momento exista una única instancia de esta clase actuando como "concentrador" de la información obtenida

[26] El patrón de diseño Singleton establece la manera de lograr que una clase solamente tenga un número fijo de instancias (generalmente 1). Como ejemplo se puede considerar un sistema de impresión de documentos en el cual es necesario contar con un "Spooler" de impresión (el cual debe ser único en todo el sistema).

recientemente. Además, este administrador de contextos también puede ser considerado como un *"Mediator"*[27] [Gamma et al. 94] teniendo en cuenta su responsabilidad de gerenciar la colaboración entre las fuentes de eventos y los destinatarios finales de tales notificaciones.

- ContextListeners: estos elementos tienen como principal responsabilidad el notificar a las aplicaciones de los cambios en el contexto. Una aplicación debe "registrar" su interés (representado por un *"listener"*) en un cambio de contexto en particular. Una vez que se produce el cambio esperado, el "listener" enviará una notificación adecuada a la aplicación que lo ha registrado. Existen al menos dos alternativas en Java para que dichos "listeners" notifiquen a la aplicación; las cuales serán comentadas con detalle en la sección "Interacción entre los elementos del Framework y las Aplicaciones". Básicamente los esquemas son:

  - Esquema presente en Java, el cual establece la creación de instancias de clases anónimas[28], donde por cada notificación es necesario crear una nueva clase.
  - Esquema basado en Reflection[29], lo que permite definir una sola clase que es capaz de ser configurada para poder notificar a las aplicaciones de manera más genérica.

[27] El patrón de diseño Mediator indica la creación de objetos que actúan como intermediarios entre otros objetos para desacoplar el alto acoplamiento que pueda existir entre las partes involucradas.

[28] En Java se define una clase anónima como una clase que ha sido creada sin definir un nombre en particular. La idea es crear rápidamente una instancia de dicha clase para los casos en los cuales no se necesite referenciar nuevamente a la clase.

[29] La utilización de Reflection en Java permite la creación de instancias y ejecución de métodos a partir de los nombres de las clases o métodos como strings respectivamente. Por ejemplo es posible solicitar a un objeto de la clase X que ejecute un método y de la siguiente manera:

```
Class X = x.getClass();
Method m = X.getMethod("y");
m.invoke(x);
```

- Adaptadores: actualmente no existe en el mercado un estándar que establezca un formato universal de transferencia de información entre los diferentes tipos de dispositivos. Aún nos encontramos en un esquema sumamente propietario en este sentido, con lo cual es necesario contar con una capa cuya responsabilidad sea estandarizar el formato de la información recibida desde los dispositivos físicos. Esta capa está conformada por los "adaptadores", los cuales traducen la información obtenida a partir de un dispositivo físico (en formato propietario) a un formato estándar interno del framework. Resulta claro que este esquema plantea un conocimiento profundo de los formatos propietarios utilizados. Recientemente Nokia® envió un pedido de estandarización de una API para sensores móviles a la JCP[30] (ver [JSR 256 04]).

- Aplicaciones: estas aplicaciones son las que serán ejecutadas en los dispositivos móviles y que serán las que se beneficien de la obtención de la información contextual. Cada aplicación deberá registrar su interés (a través de los correspondientes "listeners") en los diferentes cambios en el contexto que puedan ser detectados por el framework. Cabe destacar que cada una de estas aplicaciones será responsable de definir un modelo de objetos que se nutra de la información contextual básica enviada a partir del framework. En otras palabras, la presencia de una clase que modela la información relevante de los usuarios en el framework no releva a una aplicación de la necesidad de modelar dicho concepto con mayor profundidad. Cabe destacar que de seguirse

La principal ventaja de Reflection radica en el hecho de que se puede definir en tiempo de ejecución que mensaje se le pedirá a una instancia que ejecute.

[30] JCP: Java Community Process: es un mecanismo mediante el cual la comunidad de usuarios y empresas relacionadas con Java controlan las extensiones a la plataforma y piden nuevas características [JCP 05].

la arquitectura planteada en el capítulo anterior, serán estas aplicaciones las que deberán entablar la comunicación con un servidor para llevar adelante la funcionalidad que se requiera.

- Dispositivos: estos elementos representan en última instancia las fuentes a partir de las cuales se obtendrá la información acerca del contexto. Cabe aclarar que la información obtenida a través de este medio es solamente información sobre aspectos físicos, no así la de carácter lógico. Existirán diferentes tipos de dispositivos, desde teléfonos móviles, GPS, hasta elementos de identificación más novedosos como las placas de identificación personal propuesta por VeriChip [VeriChip 05].

- Eventos (no se muestran explícitamente en la figura): en la figura anterior se muestran varias etiquetas (A, B, C, etc.) las cuales representan un flujo conceptual de la información. Dicha información está encapsulada en ciertas entidades (denominadas "eventos") las cuales sirven como repositorio de cierta información:
    - Origen de la información capturada, es decir desde qué dispositivo se ha obtenido la información.
    - Información propiamente dicha, por ejemplo la ubicación.
    - Precisión o calidad de la información obtenida para la captura en particular.

  La versión inicial del framework sólo cuenta con un tipo de evento genérico, pero es evidente la necesidad de contar con algunos tipos de eventos más especializados a medida que más aplicaciones se vayan incorporando. Como eventos especializados puede tomarse a modo de ejemplo aquellos que representan la

obtención de un nuevo contexto o los que representan un cambio en un contexto ya detectado.

4.4 Diagrama de colaboración

En esta sección se presenta un diagrama de colaboración a fin de ilustrar el esquema bajo el cual los diversos elementos trabajan en conjunto para llevar adelante las responsabilidades antes citadas.

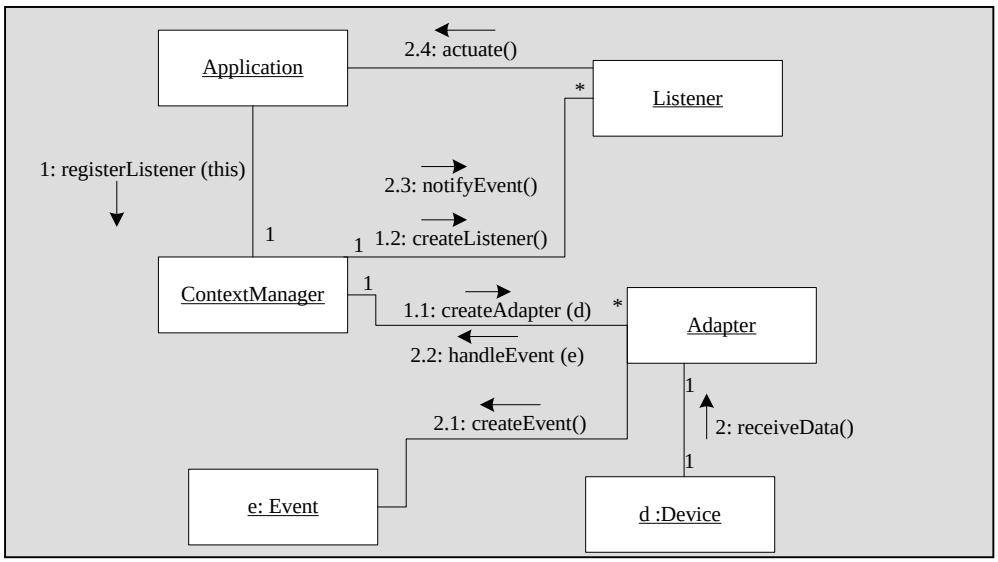

Figura 14: Diagrama de colaboración del framework.

El diagrama presentado en la figura 14 contiene dos flujos importantes:

- Flujo 1: está relacionado con la secuencia de colaboración llevada a cabo a raíz de una petición de registro de una aplicación para un contexto determinado. En caso de que ya no exista un adaptador específico para el tipo de dispositivo capaz de detectar tal contexto, el administrador de contextos (ContextManager) crea una nueva instancia de un adaptador y lo configura

apropiadamente para poder traducir la información recibida desde el dispositivo físico.

- Flujo 2: este flujo está vinculado con el manejo de las notificaciones que se producen cuando un dispositivo físico detecta un cambio en un contexto determinado; a raíz de lo cual se notifica al adaptador correspondiente y se inicia el manejo del evento creado para representar tal hecho hasta culminar en el envío de una notificación a la aplicación que ha registrado su interés en los cambios del contexto.

4.5 Diagrama de clases

A continuación se brinda un diagrama de clases que se corresponde con los elementos citados en los puntos anteriores.

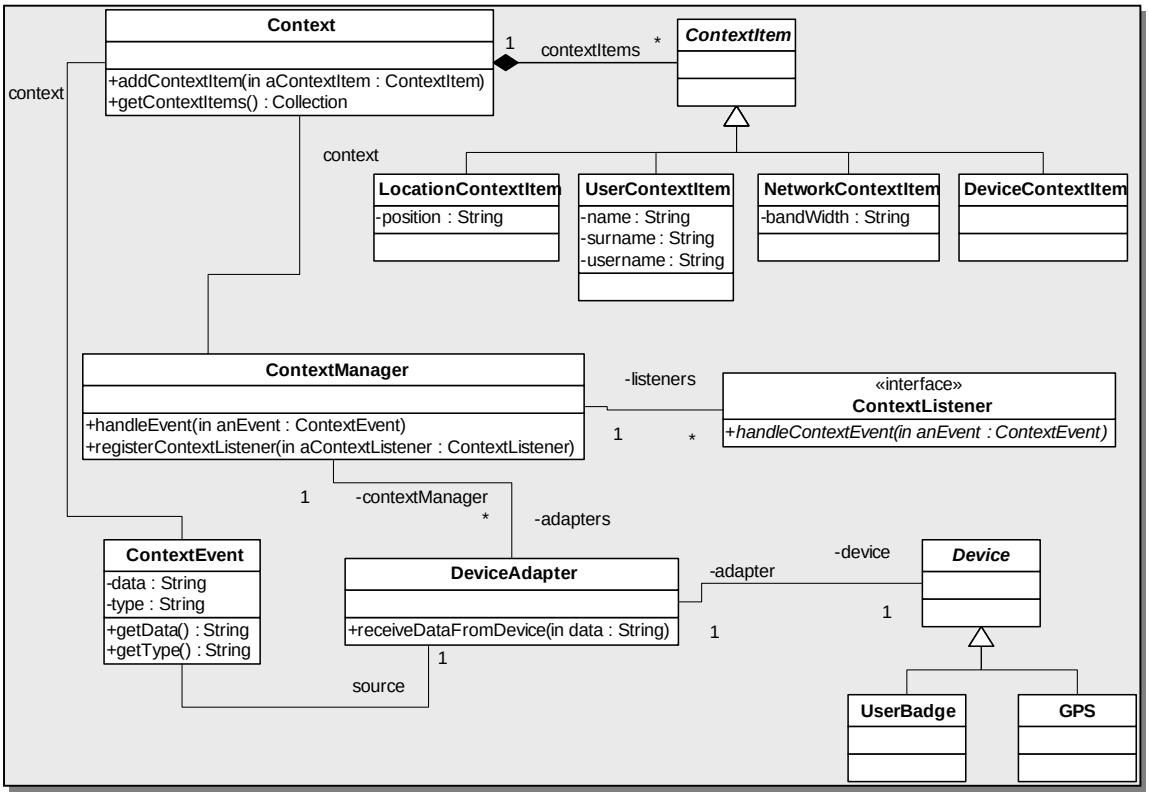

Figura 15: Diagrama de clases del framework.

El diagrama de clases presentado en la figura 15 constituye una extensión al diagrama básico presentado en la figura 12, ya que incorpora los conceptos relacionados con el framework propiamente dichos (clases *Device*, *DeviceAdapter*, etc.).

Es necesario notar una diferencia más entre ambos diagramas; en la figura 15 la clase *Context* no es la base de una jerarquía de contextos más especializados, sino que es una agregación de ítems de contextos, los cuales pueden ser utilizados para representar información contextual acerca del usuario, red, posición, etc. El cambio entre ambos diagramas se debe a que en la segunda alternativa resulta más simple el manejo de las actualizaciones que se producen cuando varios aspectos contextuales son detectados casi al mismo tiempo.

4.6 Diagrama de Secuencia

En esta sección se presenta un diagrama de secuencia para demostrar de manera clara la forma en que interactúan los objetos del framework a la hora de notificar a una aplicación de los cambios detectados en un contexto en particular.

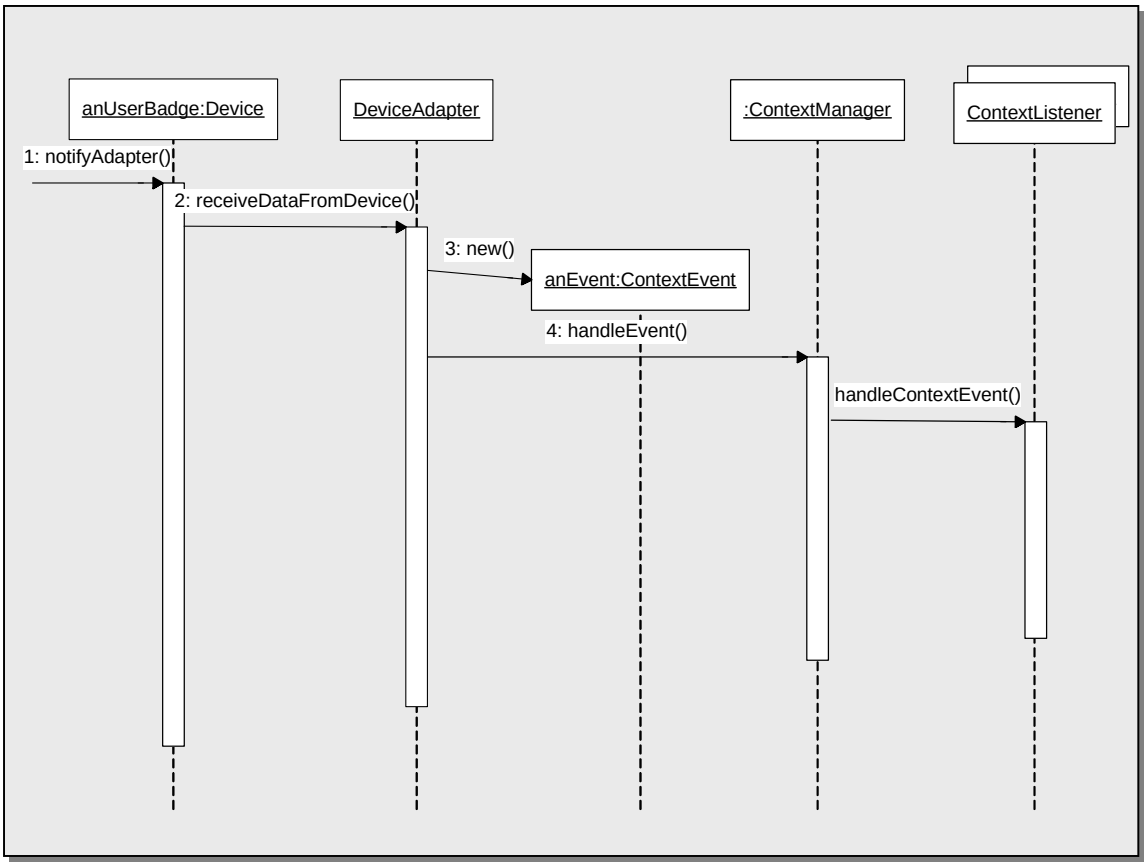

Figura 16: Diagrama de secuencia de una notificación.

4.7 Interacción entre los elementos del framework y las aplicaciones

Esta sección presenta algunos "snippets"[31] de código para clarificar la forma en la cual las aplicaciones middlets utilizan las clases del framework propuesto.

4.7.1 Registración de las aplicaciones

Tomando el lenguaje Java como lenguaje de implementación, las aplicaciones middlets tienen dos posibilidades al momento de registrar su interés en cierto cambio contextual:

[31] Snippet: trozo de código que presenta un detalle acerca de un concepto implementado con algún lenguaje de programación.

- Mediante la implementación directa de la interface *ContextListener*: en esta alternativa es la clase misma que representa la aplicación la que debe implementar la interface *ContextListener*.

```
public class MyMiddlet extends Middlet implements ContextListener{

  public MyMiddlet(ContextManager manager) {
   manager.registerContextListener(this);
  }

  public void handleContextEvent (ContextEvent anEvent) {

   //código de manejo de la notificación del evento

  }
  //otros métodos del middlet
}
```

Dada la anterior definición, ahora la aplicación debe registrarse siguiendo la forma especificada en el constructor[32] de la clase (en negritas).

- Mediante la utilización de clases anónimas: en esta alternativa no es necesario que la aplicación middlet implemente la interface, sino que se crea dinámicamente una instancia de una clase definida en tiempo de ejecución que implementa la interface necesaria.

```
public class MyMiddlet extends Middlet {

  public MyMiddlet(ContextManager manager) {
   manager.registerContextListener(new ContextListener() {

            public void handleContextEvent (ContextEvent anEvent) {

            //código de manejo de la notificación del evento

            }
   });
  }
  //otros métodos del middlet
}
```

[32] Constructor: método especial de una clase Java mediante el cual se puede crear una nueva instancia.

Ambas alternativas son perfectamente válidas, recayendo en el programador la selección que le resulte más legible.

Una vez que un dispositivo físico genere información, ésta será traducida a un formato estándar por un adaptador (el cual creará un evento) y en última instancia será notificada la aplicación para que ejecute el código presente en las líneas dentro del método *"handleContextEvent()"*.

4.8 Conclusiones

En este capítulo se presentó un framework orientado a objetos para la detección de cambios contextuales de manera genérica y que puede ser utilizado en la construcción de aplicaciones móviles bajo el estándar J2ME [J2ME 05]. La intención primordial de este framework es liberar a los programadores de la tediosa tarea de detectar y definir la manera en la cual se manejan los cambios contextuales.

Este framework se encuentra en su etapa inicial, debiendo extenderse para soportar nuevos tipos de dispositivos, integrar contextos lógicos para manejar de manera estándar toda la información contextual, así como otros aspectos que serán citados en la sección "Trabajo Futuro".

Capítulo V: Implementación de un prototipo: CAWI

La utilización de prototipos está reconocida como una excelente estrategia para detectar requerimientos funcionales y no funcionales de sistemas de software, además de proveer un producto tangible con el cual experimentar y eventualmente demostrar la factibilidad técnica de una arquitectura. A continuación se detallan las características de un prototipo desarrollado utilizando el framework propuesto en este trabajo.

5.1 Introducción

En este capítulo se presentarán las experiencias y conclusiones adquiridas en el desarrollo de un prototipo funcional que integra tanto las ideas y propuestas de arquitectura presentes en el capítulo III como el framework de contextos presentados en el capítulo IV.

El desarrollo del prototipo estuvo motivado en primera medida por la necesidad de demostrar en forma clara la utilidad real obtenida a partir de la incorporación del framework para aplicaciones sensibles al contexto. En segundo lugar, es necesario probar en una aplicación con requerimientos reales la factibilidad técnica al momento de implementar este tipo de aplicaciones basándose en una estrategia de distribución en capas tal cual lo establecido en el capítulo anterior.

El alcance de esta prueba "de campo" es sin embargo limitado a raíz de ciertas restricciones razonables. Por un lado tenemos la limitación lógica al

alcance de la funcionalidad provista por el prototipo, lo cual establece cierta falta de noción de la adaptabilidad a cada una de las circunstancias a las cuales estaría sometida la arquitectura al ser utilizada en un ambiente real "de producción". Por otro lado, en la realización de esta tesis no se utilizaron dispositivos físicos reales, sino que por el contrario se simularon los eventos relacionados con éstos; simplificando nuevamente el ambiente de prueba.

Como dominio de aplicación se seleccionó una herramienta de administración de bugs[33], pedidos de cambios y otros temas relacionados con proyectos de software que será presentada en la siguiente sección. Esta herramienta se utilizó como base para establecer los requerimientos funcionales básicos que debería presentar una herramienta móvil, la cual utiliza además información contextual para incrementar su utilidad final. En la sección "Servidor CAWI" se presentan los detalles de la construcción de la parte correspondiente al servidor de dicho prototipo. A continuación, en la sección "Aplicación móvil" se relatan las características de mayor relevancia relacionadas con la aplicación móvil en sí misma. Como se mencionó en el párrafo anterior, este prototipo se basa en simulación para la ejecución de la aplicación. La sección "Simulador" contiene una descripción del simulador desarrollado como soporte del prototipo. Finalmente se presentan algunas conclusiones obtenidas a partir de la experiencia de desarrollo del prototipo.

5.2 ITeM

La herramienta ITeM (Issue Tracking e-Management[34]) permite a los líderes y demás integrantes de los equipos de desarrollo de software el seguimiento de las distintas asignaciones. Dichas asignaciones o "tareas" van desde requerimientos tomados de la especificación de requerimientos de las

[33] Bug: error detectado en el funcionamiento de un sistema de software.
[34] Issue Tracking e-Management: Administración electrónica de temas, por sus siglas en inglés.

etapas previas hasta los bugs detectados en las etapas posteriores del desarrollo o del mantenimiento de software ya creado. Es vital la utilización de este tipo de herramientas para llevar adelante una correcta administración y un seguimiento fino de la evolución de tales tareas. Actualmente en el mercado se encuentran varias herramientas que sirven a este fin, destacándose Jira [Jira 05] y Mantis [Mantis 04], los cuales representan versiones comerciales y "open-source" respectivamente.

Esta herramienta tiene una interfase gráfica Web, es decir que para acceder a su funcionalidad es necesario contar con un navegador de Web. La herramienta presenta dos áreas lógicamente divididas:

- La primera corresponde a las funciones que puede ejecutar un usuario cuando se encuentra fuera de cualquier proyecto en particular. En esta área el usuario puede administrar sus preferencias de navegación, así como listar sus proyectos (y otras funciones de menor importancia).
- La segunda área corresponde a los proyectos de un usuario dado. Cada usuario puede ser miembro de varios proyectos, dentro de los cuales puede tener diferentes roles o perfiles (los cuales definen en última instancia el conjunto de operaciones permitidas).

Los proyectos definen cada uno un "workflow" o grafo de estados por los cuales puede pasar un requerimiento o "issue"[35] dado. Es importante notar que dicho grafo de estados es completamente dinámico; en otras palabras, éste puede ser configurado o alterado en tiempo de ejecución por el líder del proyecto para adecuar el funcionamiento del sistema a las necesidades puntuales de cada proyecto. Esta herramienta además presenta características

[35] El término "issue" será utilizado de ahora en adelante para referirse a requerimientos, bugs u otras tareas que pudiesen ser de interés para un equipo de desarrollo y que puedan ser monitoreadas y asignadas a distintos responsables a lo largo del tiempo.

de flexibilidad importantes, ya que permite por ejemplo la definición dinámica de estados (y para cada uno de éstos permite la definición del conjunto de atributos con los que cuenta el estado).

Quizás la funcionalidad provista por esta herramienta que es utilizada con más asiduidad es la que permite listar los requerimientos asignados a un usuario en un momento dado. La figura 16 presenta una captura de la pantalla que muestra el listado de los requerimientos de un proyecto en particular.

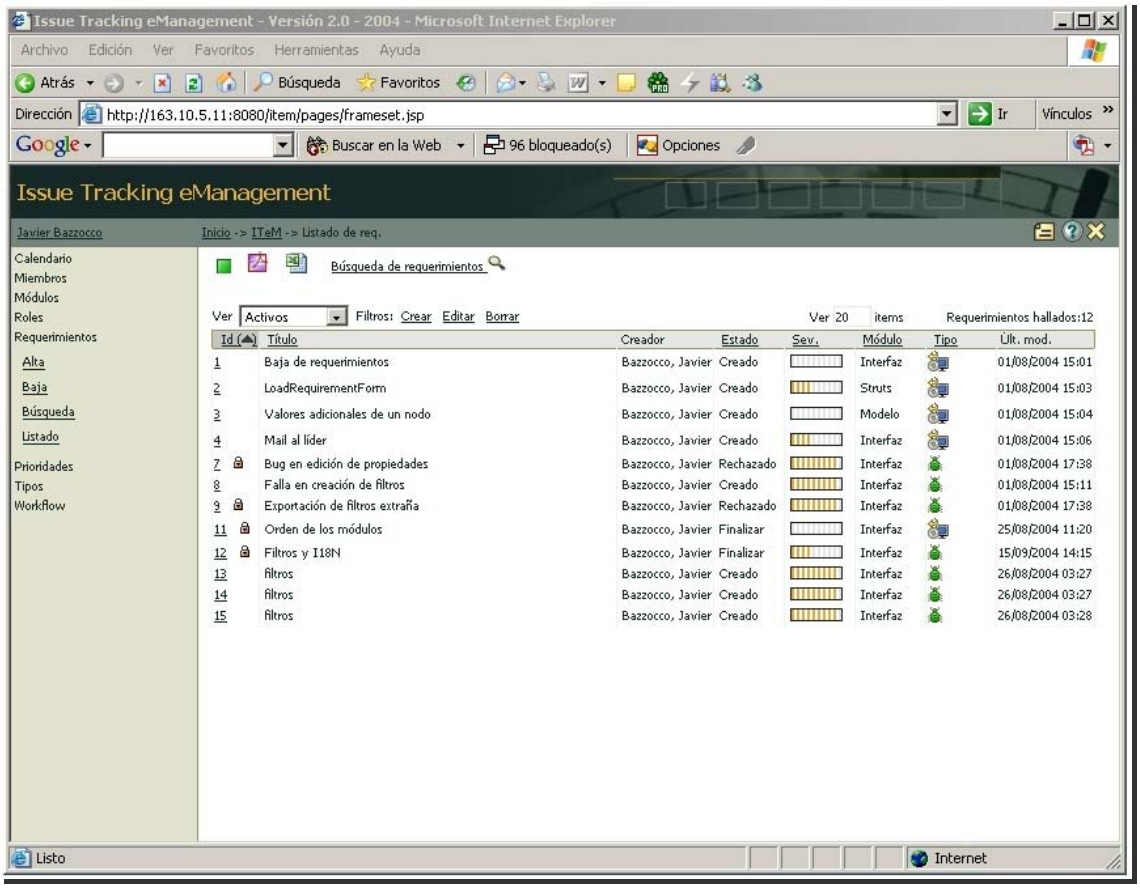

Figura 17: Visualización de la lista de requerimientos en el ITeM.

La siguiente lista presenta un breve detalle de las principales características con las que cuenta esta herramienta:

- Capacidad de manejar múltiples proyectos.

- Interfase Web, totalmente internacionalizada (soporte de varios idiomas).
- Cada proyecto permite la definición de un workflow personalizado.
- Workflow dinámico de estados para los requerimientos.
- Estados del workflow con atributos definibles en tiempo de ejecución.
- Capacidad de ejecutar tareas o actividades en cada estado del workflow definidas en tiempo de ejecución.
- Implementada con tecnología de punta y open-source:
    - Modelo totalmente implementado con Java como POJOs[36].
    - Persistencia orientada a objetos, mediante la utilización de Hibernate [Hibernate 02].
    - Interfase creada con Struts [Struts 05] y Servlets [Servlets 05].
    - Base de datos MySql [MySql 05].
    - Servidor Jakarta Tomcat [Tomcat 03].
- Notificación por emails de la asignación de requerimientos.
- Definición dinámica de tipos de requerimientos.
- Definición dinámica de prioridades para los requerimientos.
- Aspectos de personalización de la interfase gráfica:
    - Cada listado "recuerda" la cantidad de ítems que debe mostrar.
    - Cada listado recuerda el filtro que se debe aplicar para recuperar los ítems a ser mostrados.
    - Cada listado recuerda por cuales campos se debe ordenar los ítems.

[36] POJO (Plain Old Java Object): filosofía que establece el desarrollo de clases que pueden ser persistentes, pero que sin embargo no contienen ninguna indicación de esta capacidad. Debido a este hecho es posible cambiar la forma de lograr la persistencia sin alterar las clases de negocio mismas.

Para finalizar esta sección, es preciso notar que para el desarrollo del prototipo primó la intención de no modificar en el aspecto más mínimo la herramienta ITeM.

5.3 Servidor CAWI

La intención de no modificar la herramienta ITeM impuso de manera casi directa la necesidad de implementar un servidor especializado que proveyera acceso al conjunto de funcionalidad del ITeM, al tiempo que se aumenta la usabilidad del mismo mediante la integración de los aspectos de "sensibilidad al contexto". Este servidor incorpora la noción de "ubicación" a los conceptos ya manejados por el ITeM. La ubicación está relacionada con dos elementos importantes del dominio de la aplicación, por un lado la ubicación de un usuario en particular (por ejemplo un líder de proyecto) y por otro lado la ubicación en la cual se desarrollan los diferentes proyectos de desarrollo de software. Con el fin de acotar el desarrollo del prototipo, la ubicación es representada por una "habitación", en la cual se desarrollan los proyectos y en las cuales es posible detectar la presencia de un usuario. Una vez que se detecta un usuario en una habitación dada, se utiliza dicha información contextual para realizar un filtrado de los requerimientos relacionados con los proyectos localizados en la habitación detectada. Es importante notar que toda la funcionalidad sigue siendo ejecutada por el servidor ITeM integralmente, por lo que puede pensarse el servidor CAWI como un "Decorator"[37] [Gamma et al. 94] de los servicios provistos por el ITeM.

[37] El patrón de diseño Decorator establece la creación de un objeto que "envuelve" la funcionalidad de otro objeto, para poder agregar nueva funcionalidad para los clientes de dicho objeto, sin tener que alterar el objeto inicial.

El servidor CAWI[38] fue construido siguiendo las ideas planteadas en el Capítulo IV con el objetivo final de probar la factibilidad técnica de la arquitectura que fuera planteada en dicho capítulo. Se han utilizado todos los elementos detallados en las diferentes secciones del capítulo previo, como por ejemplo:

- Codificadores (encoders) para codificar las notificaciones que son enviadas al cliente.
- Decodificadores (decorders) para interpretar los mensajes recibidos desde el cliente.
- Mensajes (messages) para intercambiar información entre las "puntas" de la aplicación sensible al contexto.
- Administradores de comunicaciones (serverCommManager), el cual es responsable de la administración de todos los detalles concernientes a la comunicación.
- Comandos (serverCommands), que implementan la lógica de negocios de la aplicación.

La siguiente figura presenta las clases que componen el servidor CAWI:

- Clase *CommServlet*: este servlet es la puerta de entrada al servidor CAWI. La única instancia de esta clase recibe los pedidos del cliente representado por el Simulador y los redirige al administrador de comunicaciones para su interpretación y posterior ejecución.

- Clase *DeviceCommServlet*: este servlet es la puerta de entrada para los pedidos enviados por la aplicación que se está ejecutando en el dispositivo móvil. La necesidad de contar con dos servlets distintos nace de las diferencias de protocolo y capacidades

[38] CAWI (Context AWare ITeM): ITeM sensible al contexto.

presentes en un dispositivo móvil y en una aplicación desktop[39] como el Simulador.

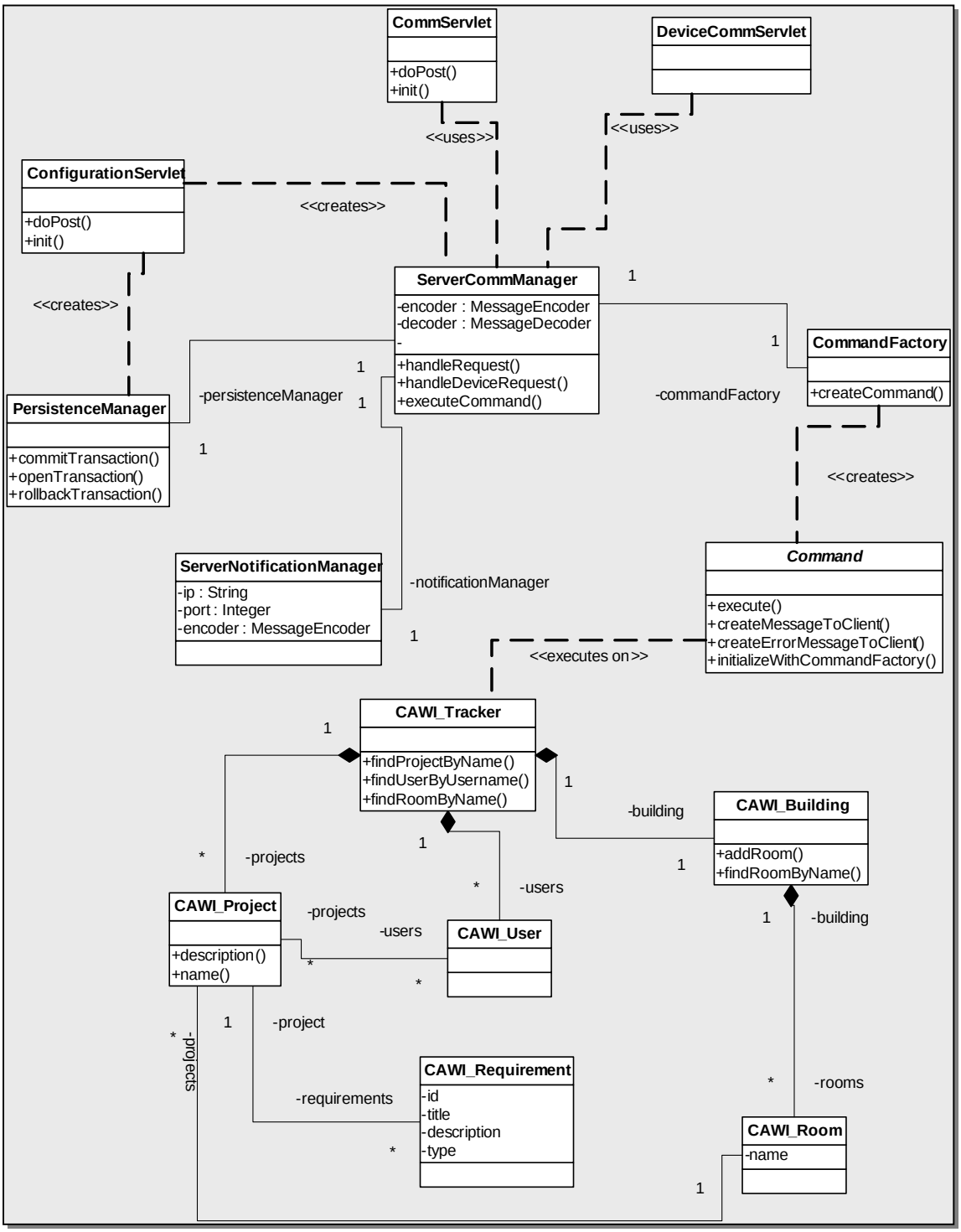

Figura 18: Modelo del servidor CAWI.

[39] Aplicación desktop es un sinónimo para aplicación de tipo pesada.

- Clase *ConfigurationServlet*: este servlet es utilizado para configurar e inicializar todos los recursos necesarios para la correcta ejecución del servidor. Por ejemplo se configura y conecta el administrador de persistencia con el repositorio de almacenamiento físico.

- Clase *PersistenceManager*: esta clase es responsable de encapsular todo el protocolo necesario para la administración de los aspectos relacionados con la persistencia de la información manejada por el servidor CAWI. Establece el protocolo que puede ser utilizado por los administradores de comunicaciones para abrir, cerrar y eventualmente dar marcha atrás a transacciones que demarcan la interacción con el repositorio de almacenamiento. Esta clase implementa el patrón de diseño *"Singleton"* [Gamma et al. 94].

- Clase *ServerCommManager*: esta clase define el comportamiento necesario para administrar la comunicación entre los clientes (ambos tipos de clientes, es decir el simulador y la aplicación móvil) y el servidor. Absolutamente toda la comunicación pasa por la única instancia de esta clase (patrón de diseño *"Singleton"* [Gamma et al. 94]) a fin de garantizar el tratamiento estándar de los pedidos de los clientes. En forma resumida, la forma de trabajo de este administrador (en conjunto con el servlet de comunicación) es la siguiente:

1. El cliente envía un pedido al servidor.
2. El servlet de comunicaciones recibe el pedido y arma un string con el contenido del pedido para invocar al administrador de comunicaciones.
3. El administrador de comunicaciones decodifica el mensaje recibido.

4. El administrador de comunicaciones crea un comando interactuando con el factory de comandos.
5. El administrador de comunicaciones ejecuta el comando sobre el modelo.
6. El comando se ejecuta sobre el modelo (posiblemente alterándolo) y retorna un resultado.
7. El administrador de comunicaciones utiliza el codificador para codificar la respuesta del comando y lo envía al administrador de notificaciones.
8. El administrador de notificaciones envía el mensaje codificado al cliente.

- Clase *ServerNotificationManager*: esta clase agrupa el conocimiento necesario para enviar las notificaciones a los diferentes clientes. En la implementación actual, el servidor por sí mismo nunca genera una notificación hacia un cliente, sino que por el contrario toda notificación tiene origen en un mensaje desde un cliente hacia el servidor. Con el fin de asegurarse la existencia de una sola instancia de esta clase en todo momento, se ha adoptado para su implementación el patrón de diseño *"Singleton"* [Gamma et al. 94].

- Clase *CommandFactory*: esta clase es responsable de definir la forma en la cual se crean cada una de las instancias de los comandos que son ejecutados sobre el modelo mantenido en el servidor a raíz de un pedido recibido desde algún cliente. Se basa en la técnica de *"double-dispatching"* para realizar dicha tarea, a fin de no definir en forma rígida y estática el nombre de las clases de los comandos que se crean.

- Clase *Command*: esta clase abstracta define el protocolo que deberán implementar todos los comandos que sean creados para implementar la lógica de la aplicación. Esta clase define también la forma básica de colaboración con la clase CommandFactory para inicializar cada una de las instancias de los comandos creadas por

esta última. Esta clase además representa una implementación del patrón de diseño *"Command"* [Gamma et al. 94].

- Clase CAWI_Tracker: esta clase representa al sistema en sí mismo. Es tanto un *"Facade"* [Gamma et al. 94] del modelo como un *"Root Object"* [Loomis 95] que permite acceder al resto de los otros objetos del mismo. Asimismo puede ser considerada como la implementación del patrón de diseño *"Decorator"* [Gamma et al. 94] ya que agrega el concepto de la ubicación a la clase Tracker presente en el diseño del ITeM original.

- Clase *CAWI_Building*: esta clase representa un edificio en el cual se encuentran las habitaciones que son administradas por el servidor CAWI, en las cuales se desarrollan proyectos de software y es posible detectar la presencia de los usuarios.

- Clase *CAWI_Project*: esta clase es un "decorator" [Gamma et al. 94] de los proyectos que son administrados por el servidor ITeM. Estos proyectos tienen la información básica y reenvían todos los pedidos de mayor información u acción a los proyectos mantenidos en el ITeM.

- Clase *CAWI_User:* esta clase representa a un usuario que puede ser identificado en una ubicación determinada. A su vez, este usuario representa un "decorator" [Gamma et al. 94] de los usuarios definidos en el servidor ITeM, ya que en el servidor CAWI solamente se mantienen los datos mínimos y esenciales para poder identificar a los usuarios.

- Clase *CAWI_Room*: esta clase define el concepto de una "habitación", en la cual no sólo se desarrollan los proyectos de software, sino que además es posible detectar la presencia de los usuarios mediante algún dispositivo de detección.

- Clase CAWI_Requirement: esta clase representa un "decorator" [Gamma '94] de los requerimientos que son administrados y monitoreados por el servidor ITeM. Esta clase contiene solamente una definición básica de un requerimiento. Todo pedido de mayor información es automáticamente reenviado al servidor ITeM para su resolución.

Con respecto a las tecnologías utilizadas, se pueden citar las más importantes:

- Servlets [Servlets 05], para establecer la comunicación con el servidor sobre el protocolo HTTP [HTTP 96].
- Base de datos MySql [MySQL 05], para almacenar toda la información manejada por este servidor.
- Mapeador Objeto / Relacional Hibernate [Hibernate 02], para persistir y posteriormente recuperar las instancias del modelo.

5.4 Simulador

Con el fin de probar la factibilidad técnica real tanto de la arquitectura para aplicaciones móviles (capítulo III) como del framework de contextos (capítulo IV) se desarrolló este prototipo. Debido a la carencia de dispositivos físicos de detección de usuarios, así como de transmisión de información, se decidió la implementación del prototipo basado en una simulación.

El simulador presenta una interfase gráfica basada en Swing [Swing 05] la cual permite entre otras funciones:

- Conexión al servidor CAWI para obtener la información manejada por éste.

- Definición de nuevas habitaciones.

- Permite obtener información acerca de un usuario en particular.

- Permite la selección de un usuario dado, a raíz de lo cual se envía una notificación a la aplicación móvil a un "socket"[40] predefinido que simula la detección de un usuario en las cercanías del teléfono móvil en el cual se está ejecutando la aplicación sensible al contexto.

- Permite posicionar un usuario en una habitación definida, lo que resulta en el envío de una notificación a un "socket" de la aplicación móvil simulando de esta manera la detección de la posición por parte de un dispositivo físico de detección.

La siguiente figura muestra una captura de la interfase gráfica del simulador, la cual muestra un solo usuario que no se encuentra ubicado en ninguna habitación en particular.

[40] Socket: canal de comunicación que permite establecer una comunicación con aplicaciones que están ejecutándose en otras máquinas.

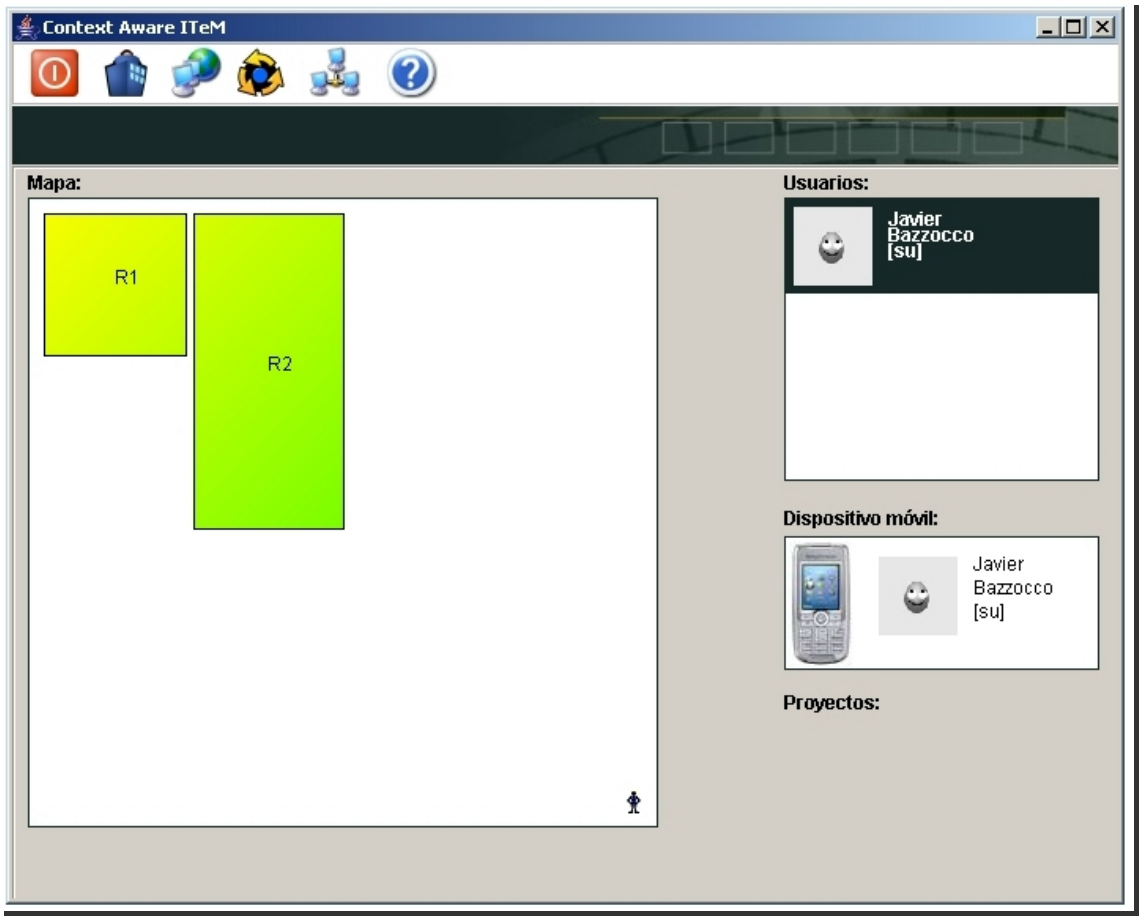

Figura 19: Interfase gráfica del simulador del prototipo.

La figura 20 muestra la interfase gráfica que permite agregar una nueva habitación al conjunto de habitaciones definidas para el edificio bajo control del servidor CAWI.

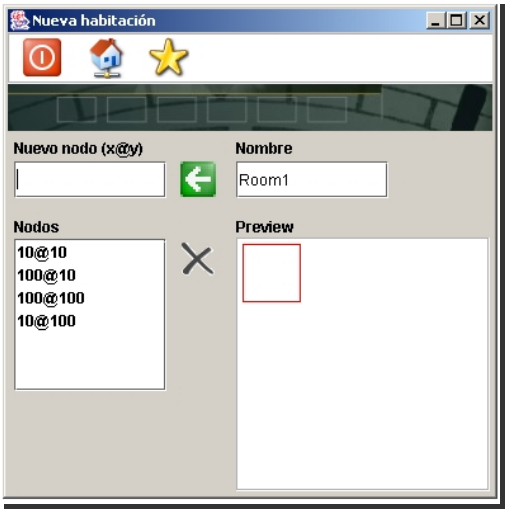

Figura 20: Interfase gráfica para dar de alta habitaciones.

5.5 Aplicación móvil

La aplicación móvil permite la utilización de toda la funcionalidad y potencia de la herramienta ITeM, al tiempo que agrega la posibilidad de utilizar información de carácter contextual para enriquecer la utilidad final del ITeM.

La información contextual tiene hoy por hoy dos claros usos en el prototipo:

1. Identificar de manera automática al usuario que ha tomado el dispositivo móvil, de manera que no sea necesario solicitar dicha información mediante una interfase gráfica en la que éste debe ingresar su nombre de usuario y clave.

2. Tomar conocimiento de la ubicación física del usuario mediante el uso de dispositivos físicos de detección, para poder filtrar los requerimientos que pertenecen a los proyectos que se desarrollan en la misma habitación en la que se encuentra el usuario.

Actualmente la funcionalidad provista por la aplicación móvil es la siguiente:

1. Cuando no se detecta a ningún usuario en las cercanías del dispositivo móvil (teléfono celular), la aplicación presenta la interfase gráfica que se muestra en la figura 21.

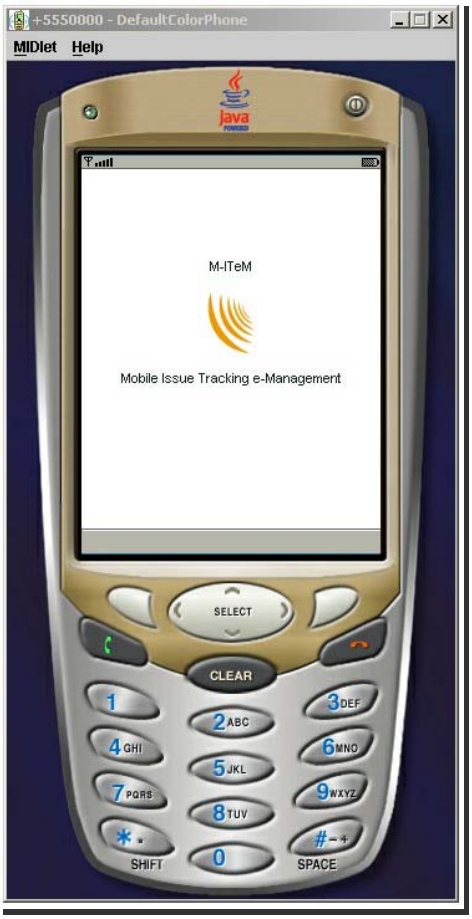

Figura 21: Aplicación móvil preparada para iniciar su ejecución.

2. Una vez que se detecta a un usuario en particular (simulando la detección del mismo a partir del envío de una notificación desde el simulador a un socket de la aplicación móvil), la aplicación móvil envía un mensaje[41] de inicio de sesión con la información contextual relacionada con el usuario detectado.
3. El servidor CAWI responde enviando un listado de los proyectos en los cuales trabaja el usuario actualmente (ver figura 22). En este punto el usuario puede seleccionar cualquier proyecto para comenzar su trabajo.

[41] El mensaje enviado es una instancia de la clase *Message* perteneciente al paquete *middleware.comm.messages* presentado en el capítulo anterior.

Figura 22: Listado de proyectos de un usuario

4. Si el usuario (todavía viendo el listado de sus proyectos) es detectado ingresando en alguna habitación (nuevamente mediante una simulación que envía una notificación desde el servidor a un socket de la aplicación móvil), dicha aplicación envía un mensaje al servidor con la información contextual relacionada con la posición actual (habitación) del usuario.
5. El servidor CAWI utiliza la información de la posición del usuario para filtrar todos los proyectos que se desarrollan en la misma habitación, y en el caso de que en la habitación solamente se lleve a cabo un proyecto, envía una notificación al cliente que contiene la información relacionada con el proyecto en particular.

6. La aplicación móvil muestra la pantalla de bienvenida del proyecto recibido (ver figura 23), en la cual el usuario puede seleccionar una operación para ejecutar (dependiendo esto último de los roles que tenga asignado el mismo dentro del proyecto en cuestión).

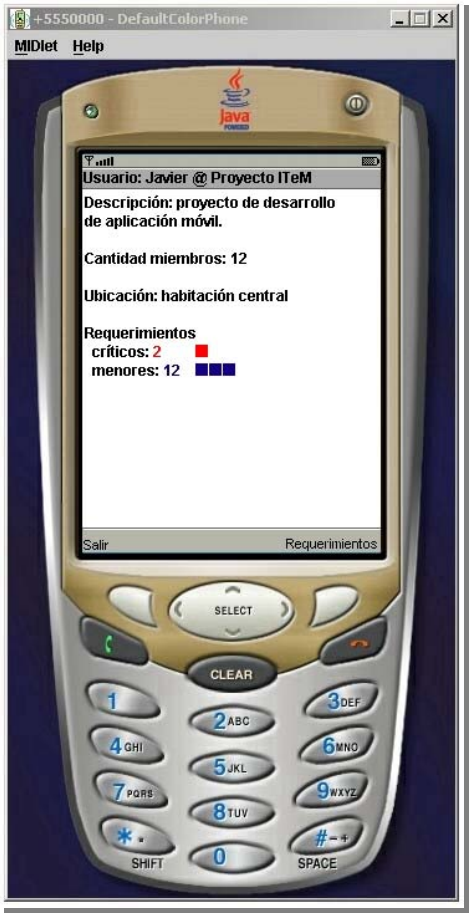

Figura 23: Pantalla de resumen del estado de un proyecto.

7. Si en cualquier momento el usuario deja el dispositivo móvil, automáticamente la aplicación retorna al estado mostrado en la figura 21.

Evidentemente este conjunto básico de funcionalidad provista por el prototipo es escasa en términos de una aplicación real; sin embargo, alcanza

para demostrar el valor agregado que se incorpora al utilizar información de carácter contextual para mejorar la experiencia de uso del usuario final.

En este punto es necesario aclarar que, si bien se siguieron en lo posible las indicaciones a nivel arquitectura planteadas en el capítulo III, el estado actual de la tecnología seleccionada para la implementación de dicha aplicación dista de ser óptimo. Este último hecho hace que en la implementación no se hayan podido incluir todos los elementos propuestos para la capa del cliente, sino que por el contrario es relativamente reducido el conjunto de dichos elementos presentes en el prototipo. A continuación se presentan varios diagramas de clases[42] que contienen los principales elementos desarrollados como parte de esta aplicación móvil prototípica.

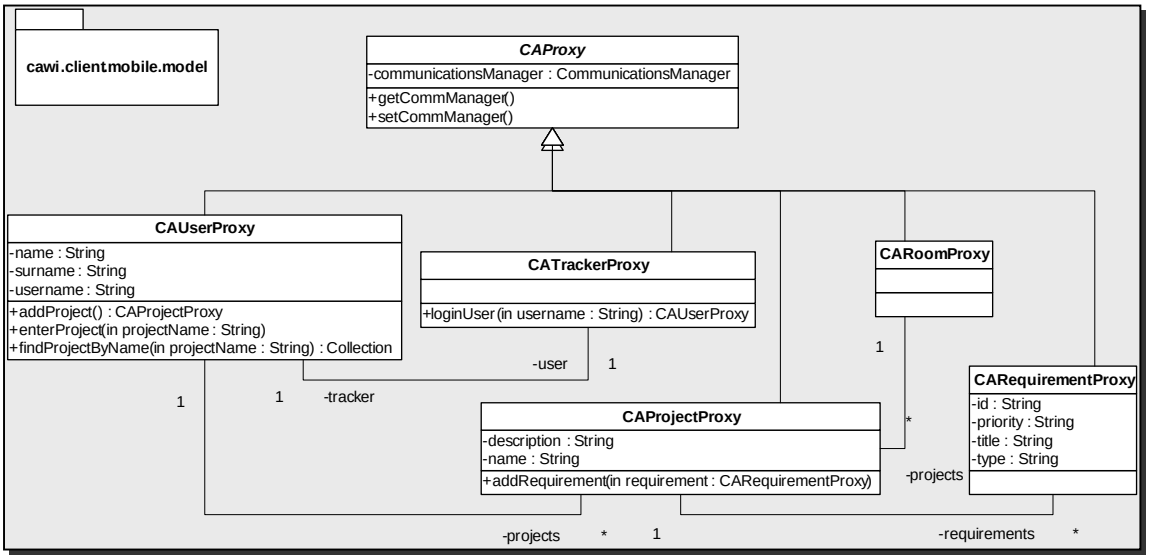

Figura 24: Diagrama de clases del modelo de la aplicación móvil.

- Clase *CAProxy*: esta clase representa el tope de la jerarquía de objetos que son utilizados como "proxies" al modelo mantenido en el servidor CAWI. Establece cierto comportamiento en común

[42] Se presentan varios diagramas de clases a fin de mantener la claridad en la explicación de cada uno de los mismos; sin embargo, aquellos que contienen clases ya definidas (como la clase *Message*) no serán presentados nuevamente.

para todas las subclases. Todos los objetos pertenecientes al modelo del lado del cliente móvil deben ser instancias de subclases de esta clase abstracta (es una implementación del patrón de diseño "*Proxy*" [Gamma et al. 94]).

- Clase *CAUserProxy*: esta clase representa al usuario que está utilizando actualmente el sistema. Todo el comportamiento provisto por esta clase deriva finalmente en una llamada remota al servidor CAWI.

- Clase *CATrackerProxy*: esta clase representa un "proxy" al objeto raíz del modelo mantenido en el servidor CAWI. A partir de este objeto es posible acceder al resto de los objetos que componen el modelo del cliente móvil.

- Clase *CAProjectProxy*: los proyectos mantenidos en el servidor CAWI son representados localmente gracias a instancias de esta clase. Todo el protocolo provisto por esta clase termina en última instancia como una invocación remota al proyecto real mantenido en el servidor.

- Clase *CARoomProxy*: en el modelo del cliente móvil se representan con esta clase las diferentes habitaciones que tiene el edificio administrado por el servidor CAWI.

- Clase *CARequirementProxy*: esta clase permite representar en forma local los requerimientos mantenidos en los servidores CAWI e ITeM. A través de esta clase es posible enviar, en forma independiente, mensajes al servidor.

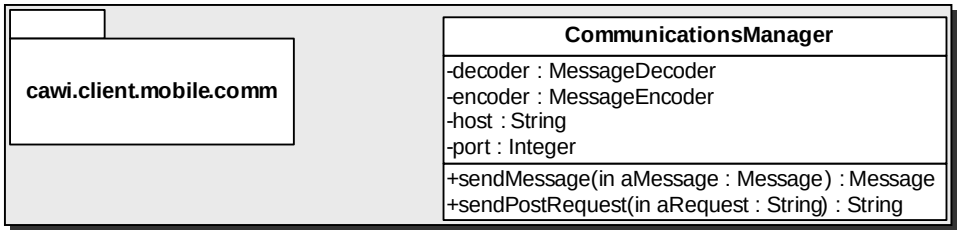

Figura 25: Diagrama de clases de la capa de comunicaciones móvil.

- Clase *CommunicationsManager*: esta clase representa un claro ejemplo de las restricciones en la implementación de la capa del cliente presentada en el capítulo anterior. Debido a las limitaciones en las tecnologías presentes en J2ME [J2ME 05], toda la "capa de comunicación" en la aplicación móvil se reduce a esta única clase que lleva adelante la comunicación desde este cliente hacia el servidor CAWI. Los "proxies" del modelo local envían mensajes al servidor a través de esta clase, la cual colabora con los codificadores y decodificadores (actualmente solamente existen decodificadores y codificadores para el formato XML) para enviar los requerimientos hacia el servidor. Es necesario notar que la implementación actual se encuentra en una etapa primitiva, ya que no implementa aspectos tales como seguridad, ni reintento de envío de mensajes por ejemplo.

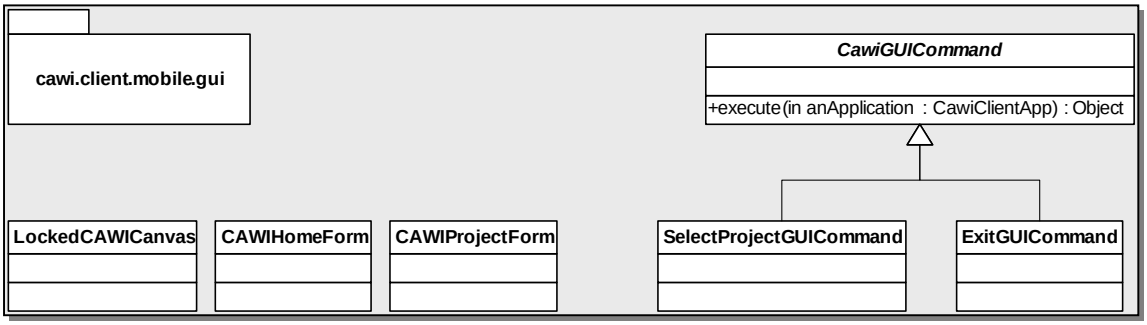

Figura 26: Diagrama de clases de las interfases gráficas y sus comandos.

La implementación de este paquete tiene dos razones fundamentales:

- La primera tiene que ver con la necesidad de agrupar las clases que representan las diferentes interfases gráficas con las que interactúa el usuario (clase *LockedCAWICanvas*, por ejemplo que presenta la interfase gráfica de la figura 21; la clase *CAWIProjectForm* que permite acceder a una descripción breve del proyecto; etc.).
- La segunda razón está relacionada con la creación de una jerarquía de comandos para interactuar con las interfases gráficas. Según la implementación estándar de J2ME, un menú está compuesto por "comandos" que son invocados cuando los selecciona el usuario. Ahora bien, estos comandos en realidad no tienen nada que ver con el patrón de diseño *"Command"* [Gamma et al. 94], sino que por el contrario son una implementación bastante rudimentaria. El siguiente segmento de código Java ayuda a ilustrar este tema.

```
1 public class MyMiddlet extends Middlet implements CommandListener {
2
3   public void startApp() {
4     form = new Form("Hello World");
5     String msg = "My second MIDlet!";
6     form.append(msg);
7     form.setCommandListener(cl);
8
9     showAlert = new Command(ALERT_LABEL, Command.SCREEN, 1);
10    form.addCommand(showAlert);
11
12    sayHi = new Command("Say Hi", Command.SCREEN, 1);
13    form.addCommand(sayHi);
14
15    display = Display.getDisplay(this);
16    display.setCurrent(form);
17 }
18
19  public void commandAction(Command c, Displayable d) {
20
21     if (c == showAlert) { alert = new Alert("Button pressed",
22         "The '" + ALERT_LABEL + "' button was pressed", null, AlertType.INFO);
23     }
24  }
25 }
```

El segmento de código de más arriba presenta una aplicación middlet que implementa la interface *CommandListener* (línea 1), la cual establece el protocolo que deberá ser implementado por todo aquel objeto que desee ser notificado de las selecciones del menú de operaciones. En la línea 7 la propia aplicación se registra como "listener" de los eventos del menú. Las líneas 9 y 10 muestras como se crean los comandos y se asignan al menú propiamente dicho (las mismas líneas deben repetirse para cada nueva opción del menú).

En el párrafo anterior se citó que dichos comandos, a pesar de su nombre, distan bastante de ser una implementación del patrón de diseño del mismo nombre. Este hecho es evidente en la línea 21 en donde se deben usar sentencias "if" (en realidad un "case") para determinar que opción del menú se seleccionó[43] (basándose en el título del comando seleccionado). Este hecho hace difícil la extensión de la funcionalidad de la aplicación, además de representar una solución demasiado estática. Con el fin de tener una implementación más robusta y flexible, se decidió la implementación de una verdadera capa de comandos (siguiendo los lineamientos del patrón de diseño).

La clase *CawiGUICommand* es el tope de la jerarquía de tales comandos, los cuales utilizan la técnica de "double-dispatching" para interactuar con la interfase gráfica y la aplicación para realizar la operación seleccionada. De esta manera cada nueva opción de menú estará representada por una subclase de *CawiGUICommand*.

Al utilizar esta solución de diseño, la implementación del middlet queda como sigue:

[43] La determinación de qué comando se seleccionó se lleva a cabo dentro del método "commandAction", el cual está definido en la interface CommandListener.

```
1 public class MyMiddlet extends Middlet implements CommandListener {
2
3   public void startApp() {
4     form = new Form("Hello World");
5     String msg = "My second MIDlet!";
6     form.append(msg);
7     form.setCommandListener(cl);
8
9     showAlert = new AlertCommand(ALERT_LABEL, Command.SCREEN, 1);
10       //la clase AletCommand es una subclase de CawiGUICommand
11    form.addCommand(showAlert);
12
13    display = Display.getDisplay(this);
14    display.setCurrent(form);
15 }
16
17  public void commandAction(Command c, Displayable d) {
18
19     CawiGUICommand comm.=(CawiGUICommand) c;
20     comm.execute(this);
21  }
22 }
```

Por último se presenta el diagrama de clases correspondiente a la aplicación en sí misma.

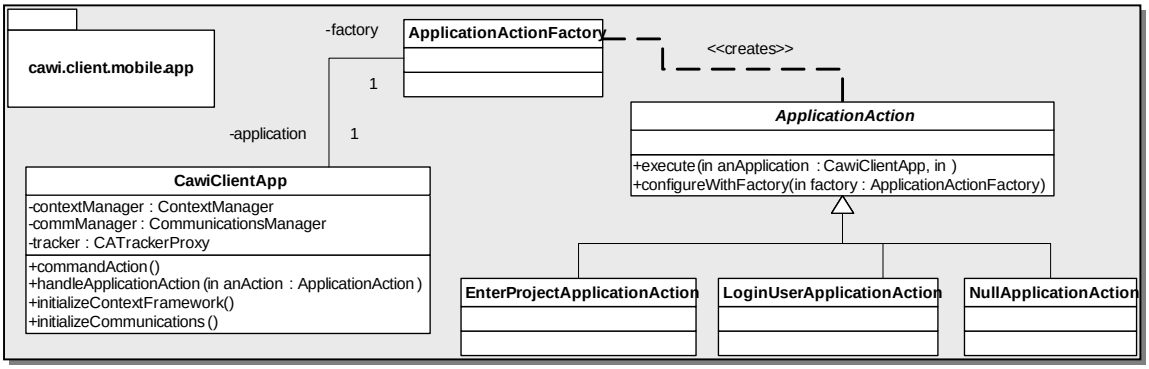

Figura 27: Diagrama de clases de la aplicación móvil.

- Clase *CawiClientApp*: esta clase representa la aplicación móvil en sí misma. Es la encargada de administrar todos los recursos disponibles, tales como comunicaciones e interfases gráficas. Notar que la única instancia de esta clase colabora con el objeto raíz del modelo local, así como con el administrador de contextos (los cuales fueron implementados tal como se presentó en el capítulo IV).

- Clase *ApplicationActionFactory*: esta clase es una implementación del patrón de diseño *"AbstractFactory"* [Gamma et al. 94] y es utilizada para crear las acciones de la aplicación.

- Clase *ApplicationAction*: esta clase abstracta es el tope de la jerarquía de "acciones", los cuales representan los comandos (patrón de diseño *"Command"* [Gamma et al. 94]) que son ejecutados sobre el modelo del cliente a partir del arribo de una notificación proveniente del servidor CAWI. Estos comandos son automáticamente creados por la capa de comunicación, mediante una colaboración entre la clase *"Message"* y la clase *"ApplicationActionFactory"*.

- Clase *NullApplicationAction*: esta clase en particular representa la acción nula, es decir una acción que al ser ejecutada no altera en nada al sistema. Esta solución permite a la capa de comunicación tratar en forma indistinta los mensajes enviados al servidor que producen una respuesta como aquellos mensajes que no producen respuesta alguna. Es una implementación del patrón de diseño *"Null Object"*[44] [Wolf 96].

5.6 Conclusiones

Como se destacó en las secciones anteriores, la funcionalidad provista actualmente por el prototipo desarrollado es realmente muy básica en el sentido que solamente se provee funcionalidad de ingreso al sistema en forma automática, listado de proyectos en los que participa el usuario detectado y

[44] El patrón de diseño Null Object establece la creación de una clase que represente el concepto de nulo para un dominio determinado, con el fin de independizar a los clientes de la presencia de valores nulos.

detalle de un requerimiento asignado. Sin embargo, a pesar del pequeño conjunto de funcionalidad provisto, el desarrollo del prototipo arroja ya algunas conclusiones interesantes:

- En primera medida, el ambiente de desarrollo de aplicaciones móviles es todavía primitivo, en relación a las tecnologías que se hallan en otras plataformas de desarrollo.

- A través de este prototipo se demostró que la orientación a objetos es perfectamente aplicable como paradigma de programación de este tipo de aplicaciones.

- Lamentablemente, a pesar de lo señalado en el punto anterior, no es amplia la adopción del paradigma orientado a objetos para implementar aplicaciones móviles.

- El prototipo demuestra además la factibilidad de la arquitectura distribuida presentada en los capítulos anteriores así como la del framework para contextos.

Capítulo VI: Beneficios de la integración de context-awareness a la Ingeniería de Software

La administración de proyectos es una compleja tarea. Existen tantas interacciones e información para ser analizadas que el líder del proyecto se ve abrumado y confundido, lo cual usualmente conduce a la pérdida de los hitos importantes del proyecto. En este capítulo se discuten las mejoras que podrían obtenerse al integrar información contextual a las diversas tareas de la administración de proyectos.

6.1 Introducción

Es habitual encontrar en la bibliografía relativa a "context- awareness" referencias a experiencias en las cuales se introdujeron fuentes de información contextual para aumentar o mejorar la experiencia de los usuarios en ramas tan disímiles (a primera vista) como guías electrónicas de museos o aplicaciones hospitalarias. Sin embargo, el panorama es absolutamente distinto cuando se trata de experiencias relacionadas con la aplicación de las ideas de "sensibilidad al contexto" para aplicaciones o herramientas utilizadas en las diferentes tareas relacionadas con Ingeniería de Software, en especial la administración de proyectos y de requerimientos.

No es casual la selección del dominio relacionado con la Ingeniería de Software como base para el desarrollo del prototipo presentado en el capítulo anterior. Dicha selección tiene varios fundamentos:

- Por un lado el deseo de expandir el área de aplicación de la tecnología de context-awareness a otras áreas de aplicación todavía no estudiadas.

- La necesidad de mejorar aún más el estado actual de las prácticas comunes de administración de proyectos, con el fin de tornarlas más fáciles de aplicar y con mayor soporte por parte de las herramientas.

- El dominio de la ingeniería de software es lo suficientemente complejo como para ser considerado un desafío a nivel de diseño, probablemente mucho más allá del que puede ser presentado por las aplicaciones usuales como las de un museo o guía de turismo.

6.2 Áreas de aplicación

Existen dentro del área de la Ingeniería de Software muchas oportunidades de aplicación de la "sensibilidad al contexto". Es claro que no todas las actividades obtendrán el mismo beneficio, ya que por ejemplo resulta (por lo menos en primera instancia) difícil ver las reales mejoras que se podrían obtener al realizar análisis post-mortem "sensible al contexto". En el lado "opuesto" tenemos por ejemplo la elicitación de requerimientos, que puede beneficiarse enormemente con la integración de información acerca del contexto, convirtiendo a éste en una nueva fuente de conocimiento del dominio. A continuación se detallan algunas áreas en las cuales se podría obtener un beneficio real a partir de la introducción de información de carácter contextual:

- Administración de proyectos: los líderes de proyecto frecuentemente se encuentran abrumados por la cantidad de información que deben manejar en forma continua para poder monitorear el rendimiento del equipo de desarrollo y eventualmente corregir el rumbo con el fin de cumplir con los hitos establecidos en el plan de proyecto. Resulta evidente entonces que cuanto de mayor nivel sea la información que reciba el líder, mayor provecho podrá extraer de la misma. Es por esta razón que se plantea la utilización de la "sensibilidad al contexto" para establecer ciertos tipos de filtros, como por ejemplo los utilizados en el prototipo, que solamente presentan la información relacionada con los proyectos que se desarrollan en la misma ubicación física que el usuario (en este caso una habitación). Bien podría extenderse el prototipo para incorporar la noción temporal para presentar además aquellos requerimientos que se vencerán en un determinado período, de modo de mantener en vista los principales hitos del proyecto. En este punto se han incorporado nociones tales como identificación del usuario (para determinar las tareas que puede ejecutar el mismo), ubicación del usuario, tiempo "actual" y ubicación de los proyectos de desarrollo.

- Elicitación de requerimientos: una de las principales tareas de todo proyecto corresponde a la "generación" o "extracción" de conocimiento acerca del dominio del problema. Podría utilizarse la noción de sensibilidad al contexto en este caso a fin de poder armar calendarios de reuniones teniendo en cuenta las agendas definidas por los conocedores del dominio (que se encuentren físicamente en las cercanías) a fin de minimizar el tiempo requerido por esta etapa. Aquí se ha considerado información

contextual acerca del tiempo, ubicación del usuario y las personas conocedoras del dominio principalmente.

- Validación de requerimientos: una vez que se han especificado los requerimientos, es necesario proceder a validarlos con las personas que son conocedoras del dominio del problema. Una vez más considerando su posición actual y su agenda es posible organizar de manera eficiente reuniones de validación. Incluso se podría incorporar información proveniente del plan de proyecto para validar primero aquellos requerimientos considerados de alto riesgo. En este punto se han considerado la ubicación de las personas, tiempo y agenda (además del cronograma) como fuente de información de carácter contextual.

- Administración de requerimientos: el correcto monitoreo y control de la línea base de requerimientos que deben ser entregados en cada etapa o iteración de un proyecto asegura una mayor posibilidad de éxito final. Al tomar información contextual del plan de proyecto, así como de la ubicación de los "stakeholders"[45] (con el fin de evacuar posibles dudas sobre cuestiones ambiguas) es posible ordenarlos teniendo en cuenta su prioridad de resolución. Nuevamente en este caso se ha tomado información contextual que contiene posiciones de personas y proyectos, así como información de otro nivel como es la correspondiente al plan del proyecto.

- Inspecciones de diversos artefactos (diseño, código): la tarea de inspección de ciertos artefactos generalmente no se realiza o se realiza de manera aislada entre el "inspector" y el miembro del

[45] Stakeholder: es toda persona o institución que tiene interés en el éxito de un proyecto de software.

equipo responsable del artefacto bajo inspección. Al incorporar en una herramienta la noción de posición del responsable del artefacto es posible presentar solamente aquellos artefactos que pueden ser inspeccionados en conjunto por el inspector y el responsable. Para poder brindar esta funcionalidad extra, es preciso incorporar nociones tales como ubicación de los usuarios y proyectos, criticidad de los artefactos a ser inspeccionados (tomando el plan de proyecto como fuente de tal conocimiento) y el perfil de cada uno de los usuarios (para presentar solamente aquellos artefactos que el inspector está en condiciones de evaluar).

- Testing: este caso es similar al anterior, en el sentido de que se pueden filtrar los artefactos que deben ser probados teniendo en cuenta no solo la ubicación del "tester" (ya que no tiene sentido presentar artefactos de otros proyectos), sino que también es necesario considerar el plan de proyecto y el cronograma del mismo a fin de establecer prioridades. Eventualmente incluso se podría considerar la posición física de la persona que desarrolló el artefacto que se está probando.

6.3 Conclusiones

En la sección anterior se presentaron algunas áreas de aplicación del concepto de "sensibilidad al contexto", en las cuales es notoria la mejora en la funcionalidad que puede ser provista al usuario final. La investigación en el área de la ingeniería de software y la aplicabilidad de "context-awareness" recién comienza; debiéndose extenderse con el correr del tiempo con el objetivo

de que se abran nuevas áreas de aplicabilidad ni siquiera consideradas en este trabajo.

Como se hizo evidente en la sección anterior, no sólo la ubicación de los elementos cobra importancia, sino también el concepto temporal, ya que en la mayoría de los ejemplos citados, fue necesario tomar información relacionada con el plan de proyecto y cronograma a fin de filtrar u ordenar los ítems más críticos. Otro de los elementos importantes además de la ubicación y tiempo es el que está relacionado con el perfil del usuario que está ejecutando la aplicación, ya que a partir de dicha información es posible personalizar la aplicación para que le permita al usuario ejecutar solamente lo que éste tiene autorizado.

Nuevamente, a modo de cierre de esta breve discusión, la aplicación de "context awareness" a la Ingeniería de Software recién comienza y es de esperarse una gran evolución y aceptación.

Capítulo VII: Trabajo futuro

A lo largo de los capítulos precedentes, se han ido planteando las características principales de la arquitectura y de una aplicación prototípica. Dicha arquitectura de ninguna manera puede y debe ser considerada como completa. Sin embargo, es suficientemente flexible para acomodar las extensiones necesarias para dotarla de las nuevas características que sean necesarias.

7.1 Introducción

En las siguientes secciones se comentarán algunos de los principales puntos que deben ser considerados al construir una aplicación real utilizando como base la arquitectura propuesta en este documento. Los puntos abarcan una gran variedad de temas, que van desde lo social hasta lo puramente tecnológico.

7.2 Violación de la privacidad

En la medida que los nuevos dispositivos continúen mejorando las posibilidades de la computación "sensible al contexto", y las ideas propuestas por dicho paradigma sean aceptadas en forma generalizada, nos veremos enfrentados a un tipo de problema que no es considerado habitualmente en el dominio de la informática. Nadie razonablemente puede negar las ventajas que podrían obtenerse al contar con las cualidades de movilidad, sensibilidad al contexto, personalización y otras muchas características, sin embargo no está claro cuál es el precio a pagar por estas nuevas ventajas. Por ejemplo, cuantas de las empresas que actualmente están instalando dispositivos para el cobro

automático del peaje en las rutas nacionales han advertido fehacientemente a sus usuarios que a partir de la instalación del dispositivo en el vehículo, la empresa cuenta con la formidable posibilidad de detectar los movimientos de las personas (si se encuentra en la ciudad, hábitos de comportamiento durante los viajes, etc.). Esto no es nuevo, desde el nacimiento de las tarjetas de créditos, las empresas emisoras han contado con valiosa información de sus clientes, sin siquiera advertir de tales prácticas a los mismos.

Es necesario establecer cuáles son los límites legales y éticos al momento de implementar soluciones "sensibles" al contexto, de modo de poder salvaguardar la privacidad de las personas. En [Stone 03] se cita un ejemplo que ayuda a aclarar este punto. En EE.UU. existe una empresa llamada VeriChip [Verichip 05] cuya intención es implantar un chip en cada persona con el fin de identificarla unívocamente. Si bien son claros los beneficios potenciales que podrían obtenerse a partir de la implantación, por ejemplo se terminarían los casos de robo de identidad, no son para nada claro cuales son problemas que podrían generarse. Por ejemplo, quién sería la autoridad encargada de decidir quien usa o no el chip?; de ser establecido que toda persona debería contar con un chip, cómo podría garantizarse que una persona (por ejemplos los delincuentes buscados por la Justicia) no se lo retiren en forma ilegal?. Estos son solamente algunos de los interrogantes que surgen cuando se consideran los aspectos de la privacidad de las personas en un ambiente en donde, paradójicamente, la identidad de las personas es lo principal.

7.3 Pruebas de campo "reales"

Durante la construcción y posteriores pruebas, el prototipo se comportó satisfactoriamente. Sin embargo, dichas pruebas solamente comprenden un subconjunto mínimo de las situaciones reales a las que se vería expuesta una aplicación en "producción". Es necesario contar con mayores equipos, y

recursos para poder organizar pruebas que demuestran claramente las verdaderas cualidades y debilidades de la arquitectura propuesta. Por ejemplo, no es posible simular en forma aproximada las condiciones que se dan al pasar de un proveedor de servicios a otro en forma continua. Tampoco es posible simular las cambiantes condiciones de la red sobre la cual se transmiten los datos de las aplicaciones. Incluso algo mucho más simple, como la carga que deben soportar los servidores cuando deben atender a múltiples clientes.

Todas las situaciones nombradas más arriba sólo pretenden demostrar la necesidad de mayores pruebas, en situaciones no simuladas, para evaluar la utilidad de la presente arquitectura.

7.4 Interoperabilidad

Al momento de brindar un servicio, cuanta más agregada (de mayor nivel) pueda estar la información, más útil será la prestación del servicio, y en consecuencia más satisfactoria será la experiencia del usuario. Incluso desde el punto de vista comercial puede notarse una ventaja respecto de servicios que manejan información en formato más "crudo". Ahora bien, ¿como se llega a tal nivel de agregación de la información? A priori podría pensarse básicamente en dos posibles esquemas: por un lado contar uno mismo (como proveedor de servicios) de suficientes fuentes distintas de información para luego agregarla. La segunda alternativa consiste en poder acceder a fuentes "externas" de información. Cada una de las alternativas tiene sus beneficios y sus desventajas. La primera alternativa tiene como clara ventaja la velocidad al momento de agregar la información y el hecho de que uno es el propietario absoluto de los datos. Sin embargo, esta solución es impracticable en la realidad por diferentes motivos. Entre dichos motivos se pueden citar el tiempo necesario para una recopilación de una cantidad suficiente de información (los sistemas de GIS hace décadas que almacenan información); la inmensa cantidad de recursos

físicos necesarios para almacenar y mantener la información; la relativa poca utilidad frente a un gasto enorme; etc. De lo dicho anteriormente se desprende que la opción más factible es contar con información provista por diferentes proveedores. Sin embargo, esta opción no está libre de problemas tampoco. Mayormente se dan problemas que tienen que ver con la continuidad de los datos que se están utilizando. Una implementación completa debería atender a los problemas relacionados con la continuidad semántica, a la continuidad topológica y a la continuidad geográfica de modo de no depender de los bordes establecidos por los diferentes proveedores para prestar servicios de calidad (ver [Bazzocco et al. 03]).

7.5 Robustez

A pesar del avance en la fiabilidad alcanzada hoy en día en Internet, todavía nos encontramos lejos de la calidad necesaria para hacer que un sistema distribuido sobre Internet sea considerado "robusto". El problema de la robustez se potencia al considerar que la operatoria habitual de este tipo de servicios se expande a múltiples proveedores diferentes, los cuales en más de una oportunidad deben continuar a pesar de la caída de la red o la desconexión momentánea del usuario móvil. Todo esto hace indispensable el planteo de un esquema que soporte transacciones móviles con el fin de hacer lo más transparente posible dichos eventos al usuario móvil. Incluso se debería establecer un esquema estándar al cual se deberían adherir los proveedores para continuar aquellas transacciones que hayan sido iniciadas por los usuarios en otros proveedores, y que por el movimiento de los usuarios, ahora tengan que ser terminadas bajo la supervisión de un proveedor distinto.

7.6 Performance y optimización

Los temas de la performance total del sistema y la optimización de los recursos necesarios para la ejecución de los servicios ciertamente merecen una atención mayor a la prestada aquí.

Existen múltiples maneras de incrementar la performance del sistema sin que esto necesariamente signifique la compra de equipos de hardware más potentes. Por ejemplo, en el prototipo presentado anteriormente, toda la información que viaja a través de la red es comprimida previamente. La compresión, al aplicarse a mensajes compuestos en su gran mayoría de texto, logra radios de compresión de alrededor del 80%. En [Virrantaus et al. 02] puede encontrarse otro excelente ejemplo de como una solución basada exclusivamente en software puede incrementar la performance del sistema. En dicho trabajo, los autores manejan diferentes niveles de detalle de la información, los cuales se van incrementando a medida que el usuario necesita de información detallada sobre una zona en particular. No tiene sentido enviar la información con el máximo nivel de detalle, ya que esto no solo implica una caída del ancho de banda disponible, sino que también implica largos tiempos de conexión (y en consecuencia aumenta directamente la probabilidad de una caída de la transferencia de información). Además la probabilidad de que se den cambios en la información es mucho más alta a nivel detallado (por ejemplo el estado de una calle) que a nivel de poco detalle (una provincia o ciudad), con lo cual al postergar lo máximo posible el envío de información detallada también en forma indirecta se está protegiendo al usuario de notificaciones de cambios sucedidos en la información que éste haya consultado recientemente.

En conclusión, esquemas como los citados anteriormente deben ser diseñados para lograr una mayor performance, sin que esto signifique mayores inversiones.

Capítulo VIII: Conclusiones

En cada uno de los capítulos precedentes se han introducido conceptos e ideas relacionadas con la aplicación de "context-awareness" a las aplicaciones móviles. En este capítulo se presentarán las conclusiones generales que pueden extraerse a fin de resumir el aporte de esta tesis.

El ambiente de desarrollo y ejecución de las aplicaciones móviles basadas en J2ME [J2ME 05] es todavía extremadamente restrictivo en cuanto a recursos disponibles. Quizás por esta razón es común encontrar en la bibliografía relacionada una tendencia a considerar como bueno un esquema en el cual se minimizan al extremo la cantidad de clases participantes en una aplicación; aún si esto lleva a situaciones como las presentadas en el capítulo V en el cual el manejo de los comandos de menú se realiza mediante un gran "case" en el cual se evalúa el título de la opción seleccionada. Está claro que esta solución resulta extremadamente estática y no tolera cambios tal como lo hace la solución propuesta en el mismo capítulo (basada en la técnica de "double – dispatching" y el patrón de diseño *"Command"* [Gamma et al. 94]). La solución basada en diseño orientado a objetos no necesariamente representa una solución costosa en términos de recursos necesarios. Durante la implementación del prototipo se halló que el tamaño promedio (medido en bytes) de una clase relacionada con un comando era de aproximadamente 800 bytes[46].

Otro importante aporte de este trabajo tiene que ver con la apertura de nuevos dominios en los cuales investigar la aplicabilidad de la sensibilidad al

[46] Como medida de comparación, un teléfono reciente como el Motorota V180 [Motorota 05] ya tiene 2 mb de espacio para aplicaciones Java (1 mb= 1048576 bytes).

contexto (más allá de los clásicos ejemplos relacionados con paseos turísticos en una ciudad o museo). En particular se presentó un trabajo enfocado en el dominio de la Ingeniería de Software (tanto en el capítulo dedicado a la implementación de un prototipo funcional como en el capítulo relativo a la integración de la sensibilidad al contexto a las herramientas de Ingeniería de Software). Los ejemplos de mejora citados en el capítulo anterior muestran claramente las ventajas que se podrían obtener en cuanto a funcionalidad y amigabilidad al incorporar información contextual. Es clara también la necesidad de investigar con mayor dedicación y esfuerzo esta nueva área de aplicación a fin de identificar los desafíos de diseño.

En los primeros capítulos se introdujo una propuesta basada en el paradigma orientado a objetos para desarrollar un framework de soporte para las aplicaciones sensibles al contexto. Como se evidenció en tales capítulos, el diseño planteado dista de ser complejo y extenso, razón por la cual no se perciben razones que disminuyan su factibilidad. Está claro que deben ahondarse los detalles "finos" que sólo se conocerán al momento de integrar dispositivos reales para la captura de la información conceptual, sin embargo es de esperarse poco impacto en el resto del framework gracias a la aplicación de técnicas de diseño que conducen a sistemas flexibles y poco acoplados.

Un punto importante que ha sido dejado de lado concientemente en este trabajo tiene que ver con la necesidad de estudiar las formas posibles en las que se pueda incorporar toda esta nueva tecnología de captura de la información sin invadir la privacidad de los usuarios. Cuantas más fuentes de información contextual se puedan incorporar, más alto será el nivel de abstracción que se podrá alcanzar. Debido a esto último, es evidente la búsqueda de mecanismos que permitan la incorporación de muchas fuentes de información. Por otro lado, como se planteó en los capítulos iniciales, al aumentar las fuentes de información, inevitablemente también irá aumentando la complejidad del

framework y de las aplicaciones que hacen uso de éste. En consecuencia, el framework deberá ganar en "inteligencia", para poder incorporar estas nuevas fuentes de información de manera implícita o transparente para el usuario final (esto es, el usuario debería tener el mínimo necesario de conocimiento de los aspectos técnicos relacionados con la incorporación de nuevas fuentes de información contextual). Sin embargo, cuanto más "automática" sea la incorporación de nuevas fuentes, más se estará invadiendo la privacidad del usuario, ya que el framework por defecto estará asumiendo que el usuario desea compartir la información obtenida a través de la nueva fuentes de información conceptual. Como ejemplo, no se ha tomado en cuenta en este trabajo que pasaría si un usuario no está dispuesto a proveer su ubicación o preferencias de usuario a fin de personalizar una aplicación determinada.

Para finalizar, en este trabajo se ha tomando un pequeño subconjunto de las aplicaciones móviles sensibles al contexto, en particular aquellas que están soportadas por clientes "pesados", es decir aquellas que pueden almacenar cierta información y con ciertos recursos de procesamiento local. El framework para aplicaciones sensibles al contexto y la arquitectura deberían ser extendidas para soportar otros tipos de aplicaciones, por ejemplo aquellas que son implementadas con clientes livianos o aquellas que requieren que el servidor mantenga información contextual en el mismo servidor, a fin de iniciar por sí mismo una "conversación" con el cliente cuando dicho servidor detecte un cambio en algunos de los contextos en los cuales el cliente está interesado.

En definitiva, a lo largo de este trabajo se ha demostrado que la orientación a objetos y las técnicas de diseño como los patrones, son herramientas perfectamente aplicables a los problemas hallados en el ambiente de las aplicaciones móviles y sensibles al contexto.

Bibliografía

[Bazzocco et al. 03] Silvia Gordillo, Javier Bazzocco, Gustavo Rossi & Robert Laurini
Engineering Location-Based Services in the Web
Web Engineering Book - 2003 (a ser publicado).

[Brown et al. 97] Brown, P.J.; Bovey, J.D.; Chen, X.
Context-Aware Applications: From the laboratory to the marketplace.
IEEE Personal Communications, 4(5) (1997) 58-64.

[Cugola et al. 02] Cugola, Gianpaolo; Jacobsen, H.Arno
Using Publish/Subscribe Middleware for mobile systems
ACM SIGMOBILE Mobile Computing and Communications Review
Volume 6, Issue 4 (October 2002)
Pages: 25 - 33

[Dey 98] Dey, A.K.
Context-Aware Computing: The CyberDesk Project.
AAAI 1998 Spring Symposium on Intelligent Environment, Technical Report SS-998-02 (1998) 51-54

[Dey 00] Anind K. Dey
Providing Architectural Support for Building Context-Aware Applications
PhD thesis, College of Computing, Georgia Institute of Technology, December 2000.

[Double dispatching 01] Sitio Web:
http://wiki.cs.uiuc.edu/VisualWorks/Double+Dispatching
31/12/2001.

[Gamma et al. 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Design Patterns Elements of Reusable Object-Oriented Software
Addison Wesley Professional Computing Series
October 1994
ISBN 0-201-63361-2.

[Gordillo et al. 97] S.Gordillo, F.Balaguer, F. Das Neves
Generating the Architecture of GIS Applications with Design Patterns
5th ACM Workshop on Geographic Information Systems.

Las Vegas, Nevada, USA. November 1997.

[Hibernate 02] Hibernate – Relational Persistence for Idiomatic Java
Sitio Web: http://www.hibernate.org

[HTTP 96] HyperText Transfer Protocol
Sitio Web: http://www.w3.org/Protocols/
Copyright © 1996-2003 W3C®

[IPAQ 05] iPAQ Pocket PC
Hewlett Packard
Sitio Web:
http://welcome.hp.com/country/us/en/prodserv/handheld.html

[J2ME 05] Java 2 Micro Edition
Sitio Web: http://java.sun.com/j2me
Copyright © 1994-2005 Sun Microsystems, Inc.

[Java 05] Lenguaje Java
Sitio Web: http://java.sun.com
Copyright © 1994-2005 Sun Microsystems, Inc.

[JIRA 05] Herramienta para el seguimiento de bugs.
Sitio Web: http://www.atlassian.com/software/jira/

[Java-M 05] Java Mobile Devices
Sitio Web: http://www.javamobiles.com

[JCP 05] Java Community Process
Sitio Web: http://jcp.org

[JSR 256 04] Java Specification Request
JSR 256 Mobile Sensor API
Solicitado por Nokia Coporation en el año 2004

[Loomis 95] M. Loomis
Object Databases: The Essentials
Addison-Wesley, 1995

[Mantis 04] Herramienta para el seguimiento de bugs de tipo open-source.
Sitio Web: http://www.mantisbt.org/

[Micro 05] MicroJava devices
Sitio Web: http://www.microjava.com/devices

[Moore 00] Ley de Moore
Versión electrónica disponible en
http://www.baquia.com/com/legacy/14184.html
Creada el 15/06/2000

[Motorota 05] Motorola V180 Teléfono móvil con capacidades Java
Sitio Web:
http://www.phonescoop.com/phones/phone.php?p=466

[MySQL 05] Base de datos relacional open-source
Sitio Web: http://www.mysql.com

[Palm 05] Palm, Inc.
Sitio Web: http://www.palm.com

[Ryan et al. 97] Ryan, N., Pascoe, J., Morse, D.
Enhanced Reality Fieldwork: the Context-Aware Archaeological Assistant.
Gaffney,V., van Leusen, M., Exxon, S. (eds.) Computer Applications in Archaeology
(1997)

[Schilit. B et al. 94] Schilit, B., Theimer, M.
Disseminating Active Map Information to Mobile Hosts.
IEEE Network, 8(5) (1994) pág. 22-32.

[Servlets 05] Tecnología perteneciente al estándar J2EE.
Sitio web: http://java.sun.com/products/servlet/index.jsp

[SGML 95] http://www.w3.org/MarkUp/SGML/

[Stone 03] Adam Stone
The Dark Side of Pervasive Computing
IEEE Pervasive Computing – 2003
1536-1268/03

[Struts 05] Implementación del MVC2 para Web basada en Jsp y Servlets.
Es un framework open-source.
Sitio Web: http://jakarta.apache.org/struts

[Swing 05] Tecnología para la construcción de interfases gráficas en Java
Sitio Web: http://java.sun.com/products/swing

[Tomcat 03] Contenedor de Servlets & JSP
Sitio Web: http://jakarta.apache.org
Copyright © 1999-2003, Apache Software Foundation

[UWA 02] Ubiquitous Web Applications
Sitio Web : http://www.uwaproject.org/
UWA Consortium Project IST 2000-25131
Piazza del Carmine 22, 09100 Cagliari, Italy

[VeriChip 05] Verichip Corporation
http://www.verichipcorp.com/

[Virrantaus et al. 02] Kirsi Virrantaus, Jouni Markkula, Artem Garmash, Vagan Terziyan, Jari Veijalainen, Artem Katanosov, Henri Tirri
Developing GIS - Supported Location - Based Services
Proceedings of the Sencond International Conference on Web Information Systems Wngineering (WISE'02)
0-7695-1393-X/02
2002 – IEEE COMPUTER SOCIETY

[Wireless 05] Wireless Java
Sitio Web: http://wireless.java.sun.com/device/

[Woolf 96] Bobby Woolf
The Null Object Pattern
1996
Sitio Web: http://citeseer.ist.psu.edu/woolf96null.html

[XML 03] e**X**tensible **M**arkup **L**anguage
Sitio Web: http://www.w3.org/XML/
Copyright © 1996-2003 W3C

Anexo I: Patrones de diseño

Los patrones de diseño aportan soluciones conocidas, probadas y robustas a problemas recurrentes en el diseño de software. Esta sección presenta una breve descripción de la intención de cada uno de los patrones de diseño que se ha utilizado en este trabajo.

Abstract Factory

Provee una interfase para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.

Composite

Compone objetos en estructuras de árboles para representar jerarquías "contenedor-parte". Este patrón permite a los clientes tratar a los objetos individuales y a las composiciones de objetos en forma uniforme.

Command

Encapsula un requerimiento como un objeto, por lo que permite parametrizar a los clientes con varios requerimientos, encolar o almacenar estos requerimientos y dar soporte a operaciones que se pueden volver atrás.

Decorator

Agrega responsabilidades adicionales a un objeto en forma dinámica. Los decoradores proveen una alternativa flexible a la subclasificación como forma de extender la funcionalidad.

Facade

Provee una interfase unificada a un conjunto de interfases de un subsistema. Este patrón permite definir una interfase de más alto nivel que hace más fácil la utilización de un subsistema.

Mediator

Define un objeto que encapsula como interactúan un conjunto de objetos. Este patrón promueve el bajo acoplamiento al evitar que los objetos se refieran entre sí explícitamente, y permite variar la interacción en forma independiente.

Observer

Define una dependencia uno a muchos entre objetos de manera que cuando un objeto cambia, todos sus dependientes son notificados y actualizados automáticamente.

Proxy

Provee un representante de otro objeto para controlar el acceso a éste.

Singleton

Asegura que una clase tenga una sola instancia, y provee un punto global de acceso a ésta.

Strategy

Define una familia de algoritmos, los encapsula y los hace intercambiables entré sí. Este patrón permite a los algoritmos variar independientemente de los clientes que los utilizan.

Template Method

Define un esqueleto de un algoritmo en una operación, delegando algunos pasos a las subclases. Este patrón permite a las subclases redefinir algunos pasos sin cambiar la estructura del algoritmo.

Anexo II: Tecnologías del Prototipo desarrollado

En esta sección se presentan brevemente algunos detalles de carácter técnico relacionados con las diferentes tecnologías que han sido utilizadas en la construcción del prototipo, con el fin de transmitir el por qué de su selección, así como también los problemas encontrados a partir de su utilización.

Http (HyperText Transfer Protocol)

Protocolo de comunicación sobre el cual se basa Internet para la publicación de páginas. Por defecto corre sobre el puerto 80, lo cual lo hace especialmente apto como mecanismo de transmisión de información ya que no se ve bloqueado por elementos de protección como firewalls[47]. Permite el envío y recepción de grandes cantidades de información (dependiendo del método utilizado se pueden enviar hasta 2gb[48] de información). Sobre este protocolo se implementó el envío y recepción de datos de la arquitectura, utilizando datos en forma de mensajes XML comprimidos para optimizar la performance.

Para más información ver [HTTP 96].

Java

Lenguaje orientado a objetos. Es especialmente apto para el desarrollo de aplicaciones distribuidas, debido a que soporta de manera natural los protocolos en los cuales se basa Internet. Una de sus características principales

[47] Firewall: software de protección de computadoras que actúa mediante el bloqueo de todos los puertos de acceso remoto. Se pueden liberar en forma controlada algunos de los puertos (por ej. el puerto 80) para que agentes externos puedan acceder a los servicios que se ejecutan en dichos puertos.
[48] 1gb (gigabyte) corresponde a 1024 megabytes, es decir a 1024 x 1024 x 1024 bytes = 1073741824 bytes.

es que la ejecución de programas escritos en este lenguaje se basa en la existencia de una máquina virtual, la cual traduce los "bytecodes" genéricos generados por el compilador a un conjunto de instrucciones de máquina apropiados. Dicha máquina virtual le permite al programador desarrollar una única solución, que luego se puede ejecutar en varias plataformas (sistemas operativos) distintos (es por esto que se dice que Java es "multiplataforma").

El lenguaje Java viene en 3 "sabores":

- J2SE (Java 2 Standard Edition): contiene las clases indispensables para desarrollar una aplicación utilizando Java. Constituye el núcleo del lenguaje. Se puede desarrollar aplicaciones "desktop" y así como aplicaciones distribuidas primitivas. En el prototipo se utilizó J2SE.
- J2ME (Java 2 Micro Edition): edición particularmente desarrollada para soportar dispositivos móviles como teléfonos celulares y palms. Se utilizó J2ME en el desarrollo del prototipo de un cliente de la arquitectura propuesta.
- J2EE (Java 2 Enterprise Edition): versión que comprende más de 11 tecnologías. Particularmente apto para el desarrollo de aplicaciones de gran envergadura y robustez. En el prototipo se utilizó para el desarrollo del servidor.

Para más detalles ver [Java 05].

Hibernate

Es una herramienta poderosa, de alta performance para persistencia objeto / relacional para Java. Hibernate permite desarrollar clases Java de manera común (incluyendo asociaciones, herencia, polimorfismo, composición, así como el framework de colecciones de Java). Posee además un lenguaje de consulta, denominado HQL (Hibernate Query Language), el cual es una

extensión con un agregado mínimo de objetos al lenguaje estándar para consultas SQL.

Hoy en día es la herramienta de mapeo objeto / relacional más popular del mercado.

Para más información ver [Hibernate 02].

Jakarta Tomcat

Implementación de referencia del "contenedor" de páginas JSP y Servlets. Es un producto desarrollo como proyecto open-source por Apache. Constituye un servidor, capaz de servir requerimientos de servlets (pequeños programas Java que corren exclusivamente en el servido, sin interfase gráfica) y de páginas con contenido dinámico. Se utilizó en el desarrollo del prototipo como sustento del servidor capaz de recibir requerimientos sobre http, los cuales eran atendidos por este servidor en primera instancia y luego enviados a las aplicaciones correspondientes.

Para más información ver [Tomcat 03].

MySQL

El servidor de bases de datos MySql es la base de datos open-source más popular del mundo (existen más de 5 millones de instalaciones que usan MySQL para dar soporte a sitios Web de alto tránsito así como a otros sistemas de negocios críticos como Associated Press, Google, Nasa, Sabre y Suzuki. Para más información al respecto ver [MySql 05].

Xml (eXtensible Markup Language)

Es un formato de texto simple y flexible derivado de SGML. Originalmente fue diseñado para soportar los desafíos de la publicación electrónica. Hoy en día juega un papel preponderante en el intercambio de información en la Web.

Se dice que es extensible ya que no es un formato fijo como HTML. XML es en realidad un "meta-lenguaje" (un lenguaje para describir lenguajes), el cual permite el diseño de lenguajes de marcas personalizados.

XML es un proyecto de World Wide Web Consortium (W3C), siendo un formato público, es decir que no es propiedad de ninguna empresa en particular.

Para más información ver [XML 03].

Servlets

La tecnología de Servlets Java provee un mecanismo simple y consistente para extender la funcionalidad de un servidor WEB y para acceder a sistemas de negocios existentes. Un servlet puede ser considerado como un applet que corre en el lado del servidor sin una interfase gráfica. Para más detalles ver [Servlets 05].