Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Trabajo Especial de Grado

Elaboración de un Gem de Ruby que encapsule un Middleware SMS para el envío y recepción de mensajes de texto

Bachiller Alejandro García Tomé, C.I.: 18.442.868

y

Bachiller Yennifer Astrid Chacón, C.I.: 17.751.356

Tutor
Prof. Sergio Rivas

Caracas, Septiembre de 2009

Acta

Quienes suscriben miembros del Jurado, designado por el Consejo de Escuela de Computación, para examinar el Trabajo Especial de Grado presentado por los **Bachilleres Alejandro García Tomé, C.I. 18.442.868 y Yennifer Astrid Chacón, C.I. 17.751.356**, con el título **Elaboración de un Gem de Ruby que encapsule un Middleware SMS para el envío y recepción de mensajes de texto**, a los fines de optar al título de Licenciado en Computación, dejan constancia lo siguiente:

Leído como fue, dicho trabajo por cada uno de los miembros del Jurado, se fijó el 11 de septiembre de 2009 a las 10:00 a.m., para que sus autores lo defendieran en forma pública, se hizo en el aula 7 de la Facultad de Ciencias de la Universidad Central de Venezuela, mediante una presentación oral de su contenido. Finalizada la defensa pública del Trabajo Especial de Grado, el Jurado decidió aprobarlo con una nota de puntos, en fé de lo cual se levanta la presente Acta, en Caracas a los once días del mes de septiembre del año dos mil nueve, dejándose también constancia de que actuó como Coordinador del Jurado el profesor Sergio Rivas.

Prof. Sergio Rivas
Tutor

Prof. Eugenio Scalise
Jurado

Prof. David Perez
Jurado

Agradecimientos y Dedicatoria

Primeramente quiero agradecer a mi familia, quienes en todo momento estuvieron a mi lado para brindarme su apoyo incondicional. A mi abuela Ceveles, quien siempre me ha dado todo su amor y ha estado pendiente de mi en todo momento. A mis hermanos Neni y TinTin por ser fuente de alegrías y risas cuando lo necesitaba. A Alito quien estuvo siempre a mi lado dandome su amor y apoyo. A mis compañeros y profesores por la ayuda y conocimientos brindados. Finalmente quiero agradecer a la Universidad Central de Venezuela, por haberme dado la oportunidad de culminar mis estudios. Quisiera dedicar este trabajo a mis padres, Yajaira y Robin, quienes han sido parte fundamental de este logro, ya que siempre me han brindado su apoyo, amor y cariño.

Yennifer Astrid Chacón.

Quisiera agradacer antes que nada a toda mi familia por darme su apoyo en todo momento, a la Universidad Central de Venezuela por permitir la culminación de mis estudios profesionales de manera exitosa, a Yennifer por apoyarme y darme su cariño siempre y finalmente a todos los profesores que durante la carrera me aportaron gran cantidad de conocimientos; igualmente a mis amigos por ayudarme siempre que fue necesario. Quisiera dedicar este trabajo a mis papas, Luis y Monica y a mi hermano Rodrigo por estar siempre conmigo brindandome su apoyo y su experiencia, siendo éstos de gran ayuda para la culminación de este trabajo.

Alejandro García Tomé.

Resumen

El presente Trabajo Especial de Grado tiene como objetivo principal la elaboración de un gem[1] de extensión para el lenguaje de programación Ruby. Este gem encapsula un middleware que permite el envío y recepción de mensajes de texto a través de dispositivos celulares conectados al computador. Para la realización de los módulos de envío y recepción de mensajes se utiliza Gnokii[2], logrando la comunicación del computador con el dispositivo celular a través de comandos AT[3]. Dichos comandos están soportados hoy en día por la mayoría de los teléfonos celulares que utilizan la tecnología GSM. Se debe utilizar un mecanismo para poder integrar la librería Gnokii(en lenguaje C/C++) con el lenguaje Ruby, para esto, debe crearse una extensión a partir del código desarrollado en lenguaje C/C++. Existen distintos mecanismos que permiten la creación de una extensión para Ruby. Se puede crear manualmente, utilizando un generador como Swig[4], o utilizar código C/C++ embebido en código Ruby. Al construir un gem de Ruby, éste puede ser distribuido fácilmente colocándolo en repositorios como Github o RubyForge, desde donde puede ser descargado y utilizado por otros usuarios.

Contactos

Alejandro Garcia Tome
email: hewital@gmail.com

Yennifer Astrid Chacon
email: ychacon@gmail.com

Sergio Rivas
Universidad Central de Venezuela
Escuela de Computación
email: sergiorivas@gmail.com

[1] Librería para ser usada desde el lenguaje de programación Ruby

[2] Librería de código abierto escrita en lenguaje C/C++, que provee diversas funcionalidades para el uso de teléfonos celulares a través del computador.

[3] ATention, conjunto de instrucciones que conforman un lenguaje para la comunicación entre un computador y un modem o dispositivo celular.

[4] Herramienta de desarrollo que hace posible conectar programas escritos en lenguaje C/C++ con otros lenguajes de alto nivel

Tabla de Contenidos

Lista de Figuras

Parte I

Introducción y Propuesta

Capítulo 1

Introducción

Hoy en día el volumen de tráfico de mensajes cortos de texto (SMS, Short Messages Service) en las redes móviles ha alcanzado unos niveles muy altos, que muestran como este servicio ha ido creciendo en popularidad y en uso, lo que permite presumir que la mensajería instantánea representará uno de los grandes avances en la telefonía móvil. Es por ésto que muchas organizaciones han comenzado a realizar adaptaciones de esta tecnología en diversas aplicaciones para sacar así provecho de sus numerosas ventajas.

Entre los servicios más comunes y populares desarrollados e implantados por miles de compañías se encuentran los servicios de "delivery" que permiten dar informaciones sobre servicios, regulaciones, eventos, entre otras muchas funcionalidades, a los suscriptores y clientes de dicha compañías, lo que da a éstos una ventaja enorme al poder recibir información "cuando sea" y "donde sea". Otros servicios como los de consulta bajo demanda son también muy populares en lo que respecta a aplicaciones que implementan y adaptan la tecnología SMS.

A lo largo del presente trabajo se exponen diversos puntos relacionados a la tecnología SMS, así como también aspectos relacionados con la integración de distintas herramientas que permitan la implementación de servicios ofrecidos por esta tecnología, pudiendo éstos, ser utilizados por diferentes usuarios de una manera fácil y rápida.

Cualquier tipo de aplicación que considere útil la adaptación del servicio de mensajería instantánea puede hacerlo sin ningún problema siempre y cuando se tomen en consideración todos los mecanismos de integración necesarios con el software correspondiente que ofrece este servicio. Esta integración puede realizarse de manera particular para una plataforma determinada en la que esté corriendo la aplicación de usuario, o puede hacerse de manera general desarrollando un middleware SMS que provea un conjunto de servicios y funcionalidades estándares a través de una Interfaz de Programación de Aplicaciones (API, Aplication Programming Interface). Estos servicios ofrecidos pueden ser utilizados de manera independiente a la plataforma en la que se esté trabajando.

Por otra parte, se propone la utilización de Ruby, ya que es un lenguaje robusto y extensible que ha probado ser bastante exitoso en el desarrollo ágil de aplicaciones.

Así pues, por todos los motivos expuestos, se desarrolló un middleware SMS que permite ser adaptado a cualquier aplicación Ruby, bajo distintas plataformas y ambientes de trabajo. Adicionalmente, se encapsuló dicho middleware en un gem de Ruby.

El presente documento está estructurado de la siguiente manera:

Parte I - Propuesta: Esta parte presenta la problemática planteada y los motivos que llevaron a la construcción de un middleware SMS. Además se explica la solución propuesta para el problema planteado.

Parte II - Marco Conceptual: En esta parte se explican los diversos aspectos teóricos relacionados con los procesos, metodologías y tecnologías adoptadas para la implementación de la solución propuesta.

Parte III - Marco Aplicativo: En esta parte se explica de que manera se adaptaron los procesos y metodologías de desarrollo utilizados durante la realización del sistema. Además se expone el proceso de implementación del sistema, documentando cada una de las iteraciones.

Parte IV - Conclusiones y Recomendaciones: En esta parte se explica de forma general y resumida todas las actividades realizadas durante la implementación del sistema, las dificultades presentadas y los aportes de la investigación. Además se realizan algunas recomendaciones para hacer un mejor uso del sistema.

Capítulo 2

Propuesta

El uso de dispositivos móviles y en particular de teléfonos celulares ha crecido de manera significativa en los últimos años, observándose cada día más personas que disponen teléfonos celulares. Estos dispositivos han ido evolucionando, y aunque en la actualidad ofrecen opciones como juegos, música y aplicaciones de entretenimiento en general, su objetivo fundamental es lograr la comunicación "wireless" de la persona que posee el dispositivo, pudiendo ésta, liberarse de la telefonía que lo ataba a establecer una comunicación desde un sitio fijo. Esta popularización de los dispositivos móviles y teléfonos celulares ha traído consigo el desarrollo de nuevas tecnologías que permitan una comunicación clara, rápida y precisa. Un ejemplo de esto es el surgimiento de la tecnología SMS de mensajería instantánea, que en sus comienzos era usada por los proveedores únicamente para dar informes de errores, servicios, etc. a los clientes, y que en la actualidad se ha convertido en uno de los canales de comunicación entre clientes más populares en todo el mundo. Esto básicamente porque la tecnología SMS permite enviar y recibir texto en cualquier momento, no importa donde se este, con un costo que dependerá de lo establecido por los proveedores de servicio.

Tomando en cuenta lo expuesto anteriormente, se puede afirmar que los teléfonos celulares han generado un gran impacto en la sociedad por las ventajas que estos proporcionan. Adicionalmente, el servicio de mensajería de texto ofrecido por estos dispositivos celulares se ha convertido en una forma de comunicación muy popular por su facilidad de acceso, las posibilidades de comunicación con otras personas y la sencillez de su concepción. Por otro lado se desea ofrecer un servicio o un conjunto de servicios (en este caso envío y recepción de SMS) que sea(n) independiente(s) de la plataforma y sistema operativo donde se ejecute(n), y de esta manera dar mayor flexibilidad, uniformidad y portabilidad a los usuarios que utilicen estos servicios.

De esta forma, se sugiere el desarrollo de un middleware, el cual provea una serie de servicios a través de un API, para la realización de aplicaciones simples o complejas que hagan uso de la tecnología SMS, haciendo abstracción de la plataforma y sistema operativo donde corran estas aplicaciones.

Para la realización del middleware, se plantea la utilización del lenguaje de programación Ruby, que por su facilidad de uso y flexibilidad ayudará a la implementación de mejoras futuras. Adicionalmente se propone el encapsulamiento de dicho middleware en un gem de extensión Ruby, de esta manera, otros desarrolladores podrán utilizar de manera sencilla las funcionalidades provistas por el middle-

ware. Utilizando RubyGems se puede manipular dicho gem fácilmente, una vez descargado desde el repositorio central de gems RubyForge.

Para lograr la comunicación con los distintos dispositivos la aplicación se basa en una librería llamada Gnokii. Gnokii es una librería de código abierto escrita en leguaje C/C++ que proporciona un API para el manejo de teléfonos celulares que sean compatibles con la tecnología GSM y que soporten los conocidos comandos AT, en particular los comandos AT extendidos que permiten ejecutar funciones relacionadas al manejo de los SMS (en la actualidad casi la totalidad de los celulares tienen soporte para los comandos AT). En base a esto deben incluirse mecanismos que permitan la generación de extensiones para Ruby en `C/C++`. En la Figura 2.1 se representa una visión general de la propuesta.

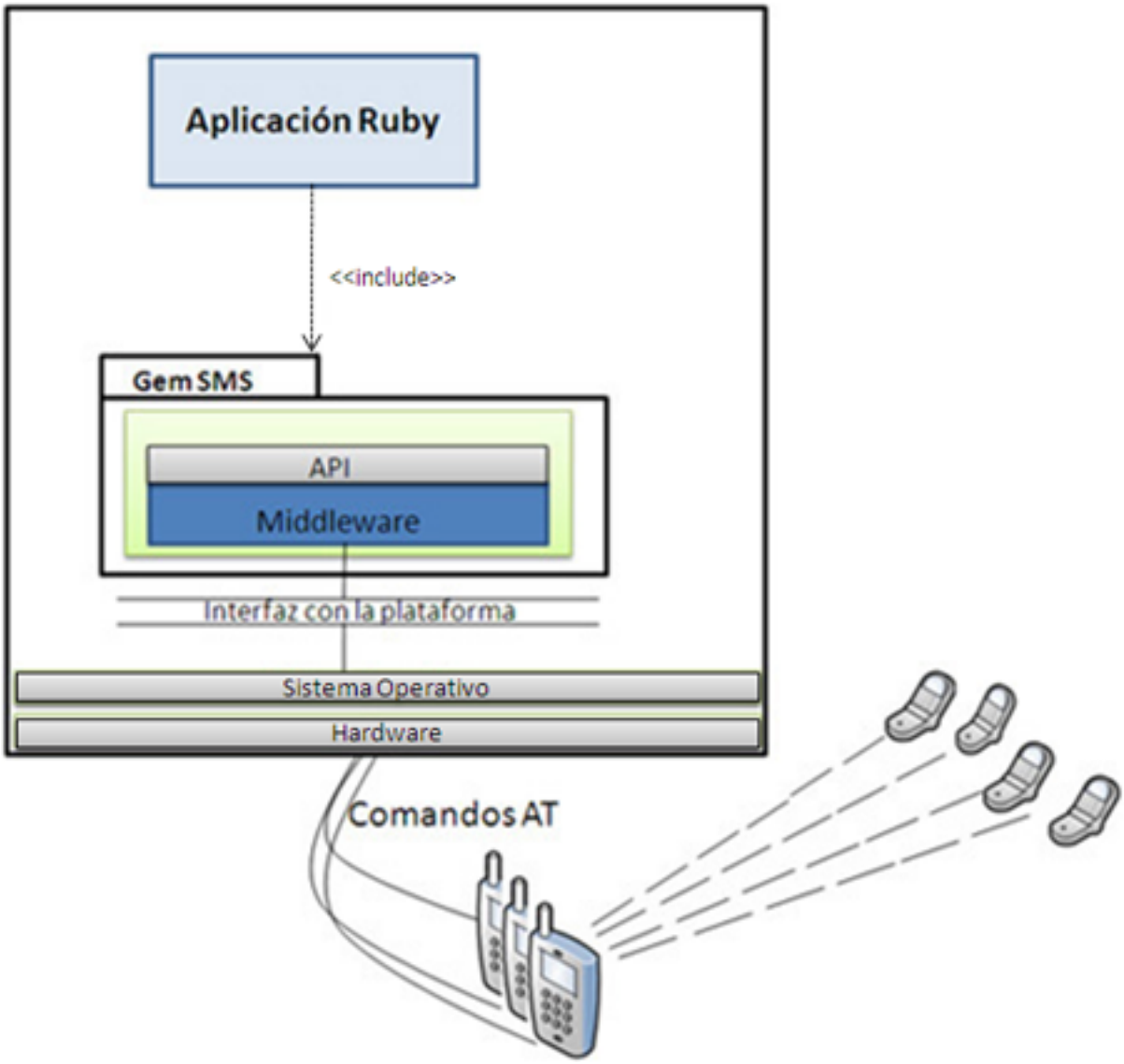

Figura 2.1: Visión general de la propuesta.

Con el escenario propuesto deben generarse extensiones para Ruby a partir del código desarrol-

lado en C/C++ utilizando alguna herramienta complementaria como SWIG, que ayuda a la creación de la misma, seguidamente dichas extensiones deben ser incorporadas a los módulos desarrollados en Ruby que forman parte del middleware SMS, logrando así, una integración completa. Posteriormente cualquier aplicación Ruby podrá hacer uso de las funciones desarrolladas en el lenguaje C/C++, pero ahora, a través de un gem que encapsula dicho middleware. Este escenario se encuentra reflejado gráficamente en la Figura 2.2.

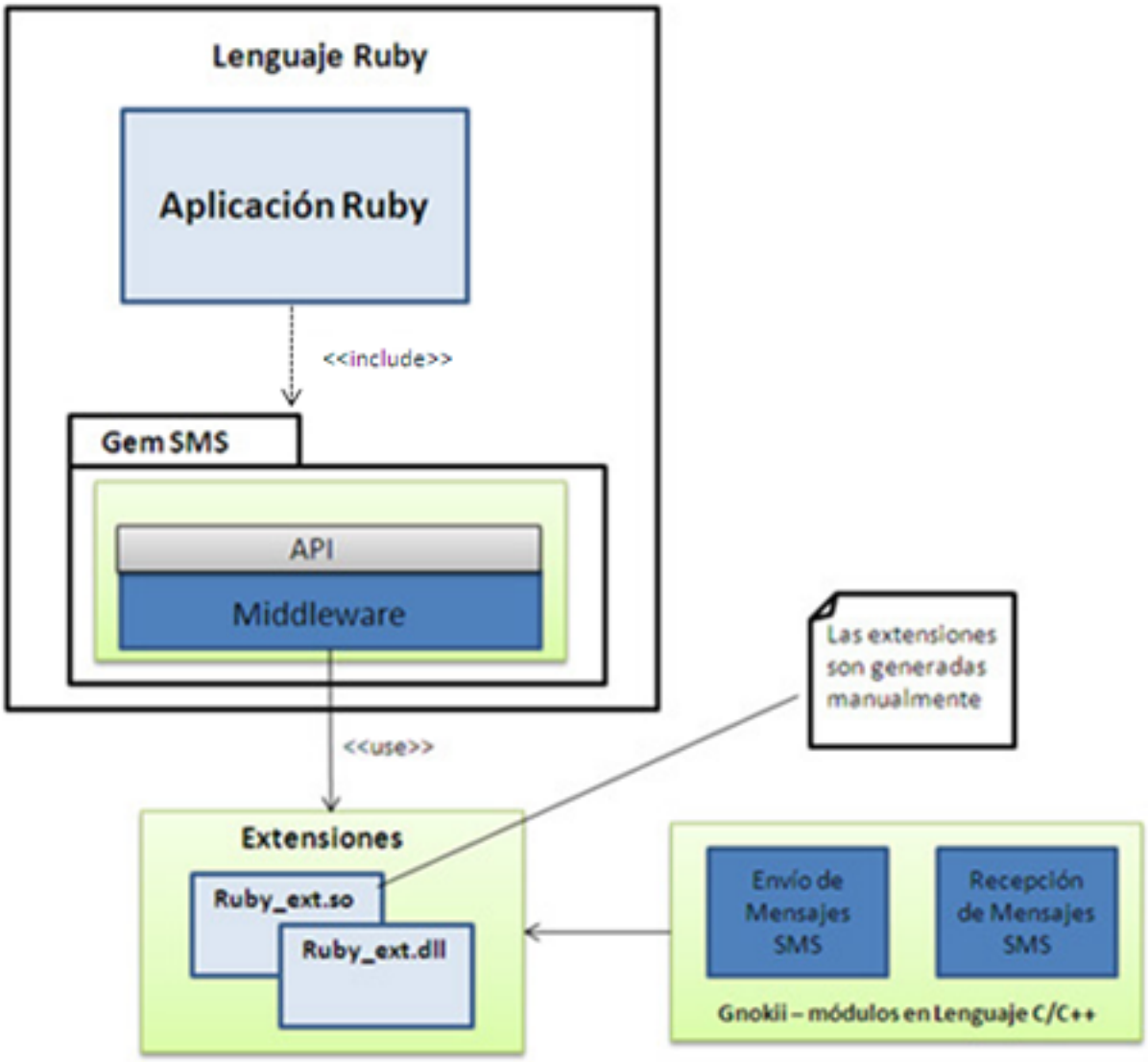

Figura 2.2: Propuesta utilizando extensiones generadas a través de SWIG

2.1 Objetivos Generales

Crear un gem de extensión de Ruby donde se encapsulen un conjunto de protocolos y servicios que permitan el envío y recepción de mensajes cortos de texto. Dichos protocolos y servicios serán ofrecidos

por un middleware SMS, a través de un API.

2.2 Objetivos Específicos

- Utilizar el proceso de desarrollo XP [1], y la metodología AM [2] para la construcción del gem.
- Construir un gem que sea portable y permita a los usuarios utilizar sus funcionalidades en diversas plataformas y sistemas operativos.
- Realizar la integración de la librería Gnokii desarrollada en C/C++ con módulos realizados en Ruby.
- Implementar módulo para el envío de mensajes cortos de texto.
- Implementar módulo para la recepción de mensajes cortos de texto.
- Probar los servicios ofrecidos por el gem con distintos dispositivos celulares.
- Publicar el gem en los repositorios de Github y RubyForge.

2.3 Tecnologías

Entre las tecnologías que se usaran en el Trabajo Especial de Grado para la realización de un gem de Ruby que permita el envío y recepcion de mensajes de texto están:

- El lenguaje de programación Ruby 1.8.6.
- Sistema de gestión de paquetes RubyGems.
- El lenguaje de programación C/C++. Compilador cl 15.00.21022.08 para Windows y el compilador gcc 4.3.2 para Linux.
- El intérprete de Ruby 1.8.6.
- El IDE de programación NetBeans 6.1.
- La librería de código abierto Gnokii 0.6.27 para el manejo de celulares.
- Dispositivos Celulares:
  - Motorola RZR V3
  - Motorola Motorizr Z3
  - Motorola ROKR E8
  - Samsung SGH - E370
  - Sony Ericsson w910i
  - Nokia 6101
  - Nokia 5220

[1] Extreme Programming
[2] Agile Modeling

2.4 Plataformas

Entre los objetivos de esta propuesta está la realización de un gem de Ruby que sea portable a distintas plataformas. Esto quiere decir que la aplicación desarrollada debe tener soporte, en principio, para varios sistemas operativos entre los que se pueden listar Windows, Linux y Mac OS.

2.5 Procesos y Metodologías de Desarrollo

Se utilizará el proceso de desarrollo XP en conjunto con la metodología AM, siguiendo sus prácticas y actividades para la construcción del gem.

2.6 Alcance

Los objetivos planteados para este trabajo definen un alcance particular. Este alcance está determinado por los siguientes aspectos:

- Utilizar la propuesta del proceso de desarrollo XP y la metodología de diseño AM.
- Integrar el API para C/C++ de la librería de código abierto Gnokii con módulos hechos en Ruby.
- Permitir la utilización de varios dispositivos celulares de manera simultánea.
- Desarrollar el sistema utilizando la propuesta de desarrollo de paquetes de software de Ruby (RubyGems).
- Ofrecer soporte multiplataforma para el gem desarrollado.
- Publicar el gem en los repositorios de Github y RubyForge.

Parte II

Marco Conceptual

Capítulo 3

Metodologías de Desarrollo

La ingeniería del software es un área de la informática que provee métodos para desarrollar software de calidad. Estas metodologías de desarrollo proveen mecanismos, técnicas y ayudas a la documentación, además detallan la información que se debe producir como resultado de una actividad y la información necesaria para comenzarla.

Son muchas las propuestas metodológicas que existen hoy en día que inciden en distintas dimensiones del proceso de desarrollo. Las metodologías ágiles dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas, proporcionando simplicidad y rapidez en la creación de sistemas.

En este capítulo se presenta resumidamente el contexto en el que surgen las metodologías ágiles, sus valores, principios y comparaciones con las metodologías tradicionales. Además se describe con mayor detalle el proceso de desarrollo XP y la metodología AM [1].

3.1 Metodologías Ágiles

Existen numerosas propuestas de metodología para desarrollar software. Tradicionalmente estas metodologías se centran en el control del proceso, estableciendo rigurosamente las actividades, herramientas y notaciones al respecto. Dadas estas reglas dichas metodologías se caracterizan por ser rígidas y dirigidas por la documentación que se genera en cada una de las actividades desarrolladas.

Este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término "ágil" aplicado al desarrollo de software. En esta reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto.

Tras esta reunión se creó The Agile Alliance [Canós and Letelier, 2002], una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a

[1] Agile Modeling

las organizaciones para que adopten dichos conceptos. El punto de partida fue el Manifiesto Ágil, un documento que resume la filosofía ágil.

Según el Manifiesto se valora:

- Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas. La gente es el principal factor de éxito de un proyecto software. Antes de construir el entorno de trabajo se debe buscar el equipo de desarrollo, no esperar que el equipo se adapte al entorno, sino que ellos lo configuren guiados por sus propias necesidades.
- Desarrollar software que funcione, más que conseguir una buena documentación. No se van a producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Estos documentos deben ser cortos y centrarse en lo fundamental.
- La colaboración con el cliente más que la negociación de un contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- Responder a los cambios más que seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

1. La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.
2. Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.
3. Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.
4. La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.
5. Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.
6. El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.
7. El software que funciona es la medida principal de progreso.
8. Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.
9. La atención continua a la calidad técnica y al buen diseño mejora la agilidad.
10. La simplicidad es esencial.

11. Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.

12. En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.

3.2 Proceso de desarrollo XP

XP es un proceso para el desarrollo de software, que se puede incluir entre las metodologías ágiles ya que agrega características importantes mencionadas en el manifiesto ágil, además incorpora características de metodologías tradicionales, utilizando lo más práctico y eficaz.

XP [xprogramming.org, 2008] se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuado para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Según Kent Beck, creador de esta metodología, "la programación extrema es una forma ligera, eficiente, flexible, predecible, científica y divertida de generar software"[Beck, 1999]. El objetivo principal de XP es muy simple: la satisfacción del cliente. Esta metodología trata de dar al cliente el software que necesita y cuando lo necesita. Por tanto, se debe responder muy rápido a las necesidades del cliente, incluso cuando los cambios sean al final del ciclo programación.

Un segundo objetivo importante es potenciar al máximo el trabajo en grupo. Tanto los jefes de proyecto, los clientes y desarrolladores, son parte del equipo y están involucrados en el desarrollo del software.

3.2.1 Prácticas

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione. XP apuesta por un crecimiento lento del costo del cambio. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de las prácticas que se describirán a continuación.

Equipo Completo

Todos los que estén contribuyendo en un proyecto XP son miembros de un equipo. Este equipo debe incluir un representante de negocios, el cliente y quien provee los requerimientos y fija las prioridades en el proyecto. Es mejor si el cliente o alguno de sus colaboradores es un usuario final ya que puede saber que se necesita.

El equipo, por supuesto, tiene que tener programadores. Los analistas pueden ayudar a los clientes a definir los requerimientos. Ninguno de estos roles es necesariamente propiedad exclusiva de un solo individuo: Todos en un equipo XP contribuyen de la manera en que pueden. El mejor equipo no tiene especialistas, solo contribuidores generales con habilidades especiales.

Planificación

XP enfoca la planificación en dos aspectos claves en el desarrollo de software, predecir que se quiere lograr para el día esperado y determinar que será lo siguiente a realizar. El énfasis está en dirigir el proyecto (lo cual es bastante sencillo), en lugar de predecir que será necesario y cuanto tiempo tomará (lo cual es bastante complicado). Existen dos pasos claves para la planificación en XP:

La planificación de las entregas es una práctica donde el cliente presenta las características deseadas a los programadores, y los programadores estiman la dificultad. Teniendo el costo estimado y los conocimientos de las características, el cliente define un plan para el proyecto.

La planificación de las iteraciones es una práctica donde el equipo recibe instrucciones cada varias semanas. El equipo XP construye software en iteraciones de dos semanas, entregando software funcional al final de cada iteración. Durante la planificación de las iteraciones, el cliente presenta las características deseadas para las siguientes dos semanas. Los programadores las dividen en tareas y estiman sus costos (en un nivel más detallado que en la planificación de entregas). Basados en la cantidad de trabajo realizado en las iteraciones anteriores, el equipo decide que se realizará en la iteración actual.

Estos pasos de planificación son muy sencillos, además proveen buena información al cliente. Este enfoque en resultados visibles es una pequeña paradoja, por un lado, el cliente está en posición de cancelar el proyecto si el progreso no es suficiente. Por otro lado, el progreso es tan visible, y la habilidad de decidir qué es lo siguiente en realizar es tan completa, que los proyectos XP tienden a dar más de lo que es requerido, con menor presión y estrés.

Test del Cliente

Como parte del proceso de presentación de las características del proyecto, el cliente de XP define uno o más test de aceptación para mostrar que cada característica funciona. El equipo construye estos test y los utilizan para realizar pruebas en conjunto con el cliente, para certificar que las funcionalidades estén implementadas correctamente. La automatización es importante ya que con la presión del tiempo, se saltan las pruebas manuales.

Versiones pequeñas

Los equipos XP hacen las entregas de dos maneras importantes:

Primero, el equipo entrega software ejecutado, probado y solicitado por el cliente en cada iteración. El cliente puede utilizar este software para cualquier propósito, ya sea para su evaluación o incluso para la entrega de éste al usuario final (muy recomendado). El aspecto más importante es que el software es visible y es entregado al cliente al final de cada iteración, mostrando resultados tangibles.

Segundo, los equipos XP también realizan entregas a los usuarios finales. Los proyectos Web realizados con XP son entregados diariamente. En proyectos internos son entregados mensual o más frecuentemente.

Diseño simple

Los equipos XP construyen software a partir de diseños simples. Se comienza con un diseño simple, y luego, a través de las pruebas de programación y las mejoras en el diseño, se mantiene de la misma manera. Un equipo XP mantiene el diseño adaptado a la funcionalidad actual del sistema. No hay pérdida de trabajo y el software está siempre listo para lo que hay que realizar en la siguiente etapa.

Diseñar con XP no es cosa de una sola vez, sino algo que lleva un tiempo considerable. Hay pasos para diseño, planificaciones de entregas y planificaciones de iteración, además, los equipos realizan sesiones rápidas de diseño y revisiones de diseño a través de refactorizaciones que se realizan a lo largo de todo el proyecto.

Programacion en parejas

Todos los productos en XP son construidos por dos programadores, sentados juntos, en el mismo computador. Esta práctica asegura que todo el código de producción es revisado al menos por otro programador siendo el resultado mejor diseñado, mejor probado y mejor códificado.

Se puede ver como ineficiente tener a dos programadores haciendo "el trabajo de un solo programador", pero lo contrario resulta ser cierto.

Desarrollo guiado por las pruebas automáticas

La programación extrema está obsesionada con la retroalimentación, y en el desarrollo de software, la buena retroalimentación requiere buenas pruebas. Los mejores equipos de XP practican el desarrollo guiado por pruebas automáticas, trabajando en ciclos muy cortos para añadir una prueba y hacer que funcione. Casi sin esfuerzo, los equipos producen códigos con casi el 100 por ciento de la cobertura de la prueba, que es un gran paso adelante en la mayoría de los casos.

Mejora del diseño

La programación extrema se enfoca en entregar valores de negocio en cada iteración. Para lograr esto a lo largo de todo el proyecto, el software debe estar bien diseñado. Así XP utiliza un proceso de mejora continua de diseño llamado refactorización.

El proceso de refactorización se centra en la eliminación de la duplicación (un signo seguro de mal diseño), y en el aumento de la cohesión del código, mientras que se reduce el acoplamiento. Alta cohesion y bajo acoplamiento han sido reconocidas como las características de buen diseño de código de al menos treinta años. El resultado es que los equipos XP empiezan con un diseño simple y generalmente, aumentan la velocidad a medida que el proyecto sigue adelante.

La refactorización, es fuertemente soportada por el diseño comprensivo de las pruebas, para asegurar que nada se dañe mientras el diseño evoluciona. Por eso, las pruebas del cliente y de los programadores son un factor crítico para continuar el desarrollo. Las prácticas XP se soportan unas con otras. Son más valoradas cuando se hacen juntas que cuando se hacen por separado.

Integración continua

Los equipos de XP mantienen el sistema completamente integrado todo el tiempo. La ventaja de esta práctica puede ser vista pensando en proyectos anteriores, donde el proceso de integración se realiza semanalmente o con menor frecuencia. Usualmente, esto conlleva a problemas de integración donde el proyecto se cae y no se sabe por qué.

Las integraciones que se hacen con menos frecuencia convellan a serios problemas en un proyecto de software. Primero que nada, aunque la integración es lo más importante para entregar un código funcional y ejecutable, el equipo no está acostumbrado a ésto, y a menudo se delega la responsabilidad a personas que no están familiarizadas con el proyecto. Segundo, las integraciones que se hacen con poca frecuencia tienen muchos errores a nivel de código. Los problemas se acarrean si no son detectados por las pruebas. Tercero, los procesos de integración pobres conllevan a un congelamiento del código. Es decir, que pasarán largos períodos de tiempo en los que los programadores pueden estar trabajando en características importantes, pero esas características deben ser aplazadas. Esto debilita la posición del equipo en el mercado o con el usuario final.

Código de propiedad colectiva

En un proyecto XP, cada pareja de programadores puede mejorar cualquier código en cualquier momento. Esto significa que todo el código recibe el beneficio de la atención de muchas personas. Lo cual incrementa la calidad del mismo y reduce sus defectos.

La propiedad colectiva puede ser un problema si las personas que contribuyen en el código lo hacen sin entenderlo. XP evita estos problemas mediante dos técnicas claves: los test de los programadores, que permiten capturar los errores, y la programación en parejas. Lo que significa que la mejor manera de trabajar con código que no es familiar es con un experto.

Normas de Codificacion

Los equipos XP siguen unos estándares de codificación, para que así, todo el código en el sistema luzca como si hubiese sido escrito por un único y muy competente individuo. Las especificaciones del estándar no son importantes, lo importante es que todo el código luzca familiar, para soportar la propiedad colectiva.

Metáforas

Una metáfora no es más que la descripción simple de cómo el programa debería funcionar. Los equipos XP se dedican a desarrollar esta metáfora a través de una visión común.

Algunas veces es difícil encontrar una metáfora. En todo caso, con o sin una imagen vívida, los equipos XP usan un sistema común de nombres para asegurarse que todos entienden cómo funciona el sistema, dónde buscar una funcionalidad o como encontrar el lugar dónde colocar dicha funcionalidad. Una metáfora puede ser un diagrama sencillo, un conjunto de figuras que describen el comportamiento de un módulo o cualquier otro elemento que sugiera una descripción del sistema.

Ritmo Sostenible

Los equipos XP se conforman a largo plazo. Trabajan fuerte a un ritmo que puede ser sostenido indefinidamente. Esto significa que el equipo puede trabajar tiempo extra si es efectivo, y que trabajan de tal manera que se maximiza la productividad.

3.2.2 Actividades

Las actividades de XP se reflejan en la Figura 3.1 y se detallan a continuación:

Planificación

- Se escriben historias de usuarios. el propósito de las historias de usuarios es describir en dos o tres oraciones los requerimientos del sistema en terminología del cliente (generalmente escritos por el mismo cliente), conduciendo a la creación de las pruebas de aceptación y proporcionando a su vez una estimación del tiempo necesario para el desarrollo. El tiempo ideal para cada historia de usuario es de 1 a 3 semanas y cerca de 80 o 20 historias como mínimo son el número ideal para el primer levantamiento de información en la actividad de planificación.

- Se crea una planificación de entrega, que debe servir para crear un calendario que todos puedan cumplir y en cuyo desarrollo hayan participado todas las personas involucradas en el proyecto. Se usará como base las historias de usuarios, participando el cliente en la elección de las actividades

Figura 3.1: Actividades de XP

que se desarrollarán. Según las estimaciones de tiempo de los mismos se crearan las iteraciones del proyecto.

- Frecuentemente se hacen pequeñas entregas. El equipo de desarrollo entrega varias versiones del sistema al cliente y así, se pueden ir introduciendo las funcionalidades que no se habían contemplado hasta la entrega.
- El desarrollo se divide en iteraciones, esto agrega agilidad al proceso de desarrollo. Cada una de ellas comienza con un plan de iteración para el que se eligen las historias de usuarios a desarrollar.
- Las personas involucradas en el desarrollo del sistema se intercambian a las distintas áreas de codificación, evitando así, los cuellos de botella, y al mismo tiempo fomentado el conocimiento del código por parte de todo el equipo.
- Se evita la sobrecarga de trabajo a miembros en particular del desarrollo, consolidando un equipo entero totalmente productivo.
- Se realizan los cambios específicos que son necesarios para adaptarlos al desarrollo del sistema.

Diseño

- Los diseños deben ser los mas sencillo posible, mientras más sencillos sean, será más fácil agregar una funcionalidad en la programación.
- Se escoge una metáfora de sistema y convenciones en cuanto a los métodos. El objetivo primordial de la metáfora es mejorar la comunicación entre los integrantes del equipo desarrollador, al crear una visión global y general de lo que se quiere desarrollar. La metáfora tiene que ser expresada en términos conocidos por las personas involucradas en el desarrollo.
- Se escriben tarjetas de Clases, Responsabilidades y Colaboración (Classes, Responsabilities and colaboration. CRC) apara diseñar sistemas como equipo. El objetivo más grande de las tarjetas

CRC es que los desarrolladores rompan con la estructura del pensamiento y aprecien más la tecnología del objeto.

- Se crean soluciones de punta para responder a los problemas de diseño o técnicos. Una solución de punta es un programa muy sencillo que explote soluciones potenciales para el desarrollo del sistema. El objetivo de estas soluciones es disminuir el riesgo de un problema técnico o incrementar la confiabilidad de la estimación de las historias de usuarios.

Codificación

- El cliente esta siempre disponible, ayudando al equipo desarrollador y formando parte de él. Todas las fases de XP requieren de la comunicación con el cliente, preferiblemente cara a cara. Durante todas las actividades el cliente ayuda con la estimación de tiempo para las historias de usuarios, ayuda con la asignación de las prioridades, se cerciora que las funcionalidades del sistemas cubran todas las historias de usuarios y participa en las reuniones de planificación para completar detalles de las tareas. Igualmente colabora con la elaboración de las pruebas funcionales.

- El código se ajusta a los estándares de codificación, manteniendo la consistencia y legibilidad del mismo, facilitando así, la comprensión y refactorización para los involucrados en el desarrollo del sistema.

- Las pruebas unitarias se crean antes que el código, facilitando y agilizando la codificación. Estas pruebas también ayudan a identificar las necesidades que realmente se tienen que considerar al momento de codificar el sistema.

- Todo el código de producción es programado por parejas, aumentando la calidad del mismo.

- La integración del código será realizada solo por una pareja.

- Los desarrolladores integran el código y entregan la integración del mismo frecuentemente. La integración continua evita la divergencia de código ya que el equipo desarrollador necesita trabajar con la última versión. También se detecta a tiempo los problemas de incompatibilidad de código.

- Se usa la propiedad colectiva del código, cualquier desarrollador puede cambiar una línea de código o refactorizar el mismo, con la finalidad de aportar nuevas ideas que mejoren el sistema. La propiedad colectiva del código se hace una práctica confiable al evitar que una sola persona se encargue de esta tarea, especialmente si esa persona puede abandonar el proyecto en un momento dado.

- La optimización del código se hace al final.

- Se evita trabajar horas extras.

Pruebas

- Todo el código debe tener pruebas unitarias asociadas y éste debe ser pasado por las pruebas antes de la entrega del sistema final.

- Si se encuentran errores en el código se crean otras pruebas para demostrar el error específico.

- Se realizan pruebas de aceptación frecuentemente y se publican los resultados. Las pruebas de aceptación se hacen en base a las historias de usuarios, estas pruebas de aceptación son conocidas como pruebas de caja negra, donde cada prueba representa un cierto resultado previsto por el sistema. Una historia de usuario no se considera completa hasta que no haya pasado satisfactoriamente las pruebas de aceptación.

3.3 Metodologia AM

AM es una metodología basada en la práctica para modelado efectivo y documentación de sistemas de software. La metodología AM es una colección de prácticas, guiadas por principios y valores que pueden ser aplicados por profesionales de software en el día a día. AM no es un proceso prescriptivo, ni define procedimientos detallados de como crear un tipo de modelo dado. En lugar de eso, sugiere prácticas para ser un modelador efectivo. Es "suave al tacto", no es duro y es rápido Se piensa en AM como un arte, no una ciencia. [agilemodeling.com, 2008]

3.3.1 Principios

Asumir simplicidad

Mientras se desarrolla se debe asumir que la mejor solución es la solución más simple. No sobrecargar el software, o en el caso de AM no representar características adicionales en el modelo que no se necesitan en el momento. Mantener el modelo tan simple como sea posible.

Bienvenida al cambio

Los requerimientos van evolucionando con el tiempo. Estos requerimientos pueden cambiar a medida que el proyecto avanza, nuevos desarrolladores son agregados y otros pueden retirarse del proyecto. Los puntos de vistas pueden cambiar al igual que las metas y los criterios de éxito del esfuerzo realizado. La consecuencia es que el ambiente de desarrollo cambia, y como resultado las propuestas y aportes deberán reflejar esa realidad.

Permitir el siguiente esfuerzo es el objetivo secundario

El proyecto puede ser considerado un fracaso aún cuando el equipo de desarrollo ofrezca un software de trabajo a los usuarios. Parte de satisfacer las necesidades de los interesados del proyecto es asegurar que el sistema sea lo suficientemente robusto como para que pueda ser extendido en el tiempo.

Cambio incremental

Un concepto importante en cuanto al modelado es que no tiene por qué hacerse bien la primera vez, de hecho, es muy poco probable que pueda hacerse perfecto en sólo un intento. Además, no es necesario captar cada detalle en el modelo, sólo se necesita que sea lo suficientemente bueno en el momento. En lugar de tratar de desarrollar un modelo que abarque todo al comienzo, se deberá comenzar por el desarrollo de un modelo pequeño, o tal vez un modelo de alto nivel, y evolucionar a lo largo del tiempo de manera incremental.

Maximizar la inversión de las partes interesadas en el proyecto

Las partes interesadas en el proyecto (clientes, desarrolladores,promotores del desarrollo) invierten recursos tales como tiempo, dinero, facilidades, entre otros, para realizar un desarrollo de software

que cumpla con todas las necesidades. Los interesados en el proyecto buscan que sus recursos sean invertidos de la mejor manera posible y que no sean malgastados por los integrantes del equipo de desarrollo. Además, merecen tener la última palabra con respecto a la forma en que esos recursos son invertidos.

Modelar con un propósito

Muchos desarrolladores se preocupan sobre si sus artefactos, tales como modelos, códigos fuentes o documentos, son lo suficientemente detallados o por el contrario, hay muchos detalles. En lo que respecta al modelado, quizás sea necesario entender mejor un aspecto del software a desarrollar, comunicar las propuestas a los altos cargos para justificar el proyecto o quizás se necesite crear documentación que describa el sistema desarrollado para las personas que lo usarán. El primer paso sería identificar un propósito válido para crear el modelo y así mismo identificar la audiencia a quien está dirigido dicho modelo. Seguidamente, con base en el propósito y la audiencia, desarrollarlo de forma tal que sea lo suficientemente acertado y detallado. Una vez cumplidas las metas del modelo estará completado por el momento y se deberán hacer cosas adicionales, como escribir código para mostrar cómo funciona el modelo. Este principio también es aplicado para realizar cambios en un modelo existente (si se está realizando un cambio) quizás aplicando un patrón conocido. Deberá existir un razón válida para realizar ese cambio (soportar un nuevo requerimiento o refactorizar el trabajo para hacerlo más limpio).

Múltiples modelos

Eventualmente se necesitará el uso de múltiples modelos de desarrollo de software, ya que cada modelo describe un simple aspecto del mismo. Considerando la complejidad del software moderno, se necesitará un amplio rango de técnicas en el conjunto de herramientas de modelado para ser realmente efectivos. Un punto importante es que no todo el tiempo se necesitarán todos esos modelos para cualquier desarrollo de sistemas, pero dependiendo de la naturaleza exacta del software que se esté desarrollando, será necesario tener por lo menos un subconjunto de los modelos. Con frecuencia se usarán más algunos tipos de modelos que otros.

Trabajo de calidad

A nadie le gusta el trabajo descuidado. A las personas que realizan el trabajo no les gusta porque en ocasiones no se sienten orgullosos de él, a las personas que realizan las actualizaciones y mantenimientos no les gusta porque es más difícil de entender y actualizar, y finalmente, a los usuarios no les gusta porque simplemente no cumple con sus expectativas.

Retroalimentación rápida

El tiempo entre una acción y la retroalimentación de esa acción es crítico. Si se trabaja con otras personas sobre un modelo (particularmente cuando se trabaja con una tecnología de modelado compartida), se obtiene casi instantáneamente una retroalimentación de las ideas. Trabajando cerca del cliente para entender los requerimientos, analizar esos requerimientos o para desarrollar una interfaz de usuario, provee oportunidades para una retroalimentación rápida.

El software es el objetivo primario

La meta del desarrollo de software es producir un software de trabajo de alta calidad que satisfaga las necesidades de los interesados en el proyecto de una manera eficaz. La principal meta no es producir

una documentación extraña, o extraños artefactos de gestión. Cualquier actividad que no contribuya directamente al logro de este objetivo debe ser evitada y cuestionada si no puede ser justificada.

Viaje con poco equipaje

Cada artefacto creado deberá ser mantenido en el tiempo. Cuanto más complejos y detallados sean los modelos, es más probable que cualquier cambio sea difícil de lograr. Un equipo de desarrolladores que decida crear y mantener un documento detallado de requerimientos, una detallada colección de modelos de análisis, modelos de arquitectura y modelos de diseño, descubrirá rápidamente que está gastando la mayor parte del tiempo en actualizaciones de documentos en lugar de escribir código.

3.4 Resumen

Las metodologías ágiles ofrecen solución a gran cantidad de proyectos. Tienen ventajas y también desventajas; una de las ventajas más destacadas es su sencillez, tanto en el aprendizaje como en su aplicación, reduciendo costos de implantación en un equipo de desarrollo. Por otra parte también posee una serie de inconvenientes y restricciones, como por ejemplo que están dirigidas a grupos pequeños y medianos. Adicionalmente, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar el proceso al fracaso.

XP es un proceso que fue diseñado para entregar al cliente el software que necesita y cuando lo necesita, respondiendo muy rápido a sus las necesidades, incluso cuando los cambios sean al final del ciclo de programación. Para lograr sus objetivos XP tiene una serie de principios, prácticas y valores básicos.

Por otra parte se describe AM como un complemento a los métodos existentes para modelado efectivo de sistemas de software. Posee prácticas guiadas por principios y valores que pueden ser aplicados por profesionales de software en el día a día.

Capítulo 4

Middleware

La modularidad y la capacidad de combinar plataformas son características que se buscan al momento del desarrollo de aplicaciones por sus efectos positivos en las mismas. Esto servirá para que los programadores construyan aplicaciones que no sólo se ejecuten en un mismo tipo de máquina o sistema operativo, sino que sean completamente transparentes a estos requisitos.

La forma más común de cumplir con este requisito es utilizar interfaces estándares de programación y protocolos que se sitúen entre la aplicación y el sistema operativo. Dichas interfaces han venido a llamarse middleware. A través de éstas, es fácil implementar una misma aplicación en tipos de máquinas diferentes. De aquí puede deducirse uno de los objetivos de diseño más importante del middleware que es la transparencia de migración, la cual expresa que los recursos accedidos pueden migrar de una plataforma a otra sin necesidad de afectar a los usuarios.

La variedad de tipos de middleware conlleva a un problema de decisión para los desarrolladores de software. En este capítulo se expone una clasificación de los tipos de middleware así como las ventajas y desventajas de cada uno.

4.1 Entendiendo y definiendo middleware

El middleware es una capa de software independiente del hardware y sistema operativo, que proporciona un conjunto de servicios no ofrecidos en dicho sistema operativo; una serie de servicios, estándares de programación y protocolos, que facilitan el desarrollo de aplicaciones. La gran ventaja es que cada computador puede ejecutar un sistema operativo distinto, y aun así, las aplicaciones montadas sobre el middleware funcionar correctamente. Esto hace más atractiva su utilización.

Generalmente un middleware es elaborado para facilitar el diseño de aplicaciones distribuidas, esto debido a que su arquitectura es muy similar a la de un entorno que utiliza un sistema operativo distribuido. Es importante considerar el papel que tiene el middleware desde un punto de vista lógico, más que desde la implementación.[Jesus Carretero Perez and Costoya, 2001, Stalling, 2005]

La Figura 4.1 muestra este punto de vista.

Todas las aplicaciones operan sobre una interfaz uniforme de programación de aplicaciones API, Aplication Programming Interface. El middleware es entonces el encargado de proporcionar la trans-

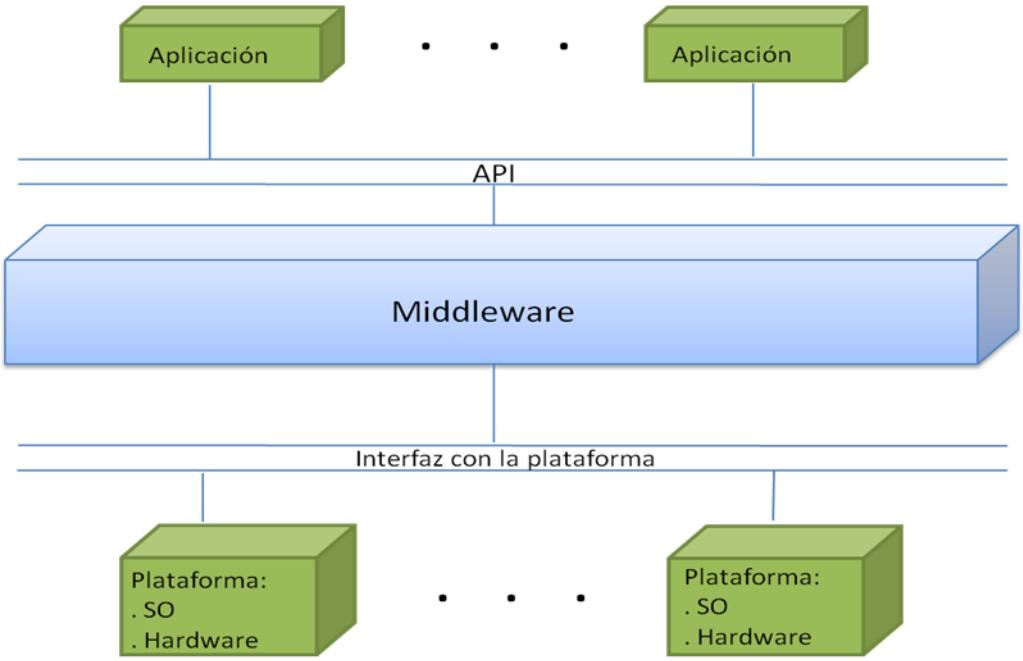

Figura 4.1: Arquitectura de un middleware

parencia entre diferentes plataformas.

4.2 Capas de middleware

Pueden existir muchas capas de middleware presentes en una configuración dada de un sistema. Por ejemplo, un sistema middleware de bajo nivel, como un servicio de broadcast síncrono y atómico, puede ser usado directamente por programadores de aplicaciones. Sin embargo, algunas veces es usado como un bloque para la construcción de middlewares de alto nivel como CORBA o middlewares orientados a mensajes que permitan proveer tolerancia a fallas, balanceo de carga o ambas.

Hay que resaltar que la mayoría de la implementación de un sistema middleware es a nivel de aplicación, capa 7 en la arquitectura de referencia OSI, aunque parte de esa implementación esta también a nivel de presentación, capa 6 en la arquitectura de referencia OSI. Así, el middleware es una aplicación para los protocolos de red, que se encuentran en el sistema operativo [Bakken, 2000].

4.3 Middleware y Sistemas Legacy

Los middleware son algunas veces referenciados como una tecnología de puente, debido a que son generalmente usados para integrar componentes legacy. El Middleware es esencial para la migración de aplicaciones mainframe que nunca fueron diseñadas para interactuar o ser enlazadas y conectadas con peticiones de servicios remotos. Los Middleware también son útiles para envolver dispositivos de red como routers y estaciones base móviles, para ofrecer integraciones a nivel de red y proporcionar mantenimiento de un API de control que provea interoperabilidad al más alto nivel. Los middleware

orientados a objetos son particularmente adecuados para la integración legacy, debido a sus características. [Bakken, 2000]

4.4 Programación con middleware

Los programadores no tienen que aprender un nuevo lenguaje de programación para programar interactuando con middlewares. En su lugar pueden usar un lenguaje existente con el que estén familiarizados, como C++ o java. Hay tres formas principales en la que se puede interactuar con un middleware:

La primera forma es proporcionando una librería de funciones que pueden ser llamadas para utilizar el middleware; los sistemas de base de datos distribuidos hacen esto.

La segunda forma es a través de un Lenguaje de Descripción de Interfaces (IDL, Interface Description Lenguage). De esta forma, se describe la interfaz para el componente remoto y un mapeo de el IDL al lenguaje de programación que es usado por los programadores. [Bakken, 2000]

La Tercera forma es que el sistema y lenguaje tengan soporte nativo para operaciones distribuidas, por ejemplo la Invocación Remota de Métodos (RMI, Remote Method Invocation) de java.

4.5 Resumen

La mayoría de las aplicaciones distribuidas exitosas a grande escala y algunas aplicaciones no distribuidas involucran el uso de middleware para resolver problemas de interoperabilidad. Middleware, como su nombre quiere dar a entender, es software que se ubica entre las aplicaciones y sistema operativo, que generalmente es la fuente de los problemas de compatibilidad.

Con la inserción de una capa de software como la de más alto nivel (en donde residirán las aplicaciones), se provee un grado de encapsulamiento o abstracción de los servicios de más bajo nivel. De hecho, el middleware introduce nuevas APIs que son usadas para invocar estos nuevos servicios de más alto nivel. Debido a ésto, el middleware ayuda a aislar las aplicaciones de cambios en la plataforma, las redes, los sistemas operativos, etc. Se puede entonces sustituir las tecnologías subyacentes, usando algunas más efectivas y eficientes, sin tener que realizar cambios a la aplicación. Adicionalmente la abstracción de los servicios a través de las APIs de alto nivel también simplifica la programación de aplicaciones, permitiendo a los programadores modificar o crear aplicaciones más rápidamente.

Capítulo 5

Servicio de Mensajes Cortos

El servicio de mensajes cortos (SMS, Short Message Service) [Bodic, 2003] es un servicio básico que permite el intercambio de texto entre suscriptores.

Actualmente están apareciendo gran cantidad de servicios basados en mensajes cortos. Además de ser usados para enviar mensajes de texto entre personas, simulando a los llamados *busca personas*, se están ofreciendo otros servicios como:

- Votaciones mediante SMS.
- Subscripción a servicios de información.
- Informe de averías en ciertos equipos. Por ejemplo, muchos cajeros automáticos envían un SMS al servicio técnico cuando detectan que hay alguna avería o les falta algún recurso: dinero, papel, etc.

Se cree que el primer mensaje corto de texto fue enviado en 1992 a través de canales de señalización de una red GSM[1] europea. Desde esta exitosa prueba, el uso de los SMS ha tenido un crecimiento muy grande. La asociación de data móvil reportó que el número total de cargos por mensajes de texto enviados de persona a persona a través de las 4 redes GSM en el reino unido en 2003 fue totalizado en aproximadamente 20.5 billones.

5.1 GSM-SMS

GSM es un sistema de comunicaciones móviles que provee servicios como transmisión/recepción de voz y transmisión/recepción de datos. Sin embargo, uno de los servicios mas populares ofrecidos por GSM son los mensajes cortos de texto. La implementación de SMS implica la inclusión de varios elementos claves en la arquitectura GSM. La Figura 5.1 muestra la interacción de estos elementos.

Los dos elementos más importantes para el soporte de SMS son: el centro SMS (SMSC, SMS Center) y el Email Gateway. Adicionalmente, es necesario un elemento llamado entidad de mensajes cortos (SME, Short Message Entity), usualmente presentada como una aplicación de software en un dispositivo móvil para el manejo de los mensajes envío, recepción, almacenamiento, etc.

[1] Sistema global para las comunicaciones moviles (proviene dde "Groupe Spécial Mobile")

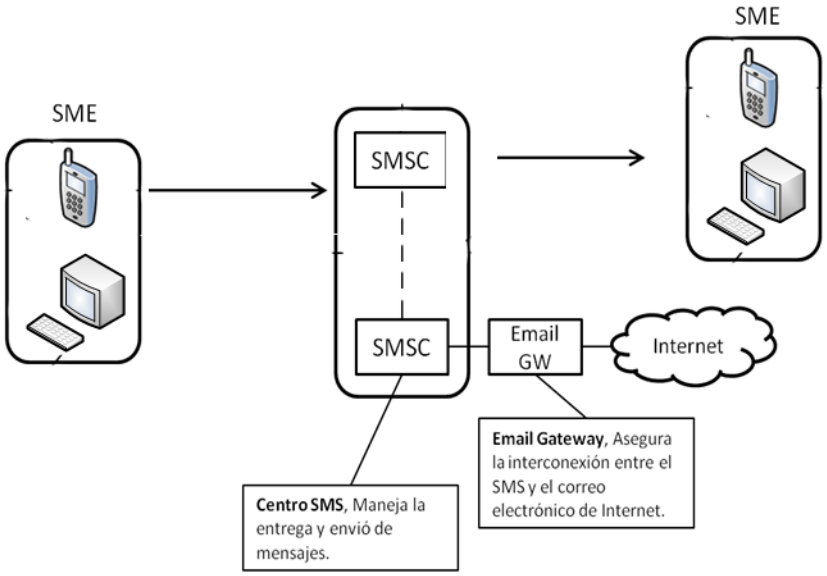

Figura 5.1: Red GSM con soporte para SMS

5.2 Entidad de Mensajes Cortos (SME, Short Message Entity)

Aquellos elementos que pueden enviar o recibir mensajes cortos de texto son llamados Entidades de mensajes cortos. Un SME puede ser una aplicación de software en un teléfono celular, como puede ser también un teléfono de envío de fax, un servidor remoto de internet, etc. El Dispositivo debe ser configurado para que pueda funcionar correctamente en una red móvil. Generalmente es pre-configurado durante su elaboración, pero una configuración manual también puede ser realizada.

En el intercambio de mensajes cortos, el SME que genera y envía el mensaje corto de texto es conocido como el SME origen, mientras que el SME que recibe el mensaje corto de texto es conocido como el SME receptor.

5.3 Centro de Servicio (SC, Service Center)

El centro de servicio o SMSC juega un papel clave, siendo su función principal la transmisión de mensajes cortos de texto entre los SMEs, así como el almacenamiento(si el SME no está disponible) y redirección de éstos. El SC puede estar integrado como parte de la red móvil, o como una entidad de red independiente. De manera práctica es muy común para los operadores de red la adquisición de uno o más SMSCs debido a la popularidad que SMS representa. En teoría, un simple SMSC podría manejar mensajes para varias redes móviles, sin embargo, este escenario se ve muy raras veces en la vida real y uno o varios SMSCs están generalmente dedicados al manejo de operaciones SMS en una sola red móvil.

Los operadores de red móvil usualmente tienen acuerdos comerciales entre ellos, para permitir el intercambio de mensajes entre las distintas redes. Esto significa que un mensaje enviado desde un SME que está asociado a una red en particular, puede ser entregado a otro SME asociado a una red distinta. Esta habilidad para los usuarios de poder intercambiar mensajes aún cuando no son subscriptores de

la misma red es, sin lugar a dudas, una de las características claves que hace de SMS un servicio tan popular y satisfactorio.

Los centros SMS actuales son generalmente capaces de procesar más de 1000 mensajes por segundo cada uno. Las grandes operadoras móviles usualmente hacen uso de varios centros SMS para satisfacer las necesidades de sus clientes.

5.4 Email Gateway

El Email Gateway hace posible una interoperabilidad entre Email y SMS, interconectando el centro SMS con Internet. Con el Email Gateway, los mensajes pueden ser enviados desde un SME a un host en internet, y viceversa. El papel del Email Gateway es convertir los formatos de mensaje (de SMS a email y viceversa) y enviar mensajes entre dominios SMS y de internet.

5.5 Características básicas de los SMS

Los SMS se caracterizan por una serie de funcionalidades que incluyen entrega de mensajes, envío de mensajes, manejo de reportes de estatus, camino de respuesta etc. A continuación se expondrán las mas importantes.

5.5.1 Envío y entrega de mensajes

Las dos características más importantes de SMS son envío y recepción de mensajes cortos.

Envío de mensajes

Los mensajes originados por los teléfonos celulares son enviados desde un suscriptor móvil (MS, móvil suscriptor) a un SMSC. Estos mensajes son direccionados a otro SME (el cual puede ser otro usuario móvil o un host en internet). Un SME origen puede especificar un tiempo de validez para el mensaje, luego del cual el mensaje es descartado. Un mensaje que deja de ser válido puede ser detectado por un SMSC durante la transferencia del mismo. Con las primeras redes GSM, no todos los dispositivos soportaban envío de mensajes cortos de texto, en la actualidad casi todos los dispositivos soportan envío de mensajes. Esta característica también es conocida como "Message-Mobile Originated (SM-MO)"

Entrega de mensajes

Los mensajes destinados a teléfonos celulares son entregados por el SMSC al MS. Casi todos los teléfonos celulares soportan la recepción de mensajes. Esta característica también es conocida como Short Message Mobile Terminated (SM-MT). Los mensajes originados por los teléfonos celulares y los mensajes destinados a teléfonos celulares pueden ser enviados y entregados respectivamente aún cuando una llamada de voz o una conexión de datos está en proceso. Los mensajes pueden ser enviados o recibidos a través de canales de señalización GSM.

5.5.2 Reportes de estado

Para un SME origen es posible solicitar la generación de un reporte de estado relacionado con la entrega del mensaje al SME destino. El reporte de estado indica al SME origen si el mensaje corto fue o no entregado de manera satisfactoria al SME destino.

5.5.3 Camino de respuesta

El camino de respuesta puede ser establecido por el SME origen para indicar que el SMSC (que maneja el envío del mensaje) está habilitado para manejar directamente una respuesta del SME destino, correspondiente al mensaje original. En esta situación, El SME destino generalmente envía el mensaje de respuesta directamente al SMSC que manejó el envío del mensaje original. Esta herramienta es usada ocasionalmente por lo operadores para permitir a los SME destino que provean un mensaje de respuesta "libre de cargos" para ellos.

Adicionalmente, para redes que soportan varios SMSCs, los operadores a veces usan esta herramienta para hacer que los mensajes de respuesta sean retornados a un SMSC específico. Por ejemplo, un operador podría tener varios SMSCs, pero sólo uno de ellos conectado al Email gateway. Con esta configuración, si un mensaje es originado desde el dominio de internet, el operador puede utilizar un camino para indicar que cualquier mensaje de respuesta asociado con el mensaje originado por email, debe ser enviado al SMSC conectado al Email Gateway. En este caso, el SMSC puede solicitar al Email Gateway la conversión de los mensajes de respuesta en mensajes de email que puedan ser entregados al destino de manera satisfactoria.

5.5.4 Período de Validez

Un SME origen tiene la posibilidad de indicar un período de validez para el mensaje. Este período de validez define el momento a partir del cual el contenido del mensaje sera descartado si un mensaje no ha sido entregado al destinatario antes de la fecha de expiración, así pues, la red descarta el mensaje sin realizar mas intentos de entrega al destinatario. Por ejemplo, un suscriptor podría mandar un mensaje con el siguiente contenido "Por favor regresa la llamada en las próximas dos horas para darte tu respuesta", adicionalmente, el suscriptor podría inteligentemente indicar que el período de validez del mensaje está limitado a dos horas, en cuyo caso si el destinatario no ha encendido su dispositivo en las dos horas siguientes al envío del mensaje, el mensaje será descartado.

El intercambio de un mensaje desde el SME origen hacia el SME destino consiste en de dos a tres pasos. Luego de la creación por parte del SME origen, el mensaje es enviado al SMSC (paso 1). El SMSC puede verificar con otros elementos de la red que quien envía el mensaje tiene permitido enviar mensajes (por ejemplo que tenga suficiente crédito prepago, que el suscriptor pertenezca a la red, etc.). El SMSC envía el mensaje al SME destino (paso 2). Si el SME destino no está disponible para que le sea entregado el mensaje, entonces el SMSC guarda el mensaje temporalmente hasta que el SME destino esté disponible, o hasta que expire el período de validez del mensaje. Mientras sea entregado el mensaje o mientras se borre el mensaje, un reporte de estado debe ser enviado de vuelta al SME origen (paso 3), en caso de haber sido solicitado por el SME origen durante el envío del mensaje.

Estos pasos se muestran en la Figura 5.2

5.6 Resumen

En la actualidad uno de los servicios con mayor popularidad ofertados por operadores móviles para dispositivos celulares es el envío y recepción de mensajes cortos, los conocidos SMS. Este servicio, que ha tenido un éxito importante en diferentes partes del mundo, calcula aproximadamente por cientos de millones los mensajes enviados cada mes en la mayoría de los paises del mundo. Los mensajes SMS son herederos directos de los mensajes envíados con los equipos localizadores de personas, los llamados *busca personas*, pero extendiendo su funcionalidad para permitir que desde cualquier dispositivo de telefonía celular o algunos dispositivos fijos, se pueda realizar un envío a otro equipo sin mediar

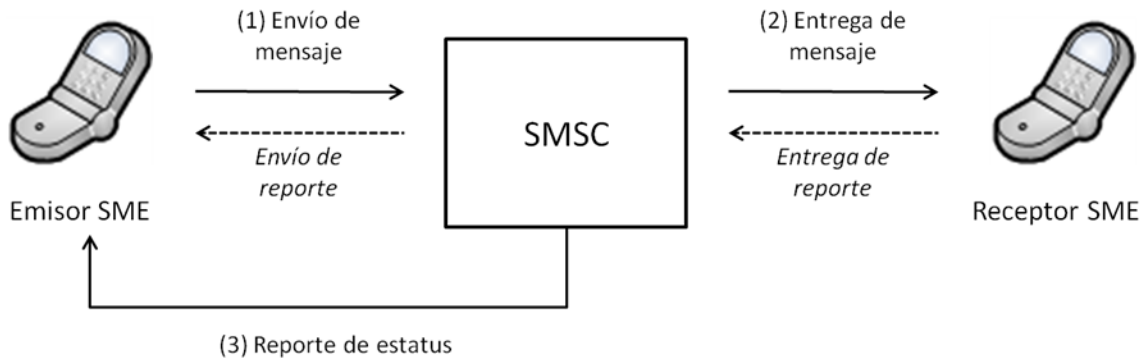

Figura 5.2: Intercambio de mensaje entre SME origen y SME destino

comunicación vocal. Las entidades más importantes que permiten el funcionamiento de los SMS son: la Entidad de Eensajes Cortos, el Centro de Servicio y el Email Gateway.

Capítulo 6

Gnokii

Gnokii es una herramienta de código abierto utilizada para el manejo teléfonos celulares conectados a un computador. Esta libreria puede ser descargada de forma gratuita a través de su página web www.gnokii.org/downloads.shtml. Originalmente fue desarrollada para funcionar en la plataforma Linux unicamente, hoy en día también se puede utilizar en otras plataformas como Windows y Mac OS. Sin embargo, algunas funciones no están disponibles en estas versiones.

Gnokii soporta muchas características diferentes, por ejemplo, envío y recepción de mensajes SMS, escritura, lectura de entradas en el libro de teléfonos y calendario, descarga de tonos, lectura del nivel de batería entre otras. Las características soportadas actualmente dependen del modelo de teléfono celular. En cuanto al envío y recepción de mensajes SMS, Gnokii trabaja bastante bien con la mayoría de los teléfonos Nokia, y además con otros teléfonos que soporten los comandos AT.

Gnokii es una herramienta de consola, pero es utilizado por varias programas GUI (Interfaz Gráfica de Usuario) para comunicarse con los teléfonos, entre los que se encuentran: Xgnokii, Gnocky, Gnome Phone Manager.

6.1 Historia

El proyecto original de Nokia 3810/3110/8110 surgió de las conversaciones entre François Dessart, Hugh Blemings y otros, su objetivo era desarrollar un reemplazo para el Cellular Data Suite(CDS) de Nokia, que se ejecuta bajo Linux. El proyecto comenzó a finales de octubre de 1998.

Un proyecto similar se inició por Staffan Ulfberg para proporcionar software para el Nokia 6110 y modelos similares de teléfono. El desarrollo de software no había empezado aún en el momento de la fusión de los proyectos.

Hacia finales de febrero de 1999, los dos proyectos fueron combinados para formar el actual proyecto Gnokii. La razón para esto fue evitar duplicar esfuerzos de codificación, y para tener solo una lista de correos para el intercambio de información acerca de los teléfonos.

Las cosas progresaron bien hasta la última parte de 2000, donde debido a otros compromisos por parte de los entonces principales autores, Pavel Janik y Hugh Blemings, las cosas se estancaron un poco. Afortunadamente, después de un largo período, algunos contribuyentes del proyecto intervienen para prestar su apoyo y seguir con el desarrollo de la misma manera que lo hacían anteriormente.

Los líderes actuales del proyecto son Pawel Kot Borbely Zoltan.

6.2 Características de Gnokii

Gnokii permite comunicarse con el teléfono a través de una conexión por cable serial, conexión USB (apoyo depende principalmente en el sistema operativo de apoyo), conexión de infrarrojos y conexión bluetooth.

Además ofrece una gran variedad de funcionalidad en las diferentes áreas donde el usuario manipula el teléfono móvil. Entre estas se encuentran:

- **SMS:** Se pueden enviar y recibir mensajes cortos de texto (SMS). Gnokii soporta los informes de entrega, mensajes con imágenes, mensajes concatenados y WAP (Protocolo de Aplicaciones Inalámbricas). Permite también enviar y recibir logos y tonos de llamada como mensajes de texto.
- **Agenda:** Gnokii ofrece la posibilidad de leer y escribir en la agenda. Las entradas en la agenda se pueden visualizar en formato legible por humanos o pueden ser exportados separadas por comas, en formato vCard, versión 3.0, o en formato LDIF. Se puede también importar datos de los mismos formatos.
- **Calendario:** Soporta tanto el calendario civil, como también las listas de tareas. Gnokii es capaz de exportar el calendario a archivos y puede importar archivos el mismo formato.
- **Manejo de llamadas:** Se pueden iniciar y responder llamadas a través de Gnokii.
- **Otras:** Entre otras características se pueden conseguir opciones de seguridad como introducir el PIN (Personal Identification Number), logos y tonos de llamadas, y muchas otras.

6.3 Instalación de Gnokii en Linux

Gnokii se encuentra incluido en algunas distribuciones de Linux, por ejemplo, Debian, Mandriva Linux y SUSE Linux en caso de estar utilizando alguna de las distribuciones anteriores, solo se necesita instalar Gnokii de acuerdo al procedimiento de instalación de paquete de la distribución de Linux, en caso contrario, se puede descargar la última versión del sitio web de Gnokii, luego compilarla e instalarla.

A continuación se describen los pasos a seguir para la instalación de Gnokii a partir del código descargado del sitio web de Gnokii:

1. Ir a http://www.gnokii.org/downloads.shtml para descargar el código de Gnokii.
2. Extraer el archivo descargado utilizando el comando *tar* Para extraer el archivo comprimido utilice el siguiente comando: tar -xzf gnokii-0.6.14.tar.gz.
3. Ir al directorio gnokii-0.6.14 que contenga los archivos extraídos.

   ```
   $> cd Gnokii-0.6.14
   ```

4. Ejecutar el script de configuración, dentro del directorio gnokii-0.6.14 con el siguiente comando:

   ```
   $> ./configure
   ```

El script de configuración soporta muchas opciones, dos comúnmente usadas son `--enable-security` y `--prefix`.

La opción de `--enable-security` es usada para permitir características relacionadas a la seguridad de Gnokii, si no se utilizan la opción de `--enable-security` en el script de configuración, no se podrán utilizar las opciones de seguridad después de la instalación.

```
$>./configure --enable-security
```

La opción `--prefix` es utilizada para especificar la ruta de la instalación. Si no se especifica una ruta con esta opción, la opción por defecto será `/usr/local`.

Para ver la lista completa de opciones del script de configuración y una breve descripción de ellos, se utiliza el siguiente comando:

```
$>./configure -help
```

Luego se procede a compilar el código fuente y finalmente proceder a instalarlo:

1. Compilar el código fuente: *$>make*
2. Si no se encuentra como usuario root, utilice el comando *su* para cambiar de usuario.
3. Comenzar el proceso de instalación `#>make install`
4. Instalación de la documentación de Gnokii `#>make install-docs`

6.4 Configuración de Gnokii

Una vez que se haya instalado Gnokii, se debe copiar el archivo `simple/gnokiirc` que se encuentra en la carpeta *Docs* al directorio *home* y nombrarlo como `.gnokiirc`. Usando un editor de texto se puede constatar que las opciones sean las correctas para el sistema que se esté utilizando. Se puede asumir que el teléfono que se está utilizando está soportado por Gnokii, sin embargo en algunos casos se va a tener funcionalidad limitada con Gnokii. Para la configuración existe una gran lista de los parámetros disponibles, pero en la mayoría de los casos solo se necesitarán especificar los parámetros modelo, conexión y puerto.

6.4.1 Parámetro Puerto (port)

El parámetro *port* especifica el puerto del computador donde el teléfono móvil se encuentra conectado. Se deben tener permisos de escritura y lectura en el puerto.

- Conexión Serial

  Si el teléfono móvil se encuentra conectado al puerto serial del computador, el valor del parámetro port puede ser `|/dev/ttySn|`, donde `n = 0,1,2...` Por ejemplo: `Port = /dev/ttyS0`

  Es importante destacar que el valor del parámetro port puede cambiar de acuerdo a la plataforma que esté siendo utilizada.

- Conexión Bluetooth

  Si el teléfono móvil se encuentra conectado al computador vía Bluetooth, el valor del parámetro port debe ser la dirección Bluetooth del teléfono móvil. Por ejemplo:

  ```
  port = 00:11:AB:B3:10:EB
  ```

- Conexión por infrarrojo (IrDA)

  Si el teléfono móvil se encuentra conectado al computador vía infrarrojo, el valor del parámetro port puede ser `/dev/ircommn`, donde `n = 0,1,2...` Por ejemplo `port=/dev/ircomm`.

- Conexión USB

  Si el teléfono móvil se encuentra conectado a través de un puerto USB del computador, el valor del parámetro port puede ser `/dev/ttyUSBn`, donde `n = 0,1,2...`. El valor puede ser también `/dev/ttyACM`, si se está utilizando el modo AT.

6.4.2 Parámetro Modelo (model)

El parámetro *model* especifica el modelo del teléfono móvil, Gnokii necesita esta información para decidir cual driver debe utilizar para comunicarse con el teléfono. Si se supone que el modelo del teléfono es Nokia 6021, se debe especificar *6021* en el parámetro model del archivo de configuración de Gnokii:

```
model = 6021
```

Si se desea conectar un teléfono con Gnokii a través del driver AT, se debe especificar el valor del parámetro model como AT. Por ejemplo, el teléfono Nokia 6021 soporta comandos AT. Así que se puede colocar 6021 o AT en el parámetro model.

```
model = AT
```

6.4.3 Parámetro Conexión (connection)

El parámetro *connection* especifica el tipo de conexión entre el computador y el teléfono móvil.

- Modo Comandos AT

  Si el modo de comandos AT está siendo utilizado, no importa como el teléfono se encuentre conectado con el computador, el valor del parámetro conexión debe ser *serial*.

  ```
  Model = AT
  Connection = serial
  ```

- Conexión Serial

  Si el teléfono móvil está conectado al computador a través del puerto serial, se debe asignar el valor *serial* al parámetro *connection* del archivo de configuración de Gnokii.

  ```
  connection = serial
  ```

- Conexión Bluetooth

  Si el teléfono móvil está conectado al computador vía Bluetooth, se le asigna el valor *bluetooth* al parámetro *connection*. `connection = bluetooth`

- Conexión Infrarrojo

  Si el teléfono móvil está conectado al computador vía infrarrojo se le asigna el valor de *infrarred* o *irda* al parámetro de configuración *connection*.

  ```
  connection = infrared
  connection = irda
  ```

- Conexión USB

  Si el teléfono móvil está conectado al computador a través de un puerto USB se le asigna el valor de *dku2libusb* al parámetro de configuración *connection* `connection = dku2libusb`.

6.5 Probar funcionamiento de Gnokii

Para probar la configuración que está siendo utilizada y verificar que esté funcionando correctamente, existe dos comandos en Gnokii generalmente utilizados para realizar pruebas, estos son -identify y -monitor.

El comando -identify le dice a Gnokii que retorne información básica sobre el teléfono móvil. Por ejemplo número de modelo, nombre del teléfono, entre otras.

```
$>Gnokii -identify
```

El comando -monitor imprime el estatus del teléfono, como el nivel de carga de la batería, número de memoria disponible y utilizada, entre otras. El comando -monitor se puede utilizar pasándole un parámetro opcional. Este puede tener el valor de *one*, que indica no actualizar el estatus del teléfono continuamente.

```
 $>gnokii --monitor once
```

O un entero especificando el intervalo en segundos para que se actualice el estatus:

```
$>sgnokii --monitor 10
```

El valor por defecto es actualizar el estatus cada segundo, sino se especifica ningún valor, se tomara el valor por defecto.

6.6 Ejemplos de uso de Gnokii

6.6.1 Desarrollar programas con Gnokii

Si solo se desea probar alguna función, se puede agregar el código que se desee en la función `foogle()` que se encuentra en el directorio `gnokii/gnokii.c`. Antes de usar alguna función que se comunique con el teléfono se debe llamar la función `businit()`, esto se conectará con el teléfono.

Incluso si la librería de Gnokii se encuentra instalada, se debe obtener el código fuente para utilizar los archivos de cabecera. En el código solo se necesita incluir el archivo:

```
#include <gnokii.h>
```

Al compilar, se debe enlazar el programa con la librería Gnokii

```
$>gcc -Wall -o foo foo.c ‘pkg-config --libs gnokii‘
```

6.7 Resumen

Gnokii es una herramienta que nos permite manejar los teléfonos celulares a través de una conexión al computador. Ofrece gran cantidad de funciones, por ejemplo enviar y recibir mensajes de texto, consultar la agenda telefónica, calendario, realizar llamadas, entre otras.

En principio Gnokii sólo fue elaborada para trabajar con los dispositivos celulares de Nokia, pero ha extendido el soporte a la gran mayoría de los dispositivos celulares GSM, utilizando para la comunicación entre el computador y el teléfono los comandos AT, que hoy en día son soportados por la mayoría de los teléfonos celulares.

Capítulo 7

Comandos AT

Los comandos AT son un conjunto de instrucciones que conforman un lenguaje para la comunicación entre un computador y un modem o teléfono celular logrando así, el control del mismo. El conjunto de comandos AT fue desarrollado en 1977 por Dennis Hayes, como una interfaz de comunicación entre el computador con un modem, para así poder configurarlo y proporcionarle instrucciones, como por ejemplo marcar un número de teléfono. Los comandos AT se denominan así por la abreviatura de attention y cada línea de comando empieza con "AT" o "at".

Aunque en general la principal finalidad de los comandos AT es la comunicación con módems, la telefonía móvil GSM también ha implementado como estándar este lenguaje para lograr la comunicación con sus dispositivos. Así, todos los teléfonos celulares GSM poseen un juego de comandos AT específico que sirven de interfaz para configurar y dar instrucciones a los terminales de dichos dispositivos. Este juego de comandos permite distintas acciones, como por ejemplo leer y escribir en la agenda de contactos, realizar llamadas de datos o de voz, enviar mensajes de texto (SMS), además de otras opciones.

Es importante destacar que la implementación de los comandos AT es responsabilidad del dispositivo GSM y no depende del medio de comunicación por el que serán enviados estos comandos, ya sea un medio infrarrojo, cable USB, Bluetooth, etc.

7.1 Funciones de los Comandos AT

A continuación se presentan algunas de las tareas que pueden ser realizadas utilizando comandos AT con un modem GSM o un teléfono celular [Home, 2008]:

- Obtener información básica del modem GSM o teléfono celular como el nombre del fabricante, el modelo del dispositivo, el número IMEI (Identidad Internacional del Equipo Móvil), y la versión del software del dispositivo.
- Obtener la información básica del subscriptor por ejemplo el número IMSI (Identidad Internacional del Subscriptor).
- Obtener el estatus actual del modem GSM o teléfono celular. Por ejemplo el estado de actividad del teléfono, la fuerza de la señal, el nivel de batería y el estado de carga.
- Establecer una conexión de datos o voz con un modem remoto.

- Enviar y recibir faxes.
- Enviar, leer, escribir o eliminar mensajes de texto y obtener notificaciones de nuevos mensajes SMS recibidos.
- Leer, escribir o buscar entradas en el directorio telefónico.
- Obtener o cambiar la configuración del modem GSM o teléfono celular. Por ejemplo cambiar la red GSM, dirección del SMS center y almacenamiento de mensajes SMS.
- Salvar y restaurar configuraciones del modem GSM o teléfono celular. Por ejemplo salvar y restaurar ajustes relacionados a la mensajería SMS como la dirección del SMS center.

Es importante destacar que los fabricantes de los teléfonos celulares generalmente no implementan todos los comandos AT, adicionalmente el comportamiento de los comandos implementados puede ser diferente que el definido en el estándar.

7.2 Comandos Básicos y Comandos Extendidos

Existen dos tipos de comandos AT: los comandos básicos y los comandos extendidos.

Los comandos básicos son comandos AT que no comienzan con "+", por ejemplo D (Dial o marcar), A (Answer o contestar), H (Hook control) y O (Return to online data state o retornar a estado online). Los comandos extendidos son comandos AT que comienzan con "+". Todos los comandos AT GSM son comandos extendidos, por ejemplo, +CMGS (Enviar mensajes SMS o Send SMS message), +CMSS (Enviar mensajes SMS almacenados o Send SMS message from storage), +CMGL (Listar mensajes SMS o List SMS messages) y +CMGR (Leer mensajes SMS o Read SMS messages).

7.3 Sintaxis General de los Comandos AT Extendidos

La sintaxis general de los comandos AT extendidos es sencilla. Debido a que todos los comandos de mensajería SMS son comandos AT extendidos, se expondrán a continuación las reglas de sintaxis para estos comandos:

Regla de sintaxis 1. Todas las líneas de comando deben comenzar con "AT" y terminar con un carácter de retorno de carro (se usará `<CR>` para representar el retorno de carro). Por ejemplo en un hyperterminal de Microsoft Windows, se puede presionar la tecla enter en el teclado para obtener el carácter de retorno de carro. *Ejemplo*: Para listar todos los mensajes SMS entrantes sin leer, almacenados en el dispositivo celular, se debe escribir "AT", luego el comando extendido "+CMGL", y finalmente el carácter de retorno de carro, de la siguiente manera:

```
AT+CMGL<CR>
```

Regla de sintaxis 2. Una línea de comando puede contener más de un comando AT. Solamente el primer comando AT debe estar precedido con "AT". Los comandos AT que sean colocados en una misma línea de comando deben estar separados con dos puntos ":" *Ejemplo*: Para listar todos los mensajes SMS entrantes sin leer, almacenados en el dispositivo celular y obtener el nombre del fabricante del dispositivo, se debe escribir "AT", seguido del comando extendido `"+CMGL"`, seguido de punto y coma, y el siguiente comando extendido `"+CGMI"`:

```
AT+CMGL; +CGMI<CR>
```

Ocurrirá un error si los dos comandos AT son precedidos con "AT" de esta manera:

```
AT+CMGL; AT+CGMI<CR>
```

Regla de sintaxis 3. Un string debe estar colocado entre comillas dobles "". *Ejemplo*: Para leer mensajes SMS desde el área de almacenamiento en modo texto, se necesita asignar el string "ALL" al comando extendido +CMGL, de esta forma:

```
AT+CMGL = "ALL"<CR>
```

Regla de sintaxis 4. Las respuestas informativas y los cogidos resultantes (incluyendo tanto los códigos resultantes finales, como los códigos resultantes no solicitados) siempre empiezan y terminan con un caracter de retorno de carro y un caracter de fin de línea *Ejemplo*: Luego de enviar la línea de comando "AT+CGMI<CR>" al dispositivo móvil, este debería retornar una respuesta similar a los siguiente:

```
<CR><LF>Nokia<CR><LF>
<CR><LF>OK<CR><LF>
```

La primera línea es la respuesta informativa del comando AT +CGMI y la segunda línea es el código del resultado final. <CR> y <LF> representan el retorno de carro y fin de línea respectivamente. El código resultado final "OK" marca el final de la respuesta. Indica que no se enviarán más datos desde el dispositivo móvil al computador. Cuando un programa terminal como el HyperTerminal de Microsoft Windows, ve un caracter de retorno de carro, mueve el cursor al comienzo de la línea actual. Cuando observa un carácter de fin de línea, mueve el cursor a la misma posición en la siguiente línea. Entonces la línea de comando "AT+CGMI<CR>" que se introdujo y la correspondiente respuesta serán desplegadas de la siguiente manera:

```
AT+CGMI
Nokia

OK
```

7.3.1 Sensibilidad a mayúsculas de los comandos AT

En la especificación SMS, todos los comandos AT se encuentran en letras mayúsculas. Sin embargo, muchos módems GSM y teléfonos celulares permiten escribir los comandos AT tanto en letras mayúsculas como en letras minúsculas. Por ejemplo en los Nokia 6021, los comandos AT no son sensibles a mayúsculas y las dos siguientes líneas son equivalentes:

```
AT+CMGL<CR>
at+cmgl<CR>
```

7.4 Códigos de resultado de los comandos AT

Los códigos de resultado son mensajes enviados desde un modem GSM o teléfono celular para dar información al usuario acerca de la ejecución de un comando AT y de la ocurrencia de un evento.

Existen dos tipos de códigos de resultado que son útiles al momento de utilizar comandos AT para mensajería SMS:

- Códigos finales resultantes
- Códigos resultantes no solicitados

7.4.1 Códigos finales resultantes

Un código final resultante marca el fin de una respuesta a un comando AT. Es una indicación de que el modem GSM o el teléfono celular han finalizado la ejecución de una línea de comando. Dos códigos finales resultantes son usados frecuentemente, estos son: OK y ERROR. Solamente uno de los dos será retornado para cada línea de comando, por lo tanto nunca se verán tanto OK como ERROR en la respuesta.

Código de resultado final OK

Indica que una línea de comando ha sido ejecutada de manera satisfactoria por el modem GSM o el dispositivo celular. Siempre comienza y termina con una caracter de retorno de carro y un caracter de fin de línea.

Código de resultado final ERROR

Indica que un error ha ocurrido cuando el modem GSM o el dispositivo celular trata de ejecutar una línea de comando. Luego de la ocurrencia del error, el modem GSM o el dispositivo celular no procesan el resto de los comandos AT contenidos en la línea de comando.

A continuación se exponen algunas causas comunes de error:

- La sintaxis de la línea de comando es incorrecta.
- El valor especificado para cierto parámetro es inválido.
- El nombre del comando AT está escrito de manera incorrecta.
- El modem GSM o dispositivo celular no soporta uno o más de los comandos AT, de los parámetros o del valor de estos en la línea de comando dada.

Al igual que el código de resultado OK, el código de resultado ERROR también comienza y termina siempre con un retorno de carro y un fin de línea.

Ejemplo: Suponga que quiere hacer que el dispositivo celular liste todos los mensajes SMS entrantes sin leer, almacenados en el área de almacenamiento de mensajes y obtener el nombre del fabricante del dispositivo móvil. Entonces se trata de escribir la siguiente línea de comando `"AT+CMGL;+CGMI<CR>"`, pero se comete el error de copiar "+CMFL" en lugar de "+CMGL". El dispositivo celular retornará el código resultante final de ERROR, como se muestra a continuación:

```
AT+CMFL;+CGMI<CR>
<CR><LF>ERROR<CR><LF>
```

Cuando ocurre el error el dispositivo celular detiene la ejecución de la línea de comando, es por esto que el segundo comando "+CGMI" no será procesado.

Si se escribe de manera incorrecta el segundo comando AT "+CGMI" en lugar del primero "+CMGL", el dispositivo celular enviará el resultado de la ejecución del primer comando "+CMGL" antes de enviar el código de resultado final, como se muestra a continuación:

```
AT+CMGL;+CGMU<CR>
<CR><LF>+CMGL: 1,"REC UNREAD","+85291234567",,"06/11/11,00:30:29+32"<CR><LF>
Welcome to our SMS tutorial.<CR><LF>
<CR><LF>ERROR<CR><LF>
```

7.4.2 Código de resultado final especifico para comandos AT de SMS

Los códigos de resultado finales OK y ERROR están disponibles para todos los comandos AT. A diferencia de OK y ERROR, el código de resultado final +CMS ERROR esta solo disponible para comandos AT de SMS. Notifica acerca de la ocurrencia de una falla en el servicio de mensajes.

Código de resultado final +CMS ERROR.

Este error es retornado cuando ha ocurrido una falla en el servicio de mensajes. Un código de error es provisto para que los programadores puedan verificar que ha causado la falla en el servicio. El código de error +CMS ERROR es específico para comandos AT de SMS. A continuación se muestran los comandos AT que pudieran generar como resultado el código de error +CMS ERROR:

- +CMGC. Nombre del comando en texto: Comando de envío (Send Command)
- +CMGD Nombre del comando en texto: Borrar mensaje (Delete Message)
- +CMGL Nombre del comando en texto: Listar mensajes (List Messages)
- +CMGR Nombre del comando en texto: Leer mensaje (Read Message)
- +CMGS Nombre del comando en texto: Enviar mensaje (Sending Message)
- +CMGW Nombre del comando en texto: Escribir mensaje en memoria (Write Message to Memory)
- +CMSS Nombre del comando en texto: Enviar mensaje almacenado (Send Message from Storage)
- +CPMS Nombre del comando en texto: Mensaje almacenado preferido (Preferred Message Storage)
- +CRES Nombre del comando en texto: Restaurar configuración (Restore Settings)
- +CSAS Nombre del comando en texto: Salvar configuración (Save Settings)
- +CSMS Nombre del comando en texto: Seleccionar servicio de mesnajes (Select Message Service)

La sintaxis de código de resultado final +CMS ERROR es:

```
<CR><LF>+CMS ERROR: error-code<CR><LF>
```

Así como los códigos OK y ERROR, el código +CMS ERROR siempre comienza y termina con el caracter de retorno de carro y un fin de línea. "error-code" es un entero que está asociado a cierto tipo de error. Una lista con algunos errores comunes y su significado puede ser encontrada en la tabla de códigos del error +CMS ERROR y sus significados se presenta en la Tabla 7.1.

A continuación se presentan algunas causas comunes para el error +CMS ERROR:

- Una tarjeta SIM no está presente en el dispositivo celular o modem GSM.
- La tarjeta SIM requiere un password, pero no se ha introducido.
- Un índice de memoria inválido es asignado a un comando AT.
- La memoria del dispositivo celular o modem GSM para almacenar mensajes está llena.
- La dirección del SMSC es desconocida o incorrecta.

A continuación se muestra un ejemplo del uso del código de error resultante +CMS ERROR. Suponiendo que existe solo un mensaje de texto almacenado en nuestro dispositivo celular y está almacenado en la ubicación de memoria con índice 1. Si por ejemplo, se introduce la línea de comando `"AT+CMGR=11"` (significa "leer el mensaje de texto en el índice de memoria 11"), el dispositivo celular retornará un error +CMS ERROR:

```
AT+CMGR=11<CR>
<CR><LF>+CMS ERROR: 321<CR><LF>
```

Tabla de códigos del error +CMS ERROR y sus significados.
La siguiente tabla lista algunos de los códigos de error +CMS y sus significados.

7.4.3 Códigos de error no solicitados de los Comandos AT

Los códigos de error no solicitados son mensajes enviados desde un modem GSM o un dispositivo celular para dar información acerca de la ocurrencia de un evento. Por ejemplo, se puede usar el comando AT +CNMI (Indicación de Nuevo Mensaje o New Message Indications) para solicitar al dispositivo celular o modem GSM que envíe el código resultado no solicitado "+CMTI" al computador cada vez que un nuevo mensaje SMS sea recibido del SMSC. A continuación se presentan algunos códigos de resultado no solicitados que están relacionados con mensajería SMS:

- +CDS Un dispositivo celular usa +CDS para redireccionar un nuevo reporte de estado SMS recibido al computador.
- +CDSI Un dispositivo celular usa +CDSI para notificar al computador que un nuevo reporte de estado SMS ha sido recibido y la localización de memoria donde está almacenado
- +CMT Un dispositivo celular usa +CMT para re direccionar un nuevo mensaje SMS recibido al computador.
- +CMTI Un dispositivo celular usa +CMTI para notificar al computador que un nuevo mensaje SMS ha sido recibido y la localización de memoria donde se encuentra almacenado.

Código +CMS ERROR	Significado
304	Uno o más parámetros asignados al comando AT son inválidos.
305	Uno o más parámetros asignados al comando AT son inválidos (para modo texto).
310	No hay una tarjeta SIM.
311	La tarjeta SIM requiere un PIN para operar. El comando AT +CPIN (nombre del comando en texto: Introduzca PIN o Enter PIN) puede ser usado para enviar el PIN a la tarjeta SIM.
313	Falla de tarjeta SIM.
314	La tarjeta SIM está ocupada.
315	La tarjeta SIM está dañada.
316	La tarjeta SIM requiere un PUK para operar. El comando AT +CPIN (nombre del comando en texto: Introduzca PIN o Enter PIN) puede ser usado para enviar el PUK a la tarjeta SIM.
320	Falla en el almacenamiento del mensaje en memoria.
321	El índice de almacenamiento en memoria asignado al comando AT es inválido.
322	El área de memoria para almacenamiento de mensajes está lleno.
330	La dirección del SMSC es desconocida.
331	No hay servicio de red disponible.
332	Ocurrió un timeout en la red.
500	Ocurrió un error desconocido.

Tabla 7.1: Algunos de los códigos de error +CMS y sus siginificados

7.5 Tipos de operaciones con comandos AT

Existen cuatro tipos de operaciones con comandos AT

- Operaciones Test. Una operación Test es usada para verificar si un comando AT específico es soportado por el modem GSM o dispositivo celular.
- Operaciones Set. Una operación Set en usada para cambiar los ajustes usados por el modem GSM o dispositivo celular para ciertas tareas.
- Operaciones de lectura. Una operación Read es usada para obtener ajustes actuales usados por el modem GSM o dispositivo celular para ciertas tareas.
- Operaciones de ejecución. Una operación de ejecución es usada para realizar una acción u obtener información/estatus acerca del modem GSM o dispositivo celular.

A continuación se describe la sintaxis de los comandos utilizados para la ejecución de estas operaciones.

1. Comando Test. Verifica si un determinado comando AT es soportado. Todos los comandos AT extendidos soportan la operación Test. La sintaxis es:

   ```
   comando=?
   ```

   Donde "comando" es un comando AT. Cuando un comando AT es usado con la sintaxis anterior es invocado el comando Test.

   Aquí se muestra un ejemplo. El comando +CGMI (Nombre del comando en texto: Request Manufacter identification) es usado para obtener el nombre del fabricante del modem GSM o dispositivo celular. Para probar si +CGMI está soportado, se puede hacer uso del comando test "+CGMI=?". La línea de comando que debe ser introducida es la siguiente:

   ```
   AT+CGMI=?
   ```

   Si el modem GSM o dispositivo celular soporta el comando AT +CGMI, el código de resultado "OK" se retornará, de esta forma:

   ```
   AT+CGMI=?
   OK
   ```

   Si el modem GSM o dispositivo celular no soporta el comando AT +CGMI, el código de resultado "ERROR" se retornará, de esta forma:

   ```
   AT+CGMI=?
   ERROR
   ```

   En el ejemplo anterior, el comando AT +CGMI no tiene ningún parámetro. Si el comando AT a ser probado tiene parámetros, los valores de los parámetros soportados por el modem GSM o teléfono celular pueden ser impresos adicionalmente. A continuación esta un ejemplo que ilustra el formato de la respuesta. +COMANDO es un comando AT ficticio que se usa en este caso como ejemplo y que tiene 4 parámetros.

   ```
   AT+COMANDO=?
   +COMANDO: (0,1),(0-10),(0,1,5-10),("GSM","UCS2")
   OK
   ```

   Los valores soportados por cada uno de los 4 parámetros están encerrados entre paréntesis. Las comas son usadas para delimitar los paréntesis y los valores dentro de los paréntesis. Un guion es usado para indicar un rango de valores. Los valores dentro de de los paréntesis pueden ser de tipo string.

   En el ejemplo anterior, la respuesta del comando test `"+COMANDO=?"` nos daba la siguiente información:

   0,1. El primer parámetro acepta 0 o 1 0-10. El Segundo parámetro acepta cualquier entero entre 0 y 10 0,1,5-10. El tercer parámetro acepta 0, 1 o cualquier entero entre 5 y 10 "GSM", "UCS2". El cuarto parámetro acepta o el string "GSM" o el string "UCS2".

Para algunos comandos AT, la operación Test no retorna los valores de los parámetros soportados. En su lugar, retorna los valores que se permiten en la información de respuesta del comando AT. Un ejemplo es el comando +CBC AT (nombre en texto del comando: Battery charge). El comando es usado para obtener información acerca del estatus de conexión y de la carga de la batería del dispositivo móvil. Dos valores son retornados en la información de respuesta del comando +CBC AT. El formato es:

+CBC: estatus de conexión, nivel de carga

Por ejemplo, si la batería se coloco en el dispositivo móvil sin cargador conectado y con un nivel de carga de 80 por ciento, el resultado de la ejecución del comando +CBC AT será:

```
AT+CBC
+CBC:0,80

OK
```

Si se ejecuta el comando test " +CBC=?", todos los valores soportados que esta permitido que aparezcan en el campo de estatus de conexión y en el campo de nivel de carga serán provistos. El resultado generalmente es:

```
AT+CBC=?
+CBC:(0,1),(0-100)

OK
```

"0,1" significa que el campo estatus de conexió,n en la información de respuesta del comando +CBC, puede tener 0 o 1 como valores, igualmente "0-100" significa que el campo de nivel de carga puede contener cualquier entero entre 0 y 100

2. Comando Set. Cambia ajustes usados para ciertas tareas.

   Una operación set cambia ajustes usados por el modem GSM o dispositivo celular para ciertas tareas. La sintaxis es:

```
comando=valor1,valor2,...valorN
```

   donde "comando" es el comando AT y valor 1 a valor N son los valores que se quieren establecer . Aquí se muestra un ejemplo, el comando AT +CSCA (nombre en texto del comando: Service Center Address) es usado para setear la dirección del SMSC (centro SMS) para el envío de mensajes SMS. Toma dos parámetros que especifican la dirección de SMSC y el "type of address". Para establecer la dirección del SMSC a por ejemplo +85291234567, se puede utilizar el siguiente comando:

```
AT+CSCA="+85291234567",145
Si el comando set se ejecuta correctamente, el codigo de resultado "OK"
   sera retornado
AT+CSCA="+85291234567",145
OK
```

Algunos comandos AT tienen parámetros opcionales. Se puede elegir no asignar valores a estos parámetros. Por ejemplo, el segundo parámetro del comando AT +CSCA es opcional. Si no se asigna ningún valor al segundo parámetro, el modem GSM o dispositivo celular usará el parámetro por defecto, que es 145 si la dirección del SMSC empieza con "+", por lo tanto esta línea de comando:

```
AT+CSCA="+85291234567"
```

Es equivalente a

```
AT+CSCA="+85291234567", 145
```

Usualmente los valores que se especifican con el comando set son colocados en memoria volátil. Si el modem GSM o dispositivo celular es apagado, los valores que se especificaron con el comando se perderán, cuando se encienda nuevamente, se cargarán todos los ajustes por defecto.

Para ajustes muy comunes, existen comandos AT para salvar/restaurar ajustes de/en memoria no volátil. Por ejemplo los comandos AT +CSAS (nombre en texto del comando: Save Settings) y +CRES (nombre en texto del comando: Restore Settings) pueden ser usados para salvar y restaurar ajustes relacionados con mensajería SMS como la dirección del SMS center.

3. Comando Read (de lectura).

   Una operación Read permite obtener los ajustes actuales usados por el modem GSM o teléfono celular para ciertas tareas, la sintaxis es:

   ```
   comando?
   ```

   Donde "comando" es un comando AT. La operación Read es soportada por todos los comandos AT que son capaces de la operación Set.

   Aquí se muestra un ejemplo que ilustra cómo usar un comando read. El comando AT +CSCA (nombre en texto del comando: Service Center Adress) es usado para setear la dirección del SMSC (centro SMS) para el envío de mensajes SMS. Necesita dos parámetros que especifican la dirección del SMSC y el "type of address". Suponiendo que se setea la dirección del SMSC a +85291234567 con el siguiente comando:

   ```
   AT+CSCA="+85291234567",145
   OK
   ```

   Si luego se ejecuta el comando read "+CSCA", el modem GSM o dispositivo celular retornará la dirección del SMSC y el "type of address" que se estableció en el paso anterior:

   ```
   AT+CSCA?
   +CSCA:"+85291234567",145
   OK
   ```

4. Comando de ejecución(Execution). Un comando de ejecución es usado para realizar una acción (por ejemplo, enviar o recibir un mensaje SMS) u obtener información/estatus acerca del modem

GSM o dispositivo celular (por ejemplo, obtener el nivel de carga de batería actual, fuerza de la señal en un momento dado). La sintaxis es:

```
comando=valor1,valor2,...valorN
```

donde "comando" es un comando AT y valor1 a valor N son los valores asignados. Si el comando AT no tiene ningún parámetro, la parte `"=valor1, valor2,...valorN"` deben ser omitidos. Cuando un comando AT es usado en la sintaxis anterior para realizar una operación de ejecución, es invocado un comando de ejecución.

Aquí se muestra un ejemplo que ilustra el uso de un comando de ejecución. El comando AT +CMSS(nombre en texto del comando: Send Message from Storage) puede ser usado para realizar una operación de ejecución, para en enviar un mensaje SMS almacenado en message storage, este tiene 3 parámetros. Ellos especifican el índice de la localización de memoria que almacena el mensaje SMS, el número telefónico de destino y el tipo de número telefónico respectivamente. Para enviar El mensaje SMS del índice 1 al número telefónico +85291234567, la siguiente línea de comando puede ser usada:

```
AT+CMSS=1,"+85291234567",145
```

Algunos comandos AT tienen parámetros opcionales. Se puede elegir no asignar valores a estos parámetros, por ejemplo, el tercer parámetro del comando AT +CMSS es opcional. Si no se asigna un valor al tercer parámetro, el modem GSM o dispositivo celular usará el valor del parámetro por defecto, que es 145 si el número telefónico de destino comienza con +. Entonces este comando:

```
AT+CMSS=1,"+85291234567"
```

Es equivalente a:

```
AT+CMSS=1,"+85291234567",145
```

A diferencia de los comandos set, los comandos de ejecución no almacenan los valores de los parámetros que le son asignados. Por ejemplo, si se envía el comando `"l AT+CMSS?"` a el modem GSM o dispositivo celular, el código resultado ERROR será retornado:

```
AT+CMSS?
ERROR
```

7.6 Resumen

Los comandos AT son instrucciones usadas para controlar tanto modems GSM como dispositivo celulares. AT es una abreviación de ATención. Cada línea de comandos empieza siempre con "AT" o "at" seguido del comando correspondiente.

En particular existen dos tipos de comandos AT, los comandos AT básicos y los comandos AT extendidos. Todos los comandos destinados a la mensajería SMS son comandos AT extendidos.

Queda claro que la implementación de los comandos AT corre a cuenta del dispositivo y no depende del canal de comunicación a través del cual estos comandos sean enviados, ya sea cable serial, canal Infrarrojos, Bluetooth, entre otros. De esta forma, es posible distinguir distintos teléfonos celulares del mercado que permiten la ejecución total del juego de comandos AT o sólo parcialmente.

Capítulo 8

Ruby

En este capítulo se describen las características principales de Ruby, un lenguaje de programación que cada día va ganando nuevos adeptos. Este lenguaje es considerado muy intuitivo, casi al nivel del lenguaje humano, por lo tanto es muy fácil de usar y aprender.

8.1 Introducción a Ruby

Ruby es un lenguaje de scripting orientado a objetos, que combina una sintaxis inspirada en Perl, con características de programación orientada a objetos similares a Smalltalk. Comparte también funcionalidades con otros lenguajes de programación como Python y Lisp. Ruby es un lenguaje de programación interpretado y su implementación oficial es distribuida bajo una licencia de software libre. Ruby puede ser ejecutado en distintas plataformas, incluyendo Linux, distintos sabores de UNIX, MS–DOS, Windows, BeOS y MacOS X.

Este lenguaje de programación fue creado por el japonés Yukihiro Matsumoto y presentado públicamente en el año 1995. El creador del lenguaje, ha dicho que Ruby está diseñado para la productividad y la diversión del desarrollador. Es importante destacar que el diseño del lenguaje se enfoca en el humano y no en la máquina.

Ruby [Ruby-Lang, 2008] es considerado un lenguaje flexible, ya que permite a sus usuarios alterarlo libremente. Las partes esenciales de Ruby pueden ser quitadas o redefinidas a placer, es por esto que se pueden agregar funcionalidades a partes ya existentes.

Este lenguaje de programación tiene un conjunto de funcionalidades entre las cuales se encuentran:

- Manejo de excepciones para facilitar el control de errores.
- Permite escribir extensiones en Lenguaje C de una manera fácil, brindando un elegante API para realizarlas. También está disponible una interfaz SWIG, que facilita el proceso de construcción de extensiones.
- Puede cargar bibliotecas de extensión dinámicamente, si lo permite el sistema operativo.
- Tiene manejo de hilos threading independiente del sistema operativo. De esta forma, tienes soporte multi –hilo en todas las plataformas en las que corre Ruby, sin importar si el sistema operativo lo soporta o no.

- Ruby es fácilmente portable: se desarrolla mayoritariamente en GNU/Linux, pero corre en varios tipos de UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, entre otros.

8.2 RubyGems

RubyGems [Berube, 2007] permite distribuir e instalar código Ruby en donde sea que se pueda instalar Ruby. Específicamente, RubyGems es un sistema de administración de paquetes para aplicaciones Ruby y librerías que permite instalar código Ruby, llamado gems.

8.2.1 ¿Por qué usar RubyGems?

Primeramente, RubyGems permite instalar software Ruby de una manera fácil [Fulton, 2006], por ejemplo para instalar un gem solo necesitamos colocar en una consola de Windows o Linux/Mac OS X el siguiente comando:

```
gem install <nombre del gem>
```

Por supuesto, para poder utilizar este comando se necesita tener instalado RubyGems.

8.2.2 Ventajas de los Gems

- Los gems proveen un mecanismo estándar de describir el software de Ruby y sus requerimientos: Permite definir un gemspec, donde se describe el software, este incluye nombre, versión, descripción y otros aspectos relacionados al gem.
- Proveen un repositorio central de software: Uno de los aspectos que provee RubyGems es que da acceso a RubyForge, un repositorio centralizado del software de Ruby. Sin RubyForge, se tendría que buscar el gem, luego descargarlo, para posteriormente instalarlo, además de realizar lo mismo con sus dependencias. En caso de utilizar RubyForge, RubyGems puede automáticamente localizar el software y sus dependencias.
- Permite distribuir gems usando un servidor: RubyGems viene con la tecnología necesario para configurar fácilmente un servidor de gems en una red local o en Internet.
- Maneja las dependencias de software: RubyGems se encarga de resolver las dependencias automáticamente. Esto significa que cuando se instala un gem, el sistema puede determinar automáticamente que otros gems son requeridos.
- Maneja múltiples versiones de software de manera inteligente: RubyGems puede almacenar múltiples versiones de un software y permite hacer peticiones de paquetes de software de acuerdo a una versión especifica, lo cual resulta muy conveniente si se requiere la última versión de un gem o una versión específica.
- Puede ser utilizado de manera transparente en lugar de las librerías regulares de Ruby: Si se instala un software particular desde RubyForge, de la manera tradicional, o a través de RubyGems, la manera de usar ese paquete es exactamente la misma: con la instrucción require 'paquete', por ejemplo.
- Permite utilizar la misma tecnología sobre cualquier Sistema Operativo: RubyGems se puede utilizar sobre cualquier plataforma donde se pueda instalar Ruby.

8.2.3 Instalación de RubyGems

El sistema de RubyGems [Baird, 2007] permite utilizar una gran cantidad de paquetes de software. En general, es sencillo instalar RubyGems. Por supuesto, antes de poder instalar RubyGems se necesita tener instalado en la maquina el interprete del lenguaje de programación Ruby y sus librerías.

Algunos sistemas operativos ya traen Ruby preinstalado, si no se conoce si el programa se encuentra instalado o no se puede ejecutar un comando por consola:

```
ruby - v
```

Si luego de ejecutar el comando se recibe un mensaje de error "Comando no encontrado", entonces Ruby no está instalado en la máquina, en caso de que se encuentre instalado el mensaje devuelto será:

```
Ruby 1.8.7 {2008-08-11} [i386 - mswin32]
```

Una vez instalado el interprete de Ruby en la máquina, se descarga el paquete RubyGems –x.y.z.zip, se descomprime y se ejecuta el comando ruby setup.rb. El paquete puede ser descargado desde "http://rubyforge.org/frs/?group_id=126".

Para comprobar que se instalo RubyGems de manera exitosa se puede ejecutar el comando gem —version, el cual desplegára información de la versión instalada.

8.2.4 Creación de un gem de Ruby

Para poder crear un gem se necesita saber que se requiere para hacerlo. Un gem es un archivo .gem, son parecidos a los archivos .zip o .tar. Este archivo contiene todo lo que el gem necesita para funcionar correctamente. Generalmente este archivo contiene subdirectorios: un directorio lib, donde se almacena todo el código fuente de el gem, un directorio test, donde se encuentran las pruebas realizadas, entre otros. El gem también contiene algunos datos acerca de quien la realizó, la versión, la fecha, página web del gem, entre otros. Estos datos se encuentran en un archivo llamado gemspec.

Diseño del paquete

La primera tarea para crear un gem [Thomas, 2006] es organizar el código en una estructura de directorio que tenga sentido. Las mismas reglas que se utilizarían para crear un típico archivo tar o zip aplican para la organización del paquete. Algunas convenciones generales son:

- Colocar el código fuente Ruby dentro de un subdirectorio llamado `lib/`.
- De ser apropiado para el proyecto, incluir un archivo en `lib/proyecto.rb` que contenga los comandos `"require"` necesarios para cargar la mayoría de las funcionalidades del proyecto.
- Siempre incluir un archivo README donde se incluya un resumen del proyecto, información de contacto con el autor, entre otras. Utilizar el formato RDoc para que pueda agregarlo a la documentación que se generara durante la instalación del gem. Recuerde incluir la licencia y los derechos de autor en este archivo, ya que muchos usuarios comerciales no utilizaran el paquete a menos que los términos de la licencia estén claros.
- Las pruebas deben ir en un directorio llamado `test/`.
- Cualquier archivo ejecutable debe ir en un subdirectorio llamado `bin/`.

- El código fuente de las extensiones de Ruby debe ir en `ext/`.

Este formato se encuentra ilustrado en la Figura 8.1.

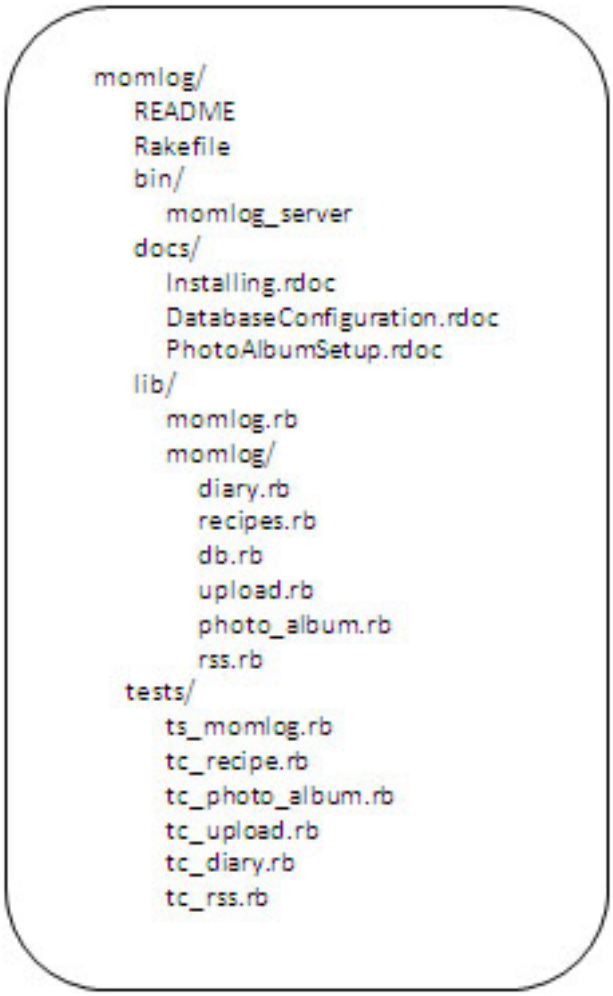

Figura 8.1: Estructura de un directorio para un gem

Especificación de un Gem

Un gemspec es una colección de metadata en Ruby que provee información clave acerca dea gem. Se pueden utilizar distintos mecanismos para la creación del gem, pero todos son conceptualmente lo mismo. A continuación se muestra un gemspec básico.

```
1 require 'rubygems'
2 SPEC = Gem::Specification.new do |s|
3    s.name = "MomLog"
4    s.version = "1.0.0"
5    s.author = "Jo Programmer"
6    s.email = "jo@joshost.com"
7    s.homepage = "http://www.joshost.com/MomLog"
```

```
8     s.platform = Gem::Platform::RUBY
9     s.summary = "An online Diary for families"
10    candidates = Dir.glob("{bin,docs,lib,tests}/**/*")
11    s.files = candidates.delete_if do |item|
12                  item.include?("CVS") || item.include?("rdoc")
13              end
14    s.require_path = "lib"
15    s.autorequire = "momlog"
16    s.test_file = "tests/ts_momlog.rb"
17    s.has_rdoc = true
18    s.extra_rdoc_files = ["README"]
19    s.add_dependency("BlueCloth", ">= 0.0.4")
20 end
```

La metadata del gem se coloca en un objeto de la clase *Gem::Especification*. Los primeros cinco atributos de la especificación anterior, dan información básica como nombre del gem, versión, nombre del autor, correo electrónico y página de inicio. En este ejemplo, el siguiente atributo es la plataforma en donde este gem puede ser ejecutada. En este caso el gem es solamente una librería de Ruby que no tiene ningún requerimiento en cuanto al sistema operativo, por esto se establece la plataforma como RUBY. Si el gem estuviese escrita para Windows solamente, por ejemplo, la plataforma seria WIN32.

El sumario del gem es una breve descripción que aparecerá cuando se ejecute un query para buscar un gem. El atributo *files* es un arreglo de rutas a los archivos que serán incluidos cuando el gem sea construido.

Agregando Pruebas y documentación

El atributo `test_file` contiene la ruta relativa de un archivo Ruby incluido en el gem que debería ser cargado como *Test::Unit*.

También se tienen dos atributos que controlan la producción de documentación local del gem. El atributo `has_rdoc` especifica que se han incluido comentarios RDoc al código. Es posible ejecutar RDoc sobre código totalmente descomentado, para proveer así vistas navegables de sus interfaces, pero obviamente esto es mucho menos valioso que ejecutar RDoc en un código bien comentado.

Agregando Dependencias

Para que un gem trabaje apropiadamente, los usuarios deberán tener instalado las dependencias que necesiten, que se encuentren especificadas en el *gemspec*, a través del método `add_dependency`.

Los argumentos del método `add_dependency` son idénticos a los de `require_gem`. Luego de generar el gem, al tratar de instalarla en un sistema que no contenga las dependencias obtendríamos como resultado lo siguiente:

```
gem install pkg/MomLog1.0.0.gem
Attempting local installation of 'pkg/MomLog1.0.0.gem'
/usr/local/lib/ruby/site_ruby/1.8/rubygems.rb:50:in `require_gem':
(LoadError)
Could not find RubyGem BlueCloth (>= 0.0.4)
```

Como se está realizando una instalación local del archivo, RubyGems no intentará resolver las dependencias. En lugar de ello, informa que se necesita de otro gem para completar la instalación, en este caso el gem que necesita es `Blue-Cloth`, a continuación se puede instalar el gem que se requiere para que se pueda completar con éxito la instalación. Si este gem `"MomLog"` se encuentra en los

repositorios centrales de RubyGems y se intentara instalar en un sistema donde no se encuentren instaladas las dependencias, se mostrará un mensaje que indique si se desea instalar las dependencias, como se muestra a continuación:

```
gem install MenuBuilder1.0.0.gem
Attempting local installation of 'MenuBuilder1.0.0.gem'
ruby extconf.rb inst MenuBuilder1.0.0.gem
creating Makefile
make
...
make install
...
Successfully installed MenuBuilder, version 1.0.0
```

El método `add_dependency` puede ser llamado múltiples veces en un solo gemspec, soportando tantas dependencias como sean necesarias.

Gems para Extensiones de Ruby

Hasta ahora, solo se ha visto como realizar gems que contengan código Ruby solamente. Sin embargo muchas librerías de Ruby son creadas como extensiones nativas. Existen dos maneras de empaquetar y distribuir este tipo de librerías como gems. Se puede distribuir el gem en formato de código, luego éste debe ser compilado momento de la instalación. Alternativamente, se puede precompilar la extensión y distribuir un gem por cada plataforma que se quiera soportar.

Para gems de código, RubyGems provee un atributo adicional en el *Gem::Specificaction* llamado *extensions*. Este atributo es un arreglo de rutas a archivos Ruby que generarán Makefiles.

Un ejemplo de gemspec utilizando extensiones se puede ver a continuación.

```
1  require 'rubygems'
2  spec = Gem::Specification.new do |s|
3        s.name = "MenuBuilder"
4        s.version = "1.0.0"
5        s.author = "Jo Programmer"
6        s.email = "jo@joshost.com"
7        s.homepage = "http://www.joshost.com/projects/MenuBuilder"
8        s.platform = Gem::Platform::RUBY
9        s.summary = "A Ruby wrapper for the MenuBuilder recipe database."
10       s.files = ["ext/main.c", "ext/extconf.rb"]
11       s.require_path = "."
12       s.autorequire = "MenuBuilder"
13       s.extensions = ["ext/extconf.rb"]
14 end
15 if $0 == __FILE__
16       Gem::manage_gems
17       Gem::Builder.new(spec).build
18 end
```

Se debe incluir los archivos fuentes en la especificación de la lista de archivos *file*, por lo que serán incluidos en el paquete de distribución del gem. Cuando un gem de código es instalada, RubyGems ejecuta cada programa de que se encuentre en el directorio *extensions* y luego ejecuta el Makefile resultante.

Distribuir este tipo de gems requiere que el consumidor del mismo tenga que trabajar en el desarrollo de herramientas. Como mínimo, necesitan algún tipo de programa make y un compilador. Particularmente para usuarios de Windows, estas herramientas pueden no estar presentes. Es por esto es que sería mejor distribuir gems precompiladas.

La creación de gems precompiladas es simple,se agregan los archivos de objetos compartidos compilados (DLLs para Windows) a la lista de archivos en *files* del gemspec, y asegurarse que estén en el atributo `require_path`. Como en el caso de los gems de solo código Ruby, el comando `require_gem` modificará el `$LOAD_PATH`, y se pueden acceder a los objetos compartidos a través de *require*.

Creación de archivo Gem

Los gemspec son archivos ejecutables como un programa Ruby. Invocándolo se creara un archivo .gem.

```
ruby momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog0.5.0.gem
```

También se puede utilizar el comando *gem build* para general el archivo gem.

```
gem build momlog.gemspec
Attempting to build gem spec 'momlog.gemspec'
Successfully built RubyGem
Name: MomLog
Version: 0.5.0
File: MomLog0.5.0.gem
```

Ahora que se tiene el archivo gem, se puede distribuir como cualquier otro paquete. Se puede colocar en un servidor FTP o en un sitio Web de donde se pueda descargar. Una vez que alguien descargue el archivo en su computadora local, pueden instalar el gem de la manera siguiente:

```
gem install MomLog0.5.0.gem
Attempting local installation of 'MomLog0.5.0.gem'
Successfully installed MomLog, version 0.5.0
```

8.3 Resumen

Ruby es un lenguaje dinámico y fácil de usar que nos brinda muchas ventajas a la hora de programar. También es flexible ya que partes de Ruby pueden ser removidas o redefinidas, y también se pueden agregar nuevas funcionalidades.

Ruby cuenta con un sistema de paquetes de software, llamado RubyGems que nos permite extender las funcionalidades básicas que el lenguaje nos ofrece. También se puede empaquetar paquetes de software propios de manera sencilla, y colocarlos en repositorios donde puedan ser descargados y posteriormente utilizados por otros usuarios.

Capítulo 9

Integración del lenguaje C/C++ con Ruby

Cuando se decide utilizar un lenguaje interpretado como Ruby, se intercambia velocidad por facilidad de uso. Es mucho más fácil desarrollar un programa en un lenguaje de alto nivel, sacrificando algo de la velocidad, que desarrollarlo en lenguajes de bajo nivel, como C y C++.

Es fácil extender Ruby con nuevas características escribiendo código en el mismo lenguaje, pero de vez en cuando, se hace necesaria la creacion de extensiones que permitan realizar tareas de bajo nivel. En este caso las posibilidades son interminables, en cuanto a las tareas que pueden realizarse. Por otro lado es muy sencillo escribir extensiones de C/C++ en Ruby, si se compara Ruby con otros lenguajes dinámicos.

9.1 Extensiones de C para Ruby

Para crear una extensión [Thomas, 2006] de manera sencilla se deben seguir los siguientes pasos:

1. Debe crearse un archivo que contenga las instrucciones necesarias para generar el Makefile correspondiente que permitirá crear la extensión. Dicho archivo generalmente es llamado extconf.rb, y debe colocarse en el mismo directorio donde se encuentren los archivos fuentes escritos en lenguaje C. Un ejemplo del contenido de extconf.rb se muestra a continuación:

   ```
   require 'mkmf'
   create_makefile("my_test")
   ```

2. Ejecutar el comando ruby extconf.rb para generar el Makefile tal como se muestra a continuación:

   ```
   ruby extconf.rb
   ```

3. Se utiliza *make* o *nmake* para construir la extensión.

El resultado final será un archivo objeto compartido (llamado extensión) que podrá ser usado desde Ruby.

9.1.1 Ejemplo de Extension de C en Ruby

A continuación se muestra un ejemplo sencillo del proceso a seguir para crear una extensión. Dicha extensión es solamente una prueba de los pasos que deben seguirse para construirla, no hace nada que no pueda hacerse utilizando solamente Ruby [Thomas, 2006]. En primer lugar se requiere una librería llamada *example*. Seguidamente se crea una instancia de la clase *Class* contenida en la librería, y por último se invoca un método de ésta.

```
require 'example'
e = Example::Class.new
e.print_string("Hello World\n")
# Hello World
```

Si la librería *example* estuviese escrita en lenguaje Ruby Su implementación sería la siguiente:

```
# example.rb
module Example
  class Class
    def print_string(s)
      print s
    end
  end
end
```

Si se implementa la misma funcionalidad a través de código C, el resultado sería el siguiente:

```
#include <ruby.h>
#include <stdio.h>

static VALUE rb_mExample;
static VALUE rb_cClass;

static VALUE
print_string(VALUE class, VALUE arg)
{
  printf("%s", RSTRING(arg)->ptr);
  return Qnil;
}
Void Init_example()
{
  rb_mExample = rb_define_module("Example");
  rb_cClass = rb_define_class_under(rb_mExample, "Class", rb_cObject);
  rb_define_method(rb_cClass, "print_string", print_string, 1);
}
```

Esta pequeña librería escrita en lenguaje C (*example.c*) define un módulo, una clase y un método utilizando las funcionalidades disponibles en *ruby.h*.

En el ejemplo mostrado anteriormente, se utiliza el método `rb_define_module` para crear el módulo *Example*, el método `rb_define_class` para definir la clase `Example::Class` (que hereda de la clase *Object*) y finalmente `rb_define_method` para definir el método `print_string` ubicado en la clase `Example::Class`.

Las variables VALUE que se observan en el código son el equivalente en Ruby a una referencia de C , y pueden apuntar a cualquier objeto de Ruby. Adicionalmente Ruby provee un conjunto de macros y funciones que permiten manipular los VALUE [Carlson and Richardson, 2006].

Cuando se coloca la instrucción `rb_define_class_under` se está especificando al intérprete de Ruby que defina una nueva subclase de Object. El archivo de cabecera *ruby.h* define variables similares para muchos otros módulos y clases de Ruby. Para generar la extensión también es necesario crear el archivo extconf.rb como se muestra a continuación:

```
# extconf.rb
require 'mkmf'

dir_config('example')
create_makefile('example')
```

Por último la extension es construida ejecutando el programa extconf.rb

```
$> ls
example.c extconf.rb

$> ruby extconf.rb
creating Makefile

$> make
gcc -fPIC -Wall -g -O2 -fPIC -I. -I/usr/lib/ruby/1.8/i486-linux
  -I/usr/lib/ruby/1.8/i486-linux -I. -c example

gcc -shared -L"/usr/lib" -o example.so example.o -lruby1.8
  -lpthread -ldl -lcrypt -lm -lc

$> ls
Makefile example.c example.o example.so extconf.rb
```

El archivo example.so será el resultado de los pasos realizados anteriormente, y podrá ser utilizado como cualquier otra librería de Ruby. También es posible la construcción de un .dll o .bundle aplicando los pasos en otros Sistemas Operativos.

9.2 Usar librerías escritas en C desde Ruby

Si desea utilizarse una librería particular para una aplicación en Ruby [Carlson and Richardson, 2006], pero esa librería esta implementada en lenguaje C, se puede escribir una extensión que encapsule la librería de C con clases y métodos de Ruby. La idea básica es encapsular las estructuras de C en objetos de Ruby.

Ruby provee macros que facilitan la creación de la extensión. Por ejemplo, `Data_Wrap_Struct` encapsula una estructura de datos de C en un objeto de Ruby y retorna un VALUE. La macro `Data_Get_Struct` toma un VALUE y devuelve un apuntador a la estructura de datos en C.

9.3 Utilizar librería escrita en C usando SWIG

Si se quiere utilizar una librería escrita en lenguaje C desde Ruby [Carlson and Richardson, 2006], pero no desea escribirse código C para poder manipularla, puede utilizarse SWIG [1] para generar la extensión http://www.swig.org

9.3.1 ¿Qué es SWIG?

SWIG es una herramienta de desarrollo [SWIG, 2008] que hace posible conectar programas escritos en C/C++ con otros lenguajes de alto nivel, principalmente lenguajes de "script" como Perl, Python, Tcl/Tk, Ruby, Guile, MzScheme y PHP; aunque entre los lenguajes soportados se incluyen otros como Java y Eiffel. Principalmente se utiliza para construir interfaces de usuario y como una herramienta para realizar pruebas sobre programas escritos en C/C++. Puede ser utilizado y distribuido libremente para uso comercial y privado.

9.3.2 Ejemplo de Extensión en SWIG

A continuación se muestra un ejemplo donde es creada una extensión utilizando Swig, la cual permite el acceso desde Ruby a funciones de una librería escrita en lenguaje C.

Primero es necesario construir un archivo de interfaz SWIG, libc.i:

```
1 %module libc
2
3 FILE *fopen(const char *, const char *);
4
5 int fread(void *, size_t, size_t, FILE *);
6 int fwrite(void *, size_t, size_t, FILE *);
7 int fclose(FILE *);
8
9 void *malloc(size_t);
```

Este archivo especifica el nombre de la extensión a crear como *libc*. Adicionalmente provee los prototipos para las funciones que se deseen utilizarse.

También es necesaria la creación del archivo *extconf.rb*, similar al creado en puntos anteriores:

```
1 # extconf.rb
2 require 'mkmf'
3 dir_config('tcl')
4 dir_config('libc')
5 create_makefile('libc')
```

Para generar la extensión de C, se procesa el archivo de interfaz utilizando el comando de swig y especificando el lenguaje destino. Seguidamente se ejecuta el programa extconf.rb para generar el Makefile y se ejecuta el comando make para compilar la extensión:

```
$> swig -ruby libc.i
$> ls
extconf.rb libc.i libc_wrap.c
```

[1] Simplified Wrapper and Interface Generator

```
$> ruby extconf.rb --with-tcl-include=/usr/include/tcl8.4
creating Makefile

$> make

$> ls
Makefile extconf.rb libc.i libc.so libc_wrap.c libc_wrap.o
```

Una vez que el módulo haya sido compilado, puede ser usado como cualquier otra extensión de Ruby. El siguiente código utiliza métodos de Ruby para llenar un archivo con datos aleatorios, luego utiliza la extensión de C para copiar el contenido del archivo en otro archivo:

```
1  random_data = ""
2  10000.times { random_data << rand(255) }
3  open('source.txt', 'w') { |f| f << random_data }
4
5  require 'libc'
6  f1 = Libc.fopen('source.txt', 'r')
7  f2 = Libc.fopen('dest.txt', 'w+')
8
9  buffer = Libc.malloc(1024)
10
11 nread = Libc.fread(buffer, 1, 1024, f1)
12 while nread > 0
13  Libc.fwrite(buffer, 1, nread, f2)
14  nread = Libc.fread(buffer, 1, 1024, f1)
15 end
16 Libc.fclose(f1)
17 Libc.fclose(f2)
18 # dest.txt contiene los mismos datos que source.txt.
19 random_data == open('dest.txt') { |f| f.read }
20 # => true
```

La gran ventaja de utilizar SWIG en lugar de escribir interfaces propias para una librería de C, es que efectivamente SWIG las escribe, ahorrando el trabajo de hacerlo manualmente.

Adicionalmente el archivo de interfaz necesita tener información sobre la extensión, es decir, no basta con colocar el prototipo de las funciones actuales, sino que se requiere como mínimo una línea `%module` que le indique a SWIG como llamar a la extensión que está generando. Dependiendo del código C, también se necesitará especificarle a SWIG como manejar los constructores en C que no son mapeados directamente con Ruby.

Existen dos maneras de crear un archivo de interfaz. La manera más simple es copiar el prototipo para las funciones en C, al archivo de interfaz de SWIG. Alternativamente, se puede utilizar el `%import nombre_archivo` para incluir el archivo de cabecera de C en un archivo de interfaz de SWIG.

9.4 Resumen

A pesar de las ventajas que ofrece el lenguaje Ruby, muchas veces puede existir la necesidad de utilizar funciones a bajo nivel, teniendo que recurrir a lenguajes como C/C++.

Afortunadamente, Ruby provee ciertos mecanismos que permiten utilizar programas realizados en lenguaje C desde programas hechos en Ruby. Uno de estos mecanismos es la creación de extensiones de C para Ruby, las cuales pueden generarse de manera manual o pueden ser creadas a través de SWIG.

Parte III

Marco Aplicativo

Capítulo 10

Adaptación del Proceso de Desarrollo XP y la Metodología AM

En este capítulo se detallan las prácticas y principios que se toman del proceso de desarrollo XP y la metodología AM para la creación de un gem para Ruby.

Se ha tomado este proceso de desarrollo y la metodología debido a que se desea que el desarrollo sea flexible, el diseño simple y abierto al cambio, donde el objetivo principal sea la realización del gem y no producir una extensa documentación que necesite muchos artefactos.

XP especifica un conjunto de prácticas y actividades que se utilizan a lo largo de la elaboración del gem, utilizando también los principios y prácticas de la Metodología AM.

10.1 Iteraciones

Siguiendo el proceso de desarrollo de XP se utilizan iteraciones. En cada una de ellas se planifican un conjunto de actividades y tareas que deben desarrollarse en un período de tiempo determinado. Las iteraciones son relativamente cortas, es por esto que su duración se estima en una semana aproximadamente.

10.2 Actividades

En cada iteración se aplican las cuatro actividades que propone XP, estas no necesariamente se tienen que realizar en un orden específico y se puede intercambiar de una a otra cuando se considere necesario. Estas actividades se utilizan como se detalla a continuación:

10.2.1 Planificación

En esta actividad se determina el alcance que se le da a cada iteración, definiendo que objetivos y metas deben ser realizadas y quien podría ser el encargado de realizarlas. También es posible definir modificaciones de las responsabilidades o tareas en una iteración, ya sea una iteración en curso o una iteración pendiente. De esta manera se evidencia la flexibilidad de esta metodología.

También se utilizan historias de usuario, donde se describen los requerimientos del sistema. Éstas sirven de base para las pruebas funcionales y ayudan a proporcionar una estimación del tiempo

necesario para el desarrollo. El formato para escribir las Historias de Usuario se presenta en la Tabla 10.1

Número	Fecha	Descripción

Tabla 10.1: Formato de historias de usuario

10.2.2 Diseño

Siguiendo el proceso de desarrollo de XP y la metodología AM, el diseño debe ser simple y sencillo. Se define una metáfora del sistema que muestra una visión global de lo que se quiere desarrollar. Adicionalmente, se pueden usar diagramas y modelos simples para entender cada uno de los módulos a implementar. En esta actividad se utiliza la refactorización para cambiar el diseño o código del sistema por otro que se adapte mejor a la solución.

10.2.3 Codificación

Para esta actividad se codifica en los lenguajes de programación C/C++ y Ruby manteniendo la consistencia y legibilidad del código, facilitando así, la comprensión para todos los involucrados en el desarrollo del sistema.

La integración del código se hace frecuentemente para que los miembros del equipo puedan trabajar siempre con la última versión. Se establecen estándares de codificación, manteniendo la consistencia y legibilidad del código, facilitando la comprensión para los involucrados en el desarrollo del sistema.

10.2.4 Pruebas

Para cada una de las iteraciones se realizan códigos de pruebas donde se puede constatar si la aplicación realiza la actividad esperada o no.

Durante cada iteración las historias de usuario seleccionadas en la planificación de iteraciones son sometidas a pruebas funcionales. Se especifican escenarios, verificando así cuando una historia de usuario ha sido implementada de manera correcta. Las historias de usuario no se consideran completadas hasta que no hayan pasado satisfactoriamente las pruebas funcionales.

Cabe destacar que a lo largo del presente trabajo, es posible que algunas iteraciones no implementen las cuatro actividades antes descritas, debido a que no son requeridas.

10.3 Planificación de iteraciones

El desarrollo del sistema se divide en doce iteraciones. El objetivo de cada iteración es obtener una versión del sistema que incluya la implementación de las historias de usuario planteadas en la planificación del proceso de desarrollo XP. Los resultados de las iteraciones son evaluados por los desarrolladores hasta que se cumplan todos los objetivos y metas planteados.

Cada iteración a grandes rasgos incluye lo siguiente:

- Iteración 0: Definición de los requerimientos generales del sistema tomando como base la propuesta elaborada.

- Iteración 1: Desarrollo del módulo de envío de mensajes de texto en lenguaje C, con la implementacion de sus distintas opciones.
- Iteración 2: Creación del wrapper que permita utilizar el módulo de envio desde el lenguaje de programación Ruby en el sistema operativo Linux.
- Iteración 3: Creación del wrapper que permita utilizar el módulo de envio desde el lenguaje de programación Ruby en el sistema operativo Windows.
- Iteración 4 y 5: Generación de la capa de conexión y administración de conexión del middleware SMS.
- Iteración 6: Implementación de la capa de envío del middleware SMS en el lenguaje de programación Ruby.
- Iteración 7: Elaboración del módulo de manejo de errores y excepciones.
- Iteración 8: Incorporación de comando AT+CMGL en la librería de código abierto Gnokii.
- Iteración 9: Creación del módulo de recepción SMS en el lenguaje de programación C.
- Iteración 10: Creación de la capa de recepción del middleware SMS.
- Iteración 11: Creación de un gem de Ruby que contenga la extensión realizada y el middleware SMS.

Capítulo 11

Desarrollo

En este capítulo se exponen las actividades de desarrollo e implementación de cada iteración, siguiendo el esquema propuesto en el Capítulo 10, documentando las actividades de diseño e implementación y las estrategias de pruebas.

11.1 Iteración 0

Del 27-Abr-2009 al 01-May-2009

En esta iteración se definen los requerimientos generales del sistema tomando como base la propuesta elaborada. Adicionalmente, los componentes de dicho sistema son representados a través de un esquema gráfico (metáfora), a partir del cual se desarrollan las demás iteraciones.

11.1.1 Planificación

El sistema debe proveer un mecanismo simple para el envío y recepción de mensajes de texto, utilizando cualquier dispositivo celular GSM. Este sistema está destinado a ser usado por programadores Ruby que requieran incorporar este tipo de tecnología en sus aplicaciones.

El sistema (gem) debe proveer las funcionalidades básicas para el envío y recepción de mensajes de texto a través de un Middleware SMS.

Requerimientos funcionales

Los requerimientos funcionales describen el comportamiento, funciones o servicios del sistema, y realizan los objetivos, tareas o actividades solicitadas por el usuario. Es por esto que para el sistema se definieron los siguientes requerimientos funcionales:

- Enviar mensajes de texto a través distintos dispositivos celulares de manera concurrente.
- Recibir mensajes de texto a través distintos dispositivos celulares de manera concurrente.
- Implementar un mecanismo que permita administrar las conexiones disponibles.

Requerimientos no funcionales

Los requerimientos no funcionales abarcan aspectos del sistema visibles para el usuario, que no están relacionados de forma directa con el comportamiento funcional del sistema. Se definieron los siguientes requerimientos no funcionales:

- Ofrecer soporte multi-plataforma.
- Colocar el gem bajo una licencia pública.
- Proveer control de errores y excepciones que proporcione robustez al sistema.

11.1.2 Diseño

Para cumplir con los requerimientos establecidos se realizó un esquema general del sistema donde se identifican cada uno de los componentes que lo conforman y la interacción entre cada uno de ellos.

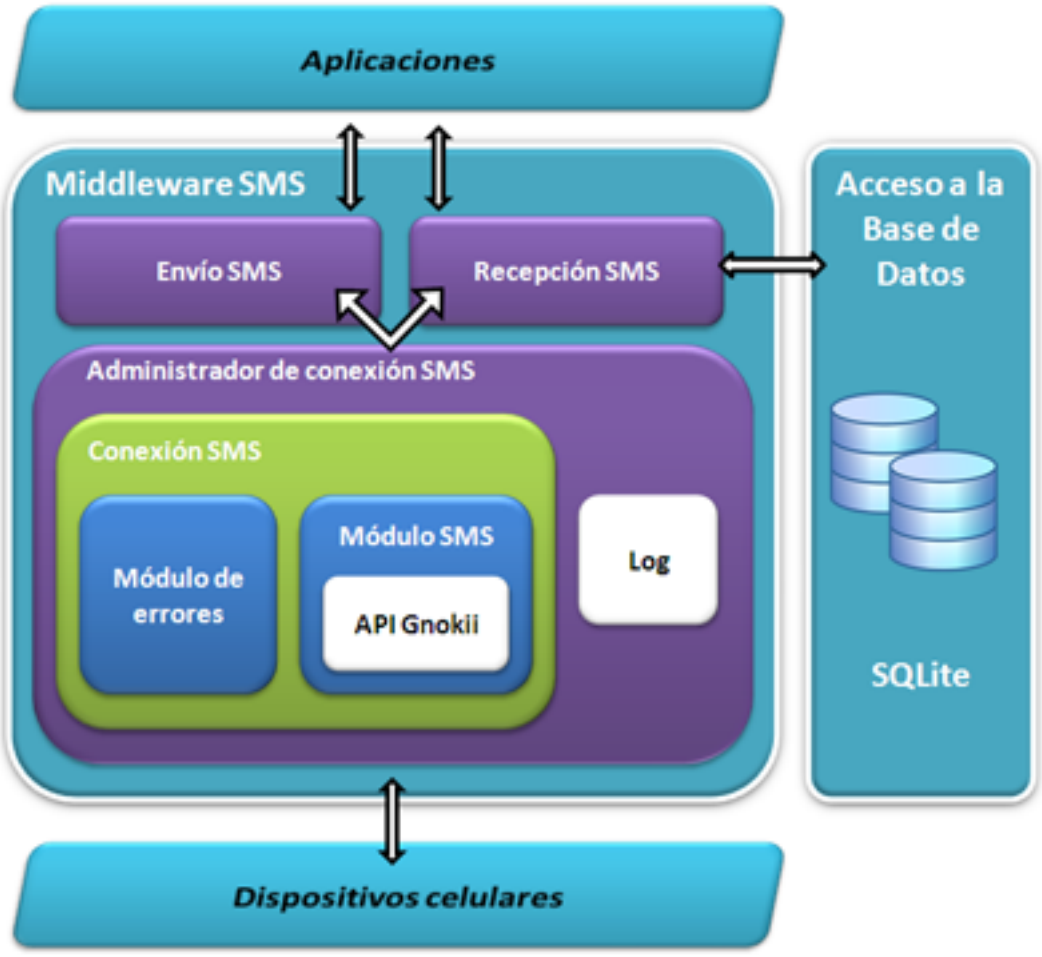

Figura 11.1: Metáfora del Sistema

En la Figura 11.1 se muestran las distintas capas que componen el middleware SMS encapsulado en el gem de Ruby. A continuación se describen las funcionalidades de cada una de las capas:

El administrador de conexiones se encarga de mantener un control de las conexiones existentes, así como de controlar el flujo y la carga de envíos.

La capa de conexión SMS se encarga de mantener el control de una conexión específica con algún dispositivo y mantener la información del dispositivo como la señal de estado, fabricante, estado de la batería, IMEI, entre otros.

La capa de envío SMS se encarga de ejecutar todas las acciones referentes al envío de mensajes de texto, implementando distintas opciones.

La capa de recepción SMS se encarga de realizar la lectura de los mensajes de texto que se encuentren en el dispositivo celular. Esta capa utiliza la base de datos para almacenar los mensajes leídos del dispositivo celular.

El módulo SMS se encarga de realizar la comunicación con el dispositivo de hardware para realizar la conexión y poder ejecutar los comandos AT, para esto se utiliza la librería de código abierto Gnokii.

El módulo de errores se encarga de manejar las distintas excepciones y errores que pueden ocurrir durante la ejecución del programa, utilizando un log para tener registrados los errores ocurridos.

Por otro lado se elaboró un diagrama de clases que permitiera identificar y relacionar las clases a desarrollar en el middleware SMS. Dicho diagrama es mostrado en la Figura 11.2

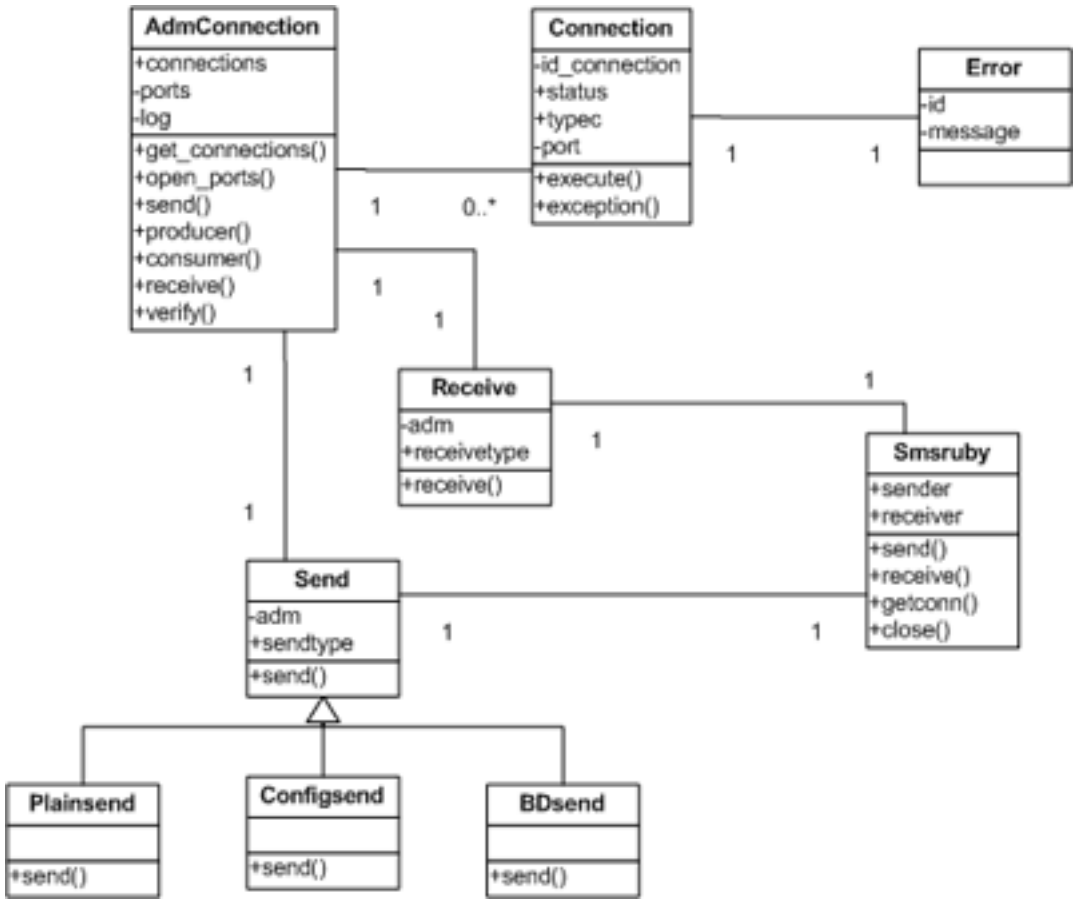

Figura 11.2: Diagrama de clases middleware SMS

11.1.3 Aspectos determinantes del Sistema

Se especifican algunos aspectos relevantes que caracterizarán el sistema desarrollado a lo largo de las iteraciones. El aspecto más importante es su estructuración en forma de gem de Ruby, lo que provee una manera estándar para distribuir el sistema a través de RubyGem. Entre otros aspectos importantes se pueden mencionar:

- El gem estará disponible para ser instalado desde los repositorios de RubyForge, Github o algún otro repositorio de manera gratuita y bajo licencia libre.
- El gem podrá ser instalado y desinstalado fácilmente a través de RubyGems.
- El gem podrá ser utilizado, modificado y mejorado.
- El gem soportará cualquier dispositivo celular que utilice la tecnología GSM y que pueda ser conectado a un computador.

Un aspecto importante que debe mencionarse es la portabilidad ofrecida por el gem. Esto implica la construcción de una extensión de Ruby para cada plataforma a la que desee portarse. Además de ofrecer funcionalidades básicas de envío y recepción de mensajes de texto a través de un middleware SMS.

11.2 Iteración 1

Del 4-May-2009 al 8-May-2009

En esta iteración se desarrolla el módulo para el envío de mensajes de texto utilizando la librería de código abierto Gnokii.

11.2.1 Planificación

Para esta iteración se establece la meta *Crear módulo para el envío de mensajes de texto en lenguaje C, con la implementación de sus distintas opciones.* En la Tabla 11.1 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	04 - 05 - 2009	Dar soporte a las distintas opciones de envío.
2	04 - 05 - 2009	Permitir el envío de mensajes largos (mayores a 160 caracteres).

Tabla 11.1: Historia de usuario - Iteración 1

11.2.2 Diseño

En esta etapa se definen y se implementan las opciones de envío más resaltantes, para así, ofrecer al usuario final la posibilidad de configurar dichas opciones dependiendo de sus necesidades.

Las opciones contempladas son las siguientes:

- Establecer el periodo de validez de un mensaje.
- Solicitar informe de entrega.
- Enviar mensajes con una longitud mayor a 160 caracteres.

En algunos casos el envío de mensajes de texto largos o concatenados, presenta problemas debido al manejo interno de las locaciones de memoria en el dispositivo celular. Por esta razón para algunos dispositivos y tipos de memoria los mensajes largos se dividen en tantos mensajes menores a 160 caracteres como sean necesario, indicando en cada uno de ellos el número del mensaje con respecto al total de divisiones.

A cada mensaje se añade un encabezado con la frase *"[smsRuby]"*

11.2.3 Codificación

La función para el envío de mensajes es definida de la siguiente manera:

```
gn_error send_sms(char *number, char *msj, char *smsc, int report, char *
    validity).
```

Donde el valor retornado es un entero perteneciente al enumerado `gn_error`. Este valor determina cual es el error específico (si lo hubo) que se obtiene como respuesta de la función de envío.

El argumento char `*msj` representa una cadena de caracteres que se envía como un mensaje de texto a un dispositivo celular identificado con el número char `*number`, especificado en el primer argumento.

El siguiente paso se basa en escribir el código en lenguaje C necesario para dar soporte a las opciones indicadas en los argumentos formales. Los siguientes fragmentos de código muestran la implementación realizada para dar soporte a las opciones mencionadas:

```
if(smsc != NULL){
   sprintf_s(sms.smsc.number, sizeof(sms.smsc.number), "%s", smsc);
   if (sms.smsc.number[0] == '+')
      sms.smsc.type = GN_GSM_NUMBER_International;
   else
      sms.smsc.type = GN_GSM_NUMBER_Unknown;
   }
else{
   data.message_center = calloc(1, sizeof(gn_sms_message_center));
   data.message_center->id = 1;
   if (gn_sm_functions(GN_OP_GetSMSCenter, &data, state) == GN_ERR_NONE) {
      sprintf_s(sms.smsc.number, sizeof(sms.smsc.number), "%s", data.
         message_center->smsc.number);
      sms.smsc.type = data.message_center->smsc.type;
   } else {
      printf("Cannot read the SMSC number from your phone.\n");
   }
      free(data.message_center);
}
```

El parámetro char `*smsc` es utilizado para establecer de manera explícita el numero de smsc a través del cual se envía el mensaje de texto. Si no es indicado ningún smsc en dicho argumento, se utiliza una función propia de la librería Gnokii para obtener el valor configurado en el dispositivo celular que ha sido conectado. En cualquiera de los casos se coloca dicho valor en el campo apropiado (*sms.smsc.number*) para que sea reconocido una vez que se proceda al envío del mensaje.

```
if(report == 1){
        sms.delivery_report = 1;
}
```

Si el valor del argumento report es 1 se solicita explícitamente un informe de entrega para el mensaje a enviar. El valor por defecto establecido es 0, y es asignado a través de la siguiente función: `gn_sms_default_submit(&sms);`

```
if (validity!=NULL){
  if (validity < 0)
    return GN_ERR_IVALIDVALIDITY;
  else
    sms.validity = atoi(validity);
}
```

Cuando el valor del argumento validity es mayor que 0, se establece el tiempo (expresado en minutos) durante el cual el mensaje a enviar tiene validez y no es descartado. En el caso de que el valor pasado como argumento sea negativo un error es arrojado indicando una opción inválida para el argumento validity. El valor constante de error `GN_ERR_IVALIDVALIDITY` debió ser agregado a la librería Gnokii puesto que ninguno de los existentes se adecuaba a la situación. Por esta razón, en este punto la librería Gnokii es recompilada.

El siguiente fragmento de código muestra como es implementada la división del mensaje en caso de que el envío de mensajes largos a través de comandos AT no esté soportado en el dispositivo celular o se trate de un mensaje largo. Adicionalmente se muestra como es colocado el encabezado que identifica a la aplicación desarrollada.

```
if ((input_len % 144) != 0)
  npart=(input_len/144)+1;
else
  npart=input_len/144;

for(i=0;i<npart;i++){
  printf("entre en el ciclo de npart pos %d\n ",i);
  if (((i+1)*144-1)>=input_len)
    end = input_len-1;
  else
    end=(i+1)*144-1;
  sprintf(temp,"[smsRuby] %d/%d ",i+1,npart);
  printf("antes de substring temp es %s ",temp);
  temp=substring(i*144,end,msj,temp,1600);
  printf("luego de la asignacion la data es :wwi %s ",temp);
  error=send_internal(number, temp, smsc, report, validity, state);
}
```

11.2.4 Pruebas

Las pruebas se hicieron bajo el siguiente ambiente:

Sistema Operativo

Debian 5.0 GNU/Linux kernel 2.6.26-1-686

Configuración de Hardware

CPU: Intel Core 2 Duo 2.2 Ghz

Memoria RAM: 2GB

Disco Duro: 250GB

Software

- GNU make
- GNU gcc
- GNU gdb

Dispositivos utilizados

- Motorola RZR V3
- Samsung SGH-E370

Pruebas Funcionales

En la Tabla 11.2 se encuentran las distintas pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Opciones de envío	Verificar el correcto funcionamiento del módulo de envío de mensajes con las distintas opciones implementadas.	Envío correcto de mensajes de texto, cumpliendo con las opciones utilizadas.
2	Mensajes largos	Constatar el correcto envío de mensajes largos a través de los distintos dispositivos celulares.	Envío correcto de mensajes largos o concatenados desde distintos dispositivos celulares.

Tabla 11.2: Pruebas funcionales - Iteración 1

- **Prueba 1 - Opciones de Envío**

Para verificar el correcto funcionamiento de las opciones de envío, se realizaron diferentes pruebas cambiando los valores de las distintas opciones, a continuacion se presentan las pruebas realizadas:

```
gn_error send_sms(char *number, char *msj, char *smsc, int report, char *
    validity).
```

1. En el primer caso solo se le asigna valores a los campos de número de teléfono destino, el mensaje a enviar y reporte. A este último se le asigna valor 0, lo que significa que no se desea recibir reporte de entrega. Se espera que el mensaje sea enviado correctamente, recibido por el dispositivo celular correspondiente y no se reciba reporte de entrega.

   - **number**: <número de teléfono>
   - **msj**: <Mensaje menor de 160 caracteres>
   - **smsc**: Null
   - **report**: 0
   - **validity**: Null
   - **Retorno**: 0
   - **Resultado**: El mensaje fue enviado exitosamente, como al tiempo de validez no se le asigno ningún valor, se le asigna el valor por defecto que establece el SMSC correspondiente. El valor del SMSC se toma del teléfono a través de comandos AT. No se recibe reporte de entrega.

2. En este caso se le asigna valores a los campos de número de teléfono destino, mensaje a enviar, número de SMSC y reporte. Se espera que el mensaje sea enviado correctamente, recibido por el dispositivo celular correspondiente y no se reciba reporte de entrega.

- **number**: <número de teléfono>
- **msj**: <Mensaje menor de 160 caracteres>
- **smsc**: <Numero del SMSC>
- **report**: 0
- **validity**: Null
- **Retorno**: 0
- **Resultado**: El mensaje fue enviado exitosamente. Como se asigna el numero del SMSC no necesita buscarlo en el sistema, sino que utiliza el valor que se pasa por parámetro. El tiempo de validez toma el valor por defecto que establece el SMSC. No se recibe reporte de entrega.

3. En este caso se le asigna valores a los campos de número de teléfono destino, mensaje a enviar y reporte. A este último se le asigna valor 1, lo que significa que se desea recibir reporte de entrega. Se espera que el mensaje sea enviado correctamente, recibido por el dispositivo celular correspondiente y se reciba reporte de entrega.

   - **number**: <número de teléfono>
   - **msj**: <Mensaje menor de 160 caracteres>
   - **smsc**: Null
   - **report**: 1
   - **validity**: Null
   - **Retorno**: 0
   - **Resultado**: El mensaje fue enviado exitosamente. En este caso se recibe un mensaje de confirmación que indica que el mensaje fue entregado al destino exitosamente.

4. En este caso se le asigna valores a los campos de número de teléfono destino, mensaje a enviar, número de SMSC, validez y reporte. Al campo validez se le asigna el valor de 1, esto significa la validez de este mensaje es de un 1. Se espera que el mensaje no sea enviado correctamente debido al tiempo de validez asignado.

   - **number**: <número de teléfono>
   - **msj**: <Mensaje menor de 160 caracteres>
   - **smsc**: Null
   - **report**: 0
   - **validity**: '1'
   - **Retorno**: 0
   - **Resultado**: El tiempo de validez del mensaje se estableció a 1 minuto, por esto para realizar la prueba se apagó el teléfono destino durante un intervalo de tiempo mayor a 1 minuto. Cuando se encendió nuevamente no se recibió el mensaje porque efectivamente éste fue descartado, dado que su tiempo de validez era solo de 1 minuto.

- **Prueba 2 - Mensajes largos**

1. En este caso se envía un mensaje con una longitud mayor a 160 caracteres. Sin hacer uso del mecanismo implementado para el envío de mensajes largos.

   - **number**: <número de teléfono>
   - **msj**: <Mensaje mayor de 160 caracteres>
   - **smsc**: Null
   - **report**: 0
   - **validity**: Null
   - **Retorno**: 0
   - **Resultado**: En este caso se presentaron dos situaciones diferentes:
     - Con el dispositivo celular Samsung SGH-E370 se pudo enviar el mensaje de manera satisfactoria.
     - Con el dispositivo celular Motorola V3 no se pudo enviar el mensaje, debido a un conflicto presentado al utilizar algunos comandos AT, por esto se decidió que al enviar un mensaje mayor a 160 caracteres, se enviarán tantos mensajes distintos como sean necesarios.

2. En este caso se envia un mensaje con una longitud mayor a 160 caracteres, utilizando el mecanismo implementado para el envío de mensajes largos.

   - **number**: <número de teléfono>
   - **msj**: <Mensaje mayor de 160 caracteres>
   - **smsc**: Null
   - **report**: 0
   - **validity**: Null
   - **Retorno**: 0
   - **Resultado**: El mensaje fue enviado exitosamente a través de los distintos dispositivos.

11.3 Iteración 2

Del 11-May-2009 al 15-May-2009

Se desarrolla el wrapper del módulo de envío para ser utilizado desde Ruby con el sistema operativo Linux.

11.3.1 Planificación

Para esta iteración se establece la meta *Creación del wrapper que permita utilizar el módulo de envío SendSMS desde Ruby en Linux.* En la Tabla 11.3 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	11 - 05 - 2009	Crear un objeto compartido que pueda ser utilizado desde Ruby para el envío de mensajes.

Tabla 11.3: Historia de usuario - Iteración 2

11.3.2 Diseño

Debido a que la librería de Gnokii está escrita en Lenguaje C, es implementado un mecanismo que permite utilizarla a través de Ruby para posteriormente construir el gem. Para esto se utiliza Swig, una herramienta que permite construir un wrapper tomando funciones escritas en lenguaje C para que puedan ser usadas por lenguajes interpretados como Ruby. Seguidamente se utiliza el módulo mkmf para generar un archivo Makefile que permite compilar el wrapper y generar la extensión (archivo .so para el caso de Linux).

La Figura 11.3 muestra los insumos y procesos involucrados para generar la extensión.

11.3.3 Codificación

Una vez creado el módulo para el envío de mensajes de texto escrito en lenguaje C, se procede a crear el wrapper utilizando Swig, permitiendo el uso de dicho módulo desde Ruby. El proceso de creación del wrapper se describe a continuación.

1. Se crea un archivo de interfaz (archivo .i) que contiene los prototipos de las funciones que desean utilizarse.

```
1  /* SendSMS.i */
2   %module SendSMS
3   %{
4   extern struct gn_statemachine *state;
5   extern void busterminate(void);
6   extern void businit(void);
7   extern int send_sms(char *number, char *msj, char *smsc, int report,
        char *validity);
8  %}
9
10  extern struct gn_statemachine *state;
11  extern void busterminate(void);
```

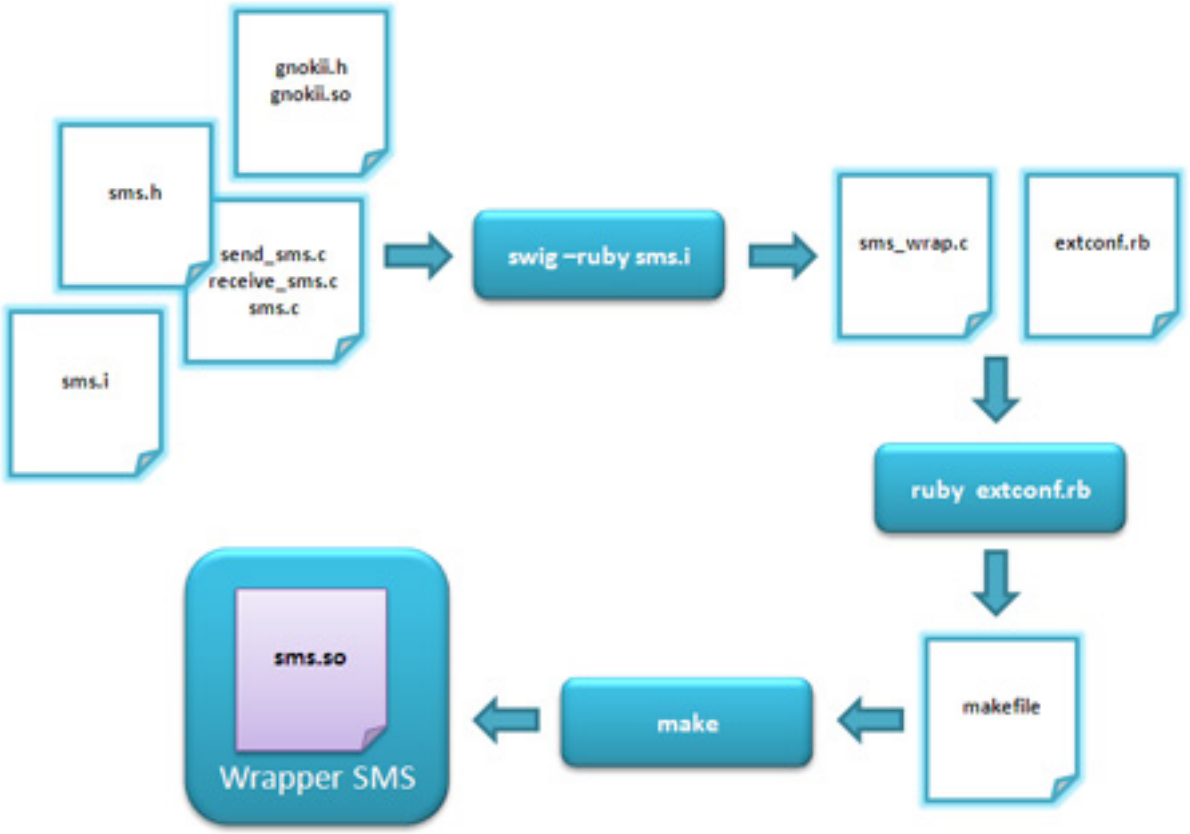

Figura 11.3: Procesos e insumos necesarios para generar la extensión

```
12  extern void businit(void);
13  extern int send_sms(char *number, char *msj, char *smsc, int report,
        char *validity);
```

2. Para generar el wrapper se utiliza el comando swig –ruby SendSMS.i.

3. Seguidamente se crea un archivo llamado *extconf.rb*.

```
1 require 'mkmf'
2 $LIBS += " -lgnokii"
3 create_makefile('SendSMS')
```

Las instrucciones colocadas en el archivo fueron las siguientes:

- `require 'mkmf'` para agregar el módulo mkmf
- `$LIBS += "-lgnokii"` para enlazar la librería Gnokii
- `create_makefile` para poder generar el archivo Makefile.

Al ejecutar el archivo *extconf.rb* a través del comando Ruby *extconf.rb* el archivo Makefile es creado.

4. Una vez obtenido el archivo Makefile, se ejecuta el comando make para compilar el wrapper y generar el archivo objeto compartido (en este caso un .so), el cual puede ser usado desde Ruby.

   Para incluir la librería en Ruby se coloca la instrucción `require 'SendSMS'`, permitiendo así la utilización de las funciones que se encuentran en el objeto compartido (extensión).

11.3.4 Pruebas

Las pruebas se realizaron utilizando el mismo entorno que en la iteracion 1 agregando distinto software:

Software

- Ruby 1.8.6
- Swig 1.3.39 para Linux.

Las pruebas se realizaron con los mismos dispositivos usados en la Iteración 1.

Pruebas Funcionales

En la Tabla 11.4 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Objeto compartido - Linux	Comprobar la creación correcta del wrapper que permita utilizar el módulo de envío en Linux.	Envío correcto de mensajes de texto haciendo uso del wrapper construido.

Tabla 11.4: Pruebas funcionales - Iteración 2

- **Prueba 1 - Objeto compartido - Linux**

Para comprobar el funcionamiento correcto de la extensión creada, se realizó un programa en Ruby que utiliza las funciones del módulo de envío contenidas en la extensión.

El siguiente código muestra el programa utilizado para las pruebas:

```
1 require 'SendSMS'
2 SendSMS.businit()
3 i = SendSMS.send_sms(param1, param2, param3, param4, param5)
4 SendSMS.busterminate()
```

- **Resultado**: Luego de ejecutar el programa anterior, se puede enviar correctamente mensajes de texto a través de los distintos dispositivos celulares utilizados.

11.4 Iteración 3

Del 18-May-2009 al 22-May-2009

Se desarrolla el wrapper del módulo de envío para ser utilizado desde Ruby con el sistema operativo Windows.

11.4.1 Planificación

Para esta iteración se establece La meta *Creación del wrapper que permita utilizar el módulo de envío SendSMS desde Ruby en Windows*. En la Tabla 11.5 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	18 - 05 - 2009	Crear un objeto compartido que pueda ser utilizado desde Ruby para el envío de mensajes de texto.

Tabla 11.5: Historia de usuario - Iteración 3

11.4.2 Diseño

Al igual que en Linux, se utiliza la herramienta *Swig* para crear el wrapper, así como el módulo mkmf de Ruby para generar el archivo Makefile. En este caso la herramienta nmake de Windows es utilizada para compilar el Makefile y generar el objeto compartido utilizado en Ruby.

En caso de haber instalado Ruby utilizando el *One-Click Installer* (compilado con Visual C++ 6.0) deben realizarse algunas modificaciones en el archivo de configuración de Ruby siempre y cuando no se cuente con el compilador *Visual C++ 6.0*. Esto se debe a que dicho archivo especifica explícitamente que el uso de cualquier otro compilador para crear los objetos compartidos requeridos, generará un error de version unmatch. En caso de no cambiar el archivo de configuración deben siempre crearse los objetos compartidos utilizando el compilador *Visual C++ 6.0*.

11.4.3 Codificación

Para cumplir con los objetivos establecidos en esta iteración se siguieron los siguientes pasos:

1. Se crea el archivo de interfaz que utiliza Swig para generar el wrapper, definiendo allí las variables y funciones que pueden ser utilizadas desde Ruby.

```
1  /* SendSMS.i */
2   %module SendSMS
3   %{
4   extern struct gn_statemachine *state;
5   extern void busterminate(void);
6   extern void businit(void);
7  extern int send_sms(char *number, char *msj, char *smsc, int report,
       char *validity);
8  %}
9
10  extern struct gn_statemachine *state;
11  extern void busterminate(void);
```

```
12  extern void businit(void);
13  extern int send_sms(char *number, char *msj, char *smsc, int report,
        char *validity);
```

2. Se genera el wrapper utilizando la herramienta Swig, a través del comando *swig –ruby SendSMS.i*.

3. Se modifica el archivo de configuración de Ruby ubicado en la ruta

   `c:\ruby\lib\ruby\1.8\i386-mswin32\config.h`

```
1 #if _MSC_VER != 1200
2 #error MSC version unmatch
3 #endif
```

   El valor 1200 del `_MSC_VER` indica que el compilador utilizado es *Visual C++ 6.0*. Por esta razón, al utilizar un compilador distinto, se procede a eliminar las líneas anteriores del archivo *config.h*, evitando así, la ocurrencia del error versión unmatch en caso de que el compilador encontrado en el sistema operativo sea distinto a *Visual C++ 6.0*.

4. Seguidamente se crea el archivo extconf.rb, donde se incluye el módulo *mkmf* para la generación del archivo *Makefile*.

```
1 require 'mkmf'
2 $libs = append_library($libs,"gnokii")
3 create_makefile('SendSMS')
```

   Con la instrucción `$libs = append_library($libs,"gnokii")` se incluye la librería *Gnokii* para poder resolver las dependencias al momento de compilación. Adicionalmente se utiliza el comando `create_makefile` para generar el archivo *Makefile*.

5. Las librerías pertenecientes a Visual `C++` y algunas otras librerías no pueden ser usadas en una aplicación `C/C++` sin un archivo .manifest que enlace la aplicación a estas librerías. Este archivo manifest no es más que una meta data adicional que se utiliza para que las aplicaciones `C/C++` puedan hacer referencia a las librerías correspondientes, esta meta data es embebida en cada ejecutable como un recurso adicional. Si una aplicación `C/C++` que dependa de alguna de estas librerias no utiliza un manifest, el intentar cargarla genera un error.

   Esto permite adicionalmente que al momento de cargar una librería se cargue la apropiada para la aplicación aunque existan diversas versiones de la misma.

   Para incluir el comando necesario para la creación del archivo manifest se coloca en el archivo

   `C:\Ruby\lib\Ruby\1.8\mkmf.rb`

   lo siguiente:

```
1 1338: #link_so = LINK_SO.gsub(/^/, "\t")
2 1339: #mfile.print link_so, "\n\n"
3 1340: link_so = LINK_SO.gsub(/^/, "\t")
4 1341: mfile.print link_so, "\n"
```

```
1342: mfile.print "\tmt.exe -manifest $(DLLIB).manifest -outputresource:
   $(DLLIB);2\n" if $mswin
1343: mfile.print "\n\n"
```

Lo que se hace es modificar el módulo mkmf de Ruby directamente, para no tener que generar el archivo manifest cada vez que se genere el Makefile.

6. Se ejecuta el comando Ruby *extconf.rb* para generar el *Makefile*.

7. Una vez generado el *Makefile*, se ejecuta el comando *nmake*. Con esto se crea un objeto compartido que puede ser utilizado desde Ruby en Windows.

11.4.4 Pruebas

Las pruebas se hicieron bajo el siguiente ambiente:

Sistema Operativo

Windows XP

Configuración de Hardware

CPU: Intel Core 2 Duo 2.2 Ghz

Memoria RAM: 2GB

Disco Duro: 250GB

Software

- Ruby 1.8.6
- Swig 1.3.39 para Windows.
- Microsoft Visual C++ 2005
- Microsoft Windows SDK for Windows Server 2008.
- Entorno Cygwin

Las pruebas se realizaron con los mismos dispositivos usados en la Iteración 1.

Pruebas Funcionales

En la Tabla 11.6 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

- **Prueba 1 - Objeto compartido - Windows**

Para comprobar el correcto funcionamiento de la extensión creada, se realizó un programa en Ruby que utiliza las funciones del módulo de envío contenidas en la extensión.

El siguiente código muestra el programa utilizado para las pruebas:

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Objeto compartido - Windows	Comprobar la creación correcta del wrapper que permita utilizar el módulo de envío en Windows.	Envío correcto de mensajes de texto haciendo uso del wrapper construido.

Tabla 11.6: Pruebas funcionales - Iteración 3

```
require 'SendSMS'
SendSMS.businit()
i = SendSMS.send_sms(param1, param2, param3, param4, param5)
SendSMS.busterminate()
```

- **Resultado**: Luego de ejecutar el programa anterior, se puede enviar correctamente mensajes de texto a través de los distintos dispositivos celulares utilizados.

11.5 Iteración 4

Del 25-May-2009 al 29-May-2009

En esta iteración se agregan funcionalidades al wrapper, se implementa parcialmente la capa de conexión, la capa de administración de conexiones del middleware SMS y la comunicación entre ambas.

11.5.1 Planificación

Para esta iteración se establece la meta *Generación de la capa de conexión y administración de conexión del middleware SMS*. En la Tabla 11.7 se encuentran las distintas historias de usuario que corresponden a esta iteración.

Número	Fecha	Descripción
1	25 - 05 - 2009	Reestructurar y modificar el wrapper para permitir a los usuarios finales la utilización de varias conexiones y dispositivos celulares.
1	25 - 05 - 2009	Elaborar la capa de conexión e implementación de comunicación entre la extensión generada en iteraciones anteriores y la capa de conexión, para dar soporte a la funcionalidad de envío.
3	25 - 05 - 2009	Crear la capa de administración de conexiones y manejo de múltiples instancias de conexión en dicha capa.

Tabla 11.7: Historia de usuario - Iteración 4

11.5.2 Diseño

Una vez creada la extensión, la idea es elaborar la capa de conexión que permita mantener un canal de comunicación con un dispositivo celular específico, así como llevar el control del estado de dicha conexión en cualquier momento. Adicionalmente deben poder manejarse varias conexiones; éstas son controladas y administradas por el administrador de conexiones.

Para poder dar soporte a múltiples conexiones simultáneas se deben realizar varias modificaciones en el wrapper. El diseño planteado hasta la iteración anterior suponía la presencia de dos módulos, uno de envío y uno de recepción, estructurado tal como representa la Figura 11.4.

El diseño fue modificado y fue estructurado tal y como representa la Figura 11.5 :

11.5.3 Codificación

La siguiente estructura definida en el archivo sms.h permite mantener las referencias a las diferentes conexiones iniciadas en el middleware.

```
1 typedef struct {
2   int activeconnect;
3   struct gn_statemachine *connections[CONNECTIONS_MAX_LENGTH];
4 }gn_connection;
```

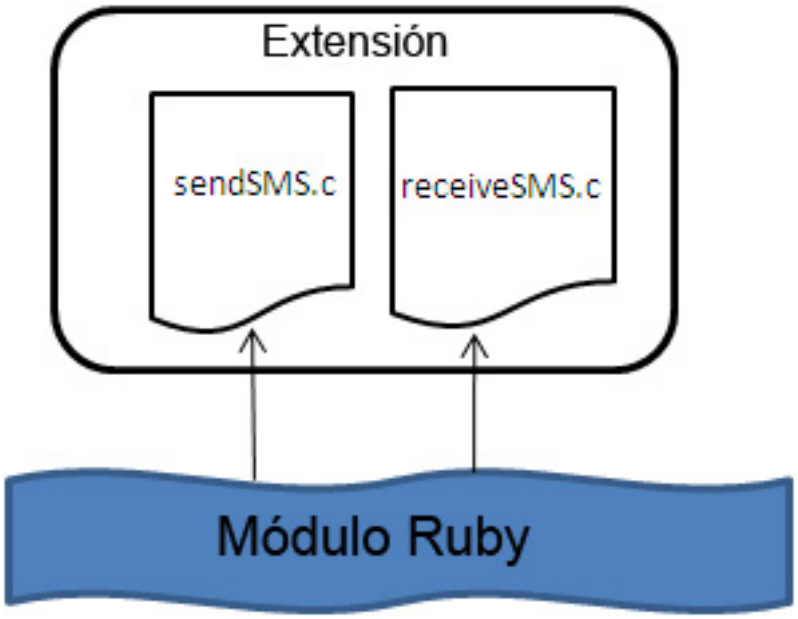

Figura 11.4: Diseño de extensión antes de modificación para soporte de múltiples conexiones

Cada posición del arreglo connections almacena un apuntador a la estructura `gn_statemachine` que contiene toda la información necesaria asociada a una conexión. Con una sola referencia a esta estructura (caso que se ve reflejado en la extensión generada en iteraciones anteriores) se hace imposible el manejo de múltiples conexiones.

La siguiente línea colocada en el archivo *sms.h* define el número máximo de conexiones que pueden ser inicializadas a través del middleware SMS. Se debe verificar siempre que el número máximo de conexiones permitidas no ha sido excedido.

```
#define CONNECTIONS_MAX_LENGTH 10
```

El primer paso de la iteración consiste en la modificación de la función de inicio *businit()*, de manera que se pueda dar soporte para el manejo de múltiples conexiones.

El siguiente segmento de código en lenguaje C muestra como se almacena la referencia a la nueva conexión en *conn.connections*, donde *conn* es una variable de tipo `gn_connection` y *connections* el vector que almacena las referencias a la estructura `gn_statemachine`.

```
1  int businit(char *nameconn, char *archpath) {
2  int free=0;
3  int idconn=0;
4  while(!free && idconn < CONNECTIONS_MAX_LENGTH){
5          if(conn.connections[idconn]==NULL){
6                  conn.activeconnect++;
7                  free=1;
8          }
9          else
10                 idconn++;
11 }
12 if(free){
```

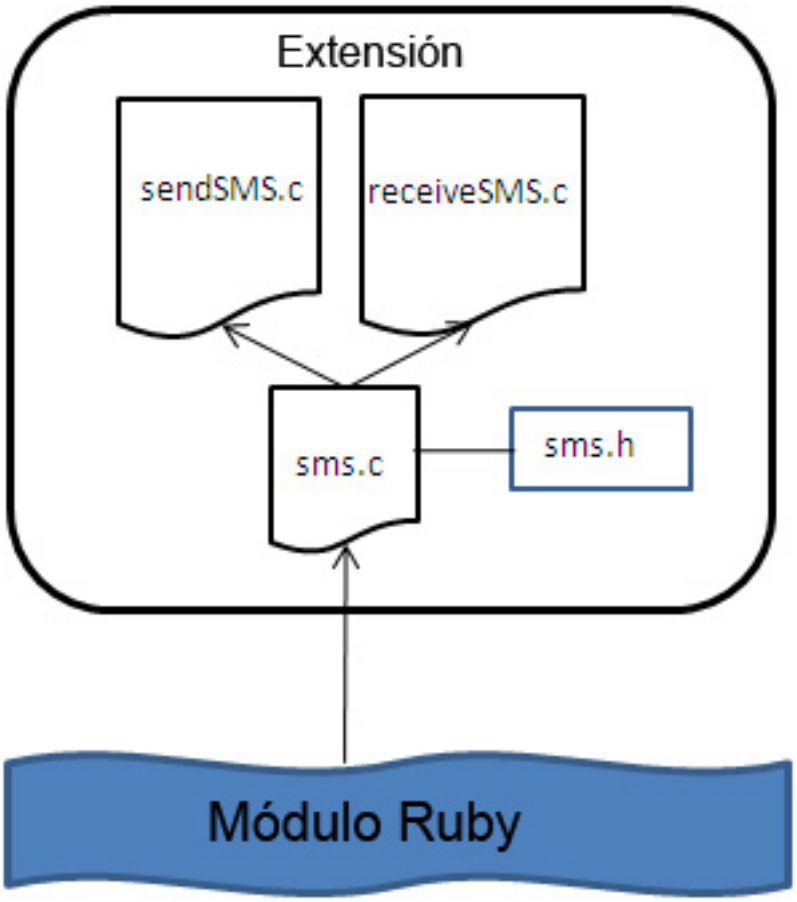

Figura 11.5: Diseño de extensión luego de modificación para soporte de múltiples conexiones

```
13 gn_lib_phoneprofile_load_from_file(archpath,nameconn,&conn.connections[
       idconn]);
14  }
15 }
```

Debido a las modificaciones realizadas para dar soporte a múltiples conexiones de manera simultánea, es necesario colocar dos definiciones de la función de envío; una en *sms.c* y una en *SendSMS.c*. La primera recibe el id de la conexión a utilizar y es la que interactúa con la capa de conexión del middleware SMS:

```
int send_sms(char *number, char *msj, char *smsc, int report, char *validity
    , int idconn);
```

La segunda (invocada por la primera) recibe una referencia a la estructura `gn_statemachine` encontrada a partir del id de conexión proporcionado:

```
int send_smsi(char *number, char *msj, char *smsc, int report, char *
    validity,struct gn_statemachine *state);
```

El archivo de interfaz (*sms.i*), necesario para la generación del wrapper, se reescribe quedando reflejado su código de la siguiente manera:

```
/* sms.i */
 %module sms
 %{
 #include "gnokii.h"
 #include "smsr.h"
 %}
 %include<smsr.h>
```

El siguiente paso es la creación de la capa de conexión del middleware SMS. Se crea la clase Connection en Ruby para representar dicha capa de la siguiente manera:

```
class Connection

attr_reader :id_connection

  def initialize(name)
     begin
        error=businit(name, '/home/urs/.gnokiirc')
        error <= 0 ? @id_connection=error.to_i*-1 : "error"
     end
  end
  def close
     busterminate(@id_connection)
  end
  def execute(hsh)
     cmd = hsh[:type]
        case cmd
           when /send/
              send_sms(hsh[:dst], hsh[:msj], hsh[:smsc], hsh[:report], hsh[:
                 validity], @id_connection)
     end
  end
end
```

La función execute permite ejecutar una función específica de la extensión ya generada. En este punto se encuentra contemplada únicamente la función de envío de mensajes.

La función close representa la terminación de una conexión particular y en la función initialize se invoca a la función *businit()* descrita anteriormente.

Posteriormente se procede a la creación de la capa de administración de conexión, permitiendo a través de ésta, la instanciación de múltiples conexiones. Para implementar las distintas funcionalidades requeridas en este punto, se define la clase *AdmConnection* de la siguiente manera:

```
class AdmConnection
  @@connections = Hash.new
  def initialize
     @array = ['/dev/ttyACM0'] #ports
     open=false
     save_file(@array)
     for i in 0..@array.size-1
```

```
8           open = open_conn("telf"+i.to_s) || open
9         end
10      def save_file (array)
11        ...
12      end
13      def open_conn(name)
14        n = @@connections.size
15        con = Connection.new(name)
16        @@connections.merge!({n => con})
17        ...
18        open
19      end
20    end
```

A través de la función initialize, se invoca al procedimiento utilizado para salvaguardar el archivo de configuración necesario para la comunicación con el dispositivo celular. Adicionalmente se crean tantas conexiones como puertos haya especificados en *@array* utilizando la función `open_conn(name)`.

Para determinar por cual conexión se enviaría el mensaje se coloca una función *random* dentro de send que obtuviera aleatoriamente el número de la conexión por la que será enviado el mensaje.

11.5.4 Pruebas

Las pruebas de esta iteración se hicieron bajo el mismo ambiente descrito en la iteración 2 y 3. Adicionalmente los siguientes dispositivos celulares fueron incorporados en las pruebas.

Dispositivos utilizados

- Sony Ericsson w910i
- Nokia 6101

Pruebas Funcionales

En la Tabla 11.8 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Múltiples conexiones	Comprobar el manejo de múltiples instancias de conexión, de acuerdo al número de dispositivos celulares conectados.	Instanciar tantos objetos conexión como dispositivos se encuentren conectados y poder invocar los métodos disponibles en el wrapper a través de cada conexión.

Tabla 11.8: Pruebas funcionales - Iteración 4

- **Prueba 1 - Múltiples conexiones**

La finalidad de las pruebas en esta iteración es comprobar que efectivamente el administrador de conexión instancia el número de conexiones que debe y que adicionalmente, éstas están siendo mapeadas satisfactoriamente al dispositivo celular correspondiente.

Para realizar las pruebas se incorporó un archivo llamado smsRuby.rb. El siguiente fragmento muestra las líneas incorporadas en dicho archivo de prueba:

```
require 'adm_connection'
admin= AdmConnection.new
@config={:dst=>['04129233403','04129233403','04129233403'],:msj=>'hola mundo
    ',:validity=>'0',:smsc=>nil,:report=>0}
admin.send(@config)
```

El programa instancia un objeto de la clase *AdmConnection*. En Ruby, cuando se instancia un objeto de una clase (se llama al método new), se ejecuta el método *initialize*. En éste caso, se instancian tantas conexiones como haya disponibles (como fue mostrado en la codificación), para luego invocar al método *send* con el fin de enviar el mensaje. Se observa que antes de proceder con el envío del mensaje se introdujeron valores para los parámetros requeridos por la función de envío.

Se colocaron impresiones en puntos claves que permitieran saber si el funcionamiento era correcto. Las impresiones fueron colocadas en los siguientes métodos.

```
def open_conn(name)
   ...
   puts 'Inicializando Conexión '+@@connections.size.to_s+'...'
   ...
end

def send(config)
   ...
   v=rand(@@connections.size)
   puts 'Para enviar el mensaje fue elegida la conexión '+v.to_s
   ...
end

def initialize(name)    # en connection
   ...
   error <= 0 ? @id_connection=error.to_i*-1 : 'error'
   @phone_model=phoneModel(@id_connection)
   puts 'Conneccion '+@id_connection.to_s+' mapeada a dispositivo '+
      @phone_model.to_s
   ...
end

def execute(hsh)
   ...
   puts 'Enviando mensaje '+hsh[:msj].to_s+' a '+ hsh[:dst].to_s+' por la
      coneccion '+      @id_connection.to_s+"\n"
   error= send_sms(hsh[:dst], hsh[:msj], hsh[:smsc], hsh[:report], hsh[:
      validity], @id_connection)
   ...
end
```

`send_sms` es una función definida en la extensión tal y como se observó en iteraciones anteriores.

- **Resultado**:

Al ejecutar el archivo de prueba con dos dispositivos celulares conectados, el resultado arrojado fue el siguiente:

```
"Inicializando Conexión 0..."
"Inicializando Conexión 1..."
"Para enviar el mensaje fue elegida la conexión 0"
Enviando mensaje hola mundo a 04129233403 por la conexión 0
Send succeeded with reference 0!
"Para enviar el mensaje fue elegida la conexión 1"
Enviando mensaje hola mundo a 04129233403 por la conexión 1
Send succeeded with reference 0!
"Para enviar el mensaje fue elegida la conexión 0"
Enviando mensaje hola mundo a 04129233403 por la conexión 0
Send succeeded with reference 0!
```

Con este resultado se puede constatar que se instancian dos conexiones distintas, una por cada dispositivo conectado. Además se hace uso de la función `send_sms` que se encuentra en el wrapper. Ésta funcionó correctamente, enviando tres mensajes desde las diferentes conexiones.

11.6 Iteración 5

Del 1-Jun-2009 al 5-Jun-2009

En esta iteración se implementa en la capa administración de conexión un esquema que permite distribuir un número "n" de mensajes, entre todas las conexiones disponibles, controlando así, el flujo y la carga de envíos. Se incorporan también, nuevas funciones a la capa de conexión que permiten conocer el estado de una conexión específica.

11.6.1 Planificación

Para esta iteración se continúa con el diseño, la codificación y las pruebas de la meta *Generación de la capa de conexión y administración de conexión del middleware SMS*. En la Tabla 11.9 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	01 - 06 - 2009	Añadir nuevas funcionalidades en la capa de conexión.
2	01 - 06 - 2009	Implementar esquema productor –consumidor para gestionar la carga de mensajes de texto a ser enviados.

Tabla 11.9: Historia de usuario - Iteración 5

11.6.2 Diseño

Una vez definidas las capas de conexión y administración de conexión se procede a incorporar nuevas funcionalidades a las mismas.

En primer lugar se añaden funcionalidades en la capa de conexión, que permiten conocer el estado de un teléfono celular específico conectado al computador (dicho celular es representado a través de una conexión). Las funciones mencionadas dan a conocer atributos como la intensidad de señal del celular, nivel de batería, modelo, fabricante, etc.

En segundo lugar se implementa un esquema productor –consumidor para gestionar el flujo y la carga de envíos entre distintas conexiones disponibles. Este esquema se implementa en el administrador de conexiones para determinar por cual conexión será enviado un mensaje específico. En particular se utiliza un esquema con buffer limitado, utilizando un productor y n consumidores. Cada consumidor representa una conexión y cada vez que envía un mensaje saca un nuevo ítem del buffer hasta que, entre todos los consumidores (conexiones) se envían todos los mensajes. En la Figura 11.6 se muestra una representación del esquema productor consumidor utilizado.

11.6.3 Codificación

Las siguientes funciones fueron incorporadas en la clase Connection que representa la capa de conexión del middleware SMS:

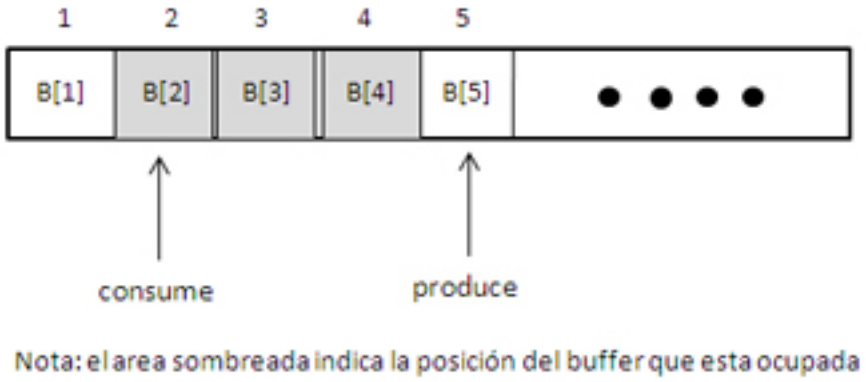

Figura 11.6: Esquema productor consumidor con buffer limitado

```
def signallevel
   return rf_level(@id_connection)
end
def batterylevel
   return bat_level(@id_connection)
end
```

Estas funciones permiten obtener el nivel de señal y de batería del dispositivo celular asociado a la conexión.

Adicionalmente fueron agregados varios atributos a la clase, cuyos valores forman parte del estado del dispositivo celular y por tanto, de la conexión.

```
@phone_model=phoneModel(@id_connection)
@phone_manufacter=phoneManufacter(@id_connection)
@phone_revsoft=phoneRevSoft(@id_connection)
@phone_imei=phoneImei(@id_connection)
```

Se puede observar que para obtener el valor deseado es invocada una función definida en la extensión creada en iteraciones anteriores. A dicha función es pasado como único argumento un `id_connection` que indica de cuál conexión se obtendrá el valor deseado.

Es creada una clase llamada *Synchronize* que permite, valiéndose de un conjunto de atributos, controlar el acceso concurrente del productor y los consumidores a los recursos críticos, así como manejar la sincronización entre ambos. El código de inicialización de los atributos mencionados se muestra a continuación:

```
def initialize
        @mutex = Mutex.new
        @mutexp=Mutex.new
    @empty = ConditionVariable.new
    @full = ConditionVariable.new
    @queue = Queue.new
    @eq = Queue.new
    @max = 10
end
```

El siguiente fragmento de código muestra como se ha realizado la implementación del productor en el administrador de conexiones para colocar en el buffer todos los ítems necesarios.

```
def producer(dest)
  dest.each do |i|
    begin
      @sync.mutex.synchronize{
        @sync.full.wait(@sync.mutex) if (@sync.count == @sync.max)
        @sync.queue.push i
        @sync.mutexp.synchronize{
          @produced += 1 if check
        }
        @sync.empty.signal if @sync.count == 1
      }
    end
  end
end
```

Donde *dest* representa una colección de números de dispositivos celulares a los que se envía el mensaje de texto. Cada consumidor extrae uno a uno los números, para así, enviar el mensaje a través de su conexión asociada, al número correspondiente. A continuación se muestra un fragmento de código que muestra la implementación realizada para los consumidores:

```
def consumer(n,max)
  Thread.current[:wfs]=0
  loop do
    @sync.mutexp.synchronize{
      Thread.exit if (@produced >= max && @sync.queue.empty?)
    }
    begin
      @sync.mutex.synchronize{
        if @sync.count == 0
          Thread.current[:wfs]=1
          @sync.empty.wait(@sync.mutex)
           Thread.current[:wfs]=0
        end
        Thread.current[:v] = @sync.queue.pop
        @sync.full.signal if (@sync.count == (@sync.max - 1))
      }
      @@connections[n].execute(to_hash(Thread.current[:v].to_s))
      @consumed+=1
    end
  end
end
```

Finalmente en la función *send*, interna al administrador de conexiones, se inicia un hilo productor (con la función producer asociada) y tantos hilos consumidores (con la función consumer asociada) como conexiones disponibles se obtengan. El segmento de código necesario para implementar lo especificado es el siguiente:

```
prod = Thread.new{producer(config[:dst])}
@@connections.each do |i,c|
   consu[i] = Thread.new{consumer(i,config[:dst].size)
end
```

Donde *config[:dst]* representa el vector que contiene el conjunto de destinatarios a los que se enviará el mensaje de texto.

11.6.4 Pruebas

Las pruebas de esta iteración se hicieron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.10 se encuentran las distintas pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Productor –Consumidor	Comprobar la correcta implementación del esquema productor –consumidor .	Producción de tantos elementos como números destinos especificados y consumo (por parte de todos los consumidores iniciados) de todos los elementos producidos.

Tabla 11.10: Pruebas funcionales - Iteración 5

- **Prueba 1 - Productor –Consumidor**

Se colocan impresiones en puntos claves que permitan saber si el funcionamiento es correcto. Las impresiones se colocan en los siguientes métodos:

```
def producer(dest)
  dest.each do |i|
    ...
         puts "productor: #{i} produced"+"\n"
    ...
  end
end

def consumer(n,max)
  ...
  Thread.current[:v] = @sync.queue.pop
  puts 'consumidor: en la connection '+n.to_s+' '+Thread.current[:v].to_s+'
     consumed'+"\n"
  ...
end
```

Se mantiene la impresión en *connection* indicando el envío de un mensaje por una conexión específica.

- **Resultado**:

Utilizando el mismo archivo de prueba de la iteración anterior, el resultado obtenido fue el siguiente:

```
04129233403 produced
consumidor: en la connection 0 04129233403 consumed
Enviando mensaje hola mundo a 04129233403 por la coneccion 0
Send succeeded with reference 0!
productor: 04129233403 produced
productor: 04129233403 produced
consumidor: en la connection 0 04129233403 consumed
Enviando mensaje hola mundo a 04129233403 por la coneccion 0
Send succeeded with reference 0!
consumidor: en la connection 0 04129233403 consumed
Enviando mensaje hola mundo a 04129233403 por la coneccion 1
Send succeeded with reference 0!
```

Se puede observar que cada uno de los elementos producidos (los números destino a los que el mensaje es enviado) fue consumido utilizando alguna de las conexiones disponibles y activas. Adicionalemnte el mensaje particular *Send succeeded with reference 0* indica que el mensaje se envió satisfactoriamente y se encuentra colocada en la librería Gnokii.

11.7 Iteración 6

Del 8-Jun-2009 al 12-Jun-2009

En esta iteración se implementa la capa de envío del middleware SMS, haciendo uso del patrón de diseño Strategy. Adicionalmente se incorpora el patrón de diseño Singleton en la capa de Administración de conexiones.

11.7.1 Planificación

Para esta iteración se establece la meta *Implementar la capa de envío del middleware SMS en Ruby.* En la Tabla 11.11 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	08 - 06 - 2009	Usar el patrón de diseño Strategy en la capa de envío.
2	08 - 06 - 2009	Incorporar el patrón de diseño Singleton en la capa de Administración de conexiones.

Tabla 11.11: Historia de usuario - Iteración 6

- Utilizar el patrón de diseño Strategy en la capa de envío.
- Incorporar el patrón de diseño Singleton en la capa de Administración de conexiones.

11.7.2 Diseño

Para el envío de un mensaje de texto deben especificarse ciertos parámetros como el destinatario y el mensaje a enviar, así como distintas opciones de envío que deseen utilizarse. Estos parámetros pueden ser especificados de distintas formas, por ejemplo pasándolos directamente al programa, a través de un archivo de configuración o inclusive por medio de una base de datos.

A través del patrón de diseño Strategy se ofrecen al usuario distintas estrategias de interacción con el sistema, utilizando diferentes variantes de un mismo algoritmo, sin conocer su implementación concreta.

La estructura de este patrón de diseño se muestra en la Figura 11.7 [Olsen, 2008].

Además, para mostrar las interacciones entre las clases que se utilizan para el envío de un mensaje, se muestra un diagrama de secuencia en la Figura 11.8

Por otra parte, al utilizar la capa de administración de conexiones, se observa como un potencial problema el hecho de que se instanciaran una y otra vez las conexiones manejadas por la capa de administración. Por esta razón se decide utilizar el patrón de diseño Singleton para asegurar que siempre se tenga una sola instancia de la clase de administración de conexiones (*AdmConnection*), y por lo tanto, una sola instancia de cada una de las conexiones que ésta maneja.

La estructura del patrón de diseño Singleton se muestra en la Figura 11.9.

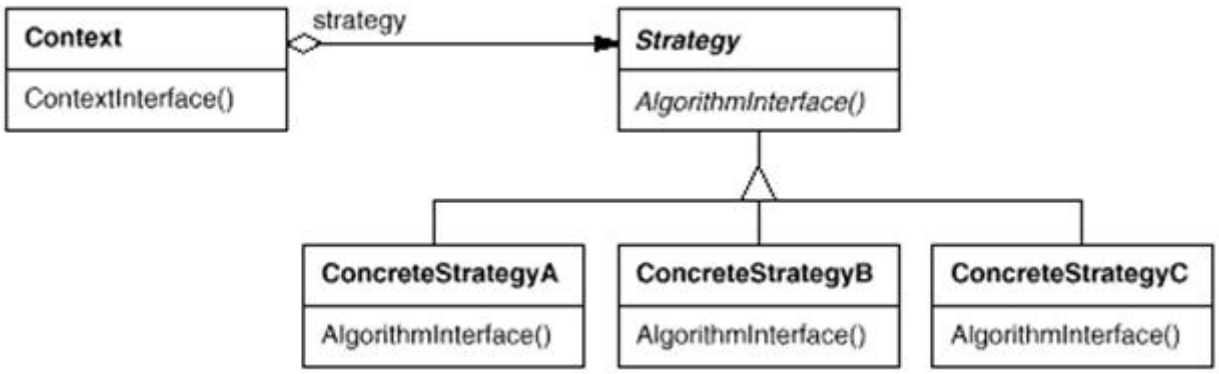

Figura 11.7: Patrón Strategy

11.7.3 Codificación

Para implementar el patrón *Strategy* se crea una clase base abstracta y distintas subclases que representan cada una de las estrategias. A continuación se muestra la implementación realizada de este patrón:

```
1  class Sender
2     attr_reader :adm, :dst, :msj, :report, :smsc, :validity
3     attr_accessor :sendtype
4     def initialize(sendtype)
5     ...
6     @sendtype = sendtype
7     end
8
9     def send
10       @sendtype.send(#parametros locales)
11    end
12 end
13
14 class Send
15
16   def send(#parametros formales)
17     # método abstracto
18   end
19
20 end
21
22 class PlainSend < Send
23   def send(context)
24     #obtengo los parámetros necesarios
25     context.adm.send(context)
26   end
27 end
28
29 class ConfigSend < Send
30   def send(context)
```

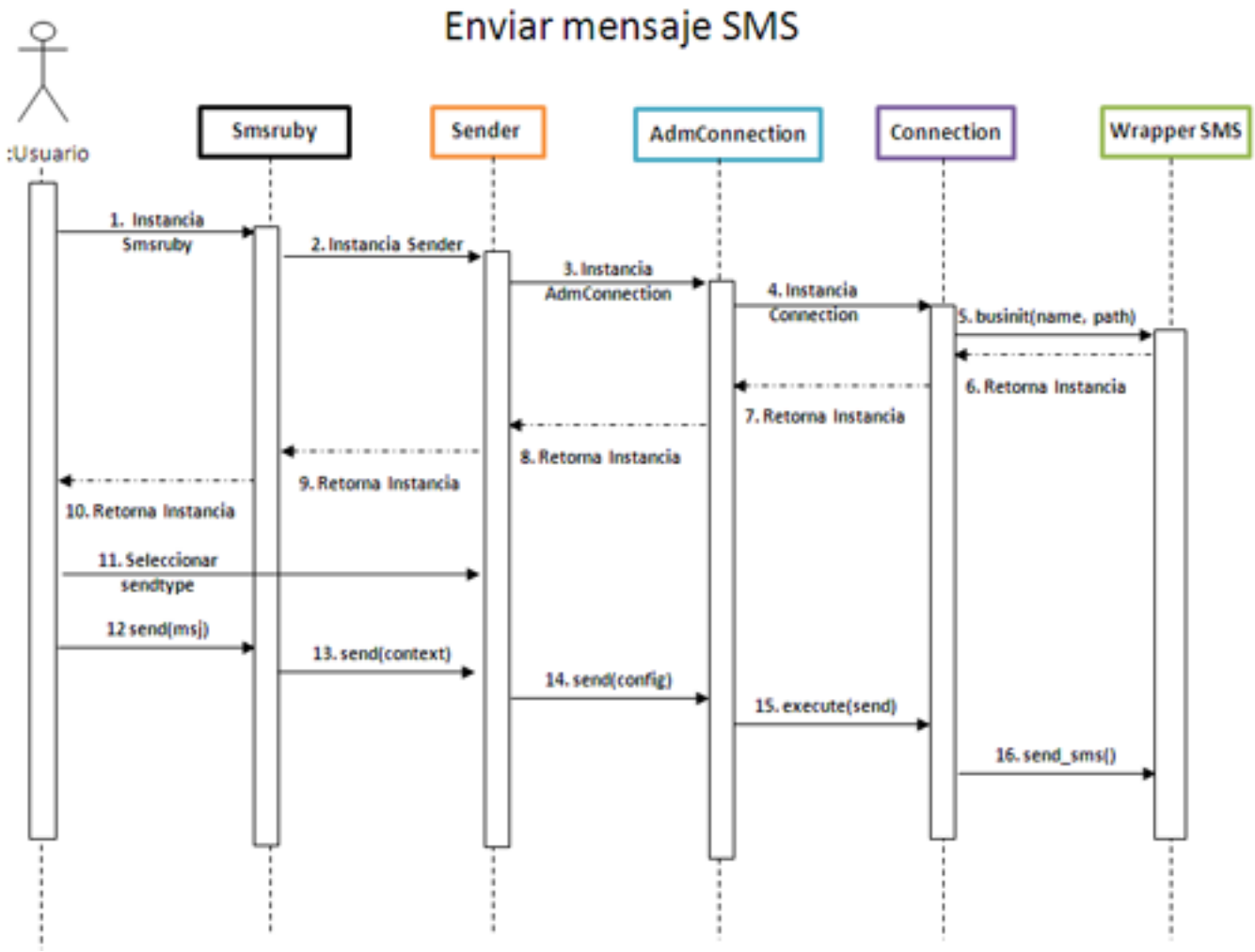

Figura 11.8: Diagrama de Secuencia - Envío de Mensajes SMS

```
31      #obtengo los parámetros necesarios a partir de un archivo de
            configuración
32      context.adm.send(context)
33    end
34  end
35
36  class BDSend < Send
37    def send(context)
38      #obtengo los parámetros necesarios de la base de datos
39        context.adm.send(context)
40    end
41  end
```

Sin embargo, al implementarse de esta manera no se cumple con una de las filosofías base de Ruby que es el *Duck Typing*, donde el tipo de un objeto está definido por lo que puede hacer, no por lo que es. En este caso las clases *PlainSend*, *ConfigSend* y *BDSend* comparten una interface común ya que

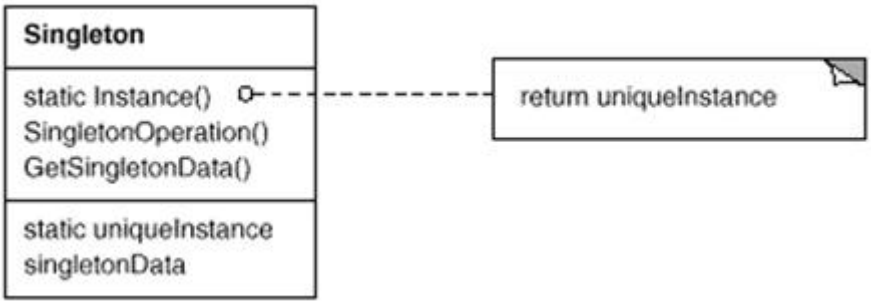

Figura 11.9: Patrón Singleton

todas implementan el método send, haciendo que no pueda distinguirse como tal una clase de otra. Para solucionar este inconveniente se implementó el patrón *Strategy* de la siguiente manera:

```
1  class Sender
2    attr_reader :adm, :dst, :msj, :report, :smsc, :validity
3    attr_accessor :sendtype
4    def initialize(sendtype)
5      @adm = AdmConnection.instance #instancia del Administrador de Conexiones
6      ...
7      @sendtype = sendtype
8    end
9
10    def send
11       @sendtype.send(self)
12    end
13 end
14
15 Class PlainSend
16   def send(context)
17      #obtengo los parámetros necesarios
18      context.adm.send(context)
19   end
20 end
21
22
23 Class ConfigSend
24   def send(context)
25      #obtengo los parámetros necesarios a partir de un archivo de
           configuración
26      context.adm.send(context)
27   end
28 end
29
30
31 Class BDSend
32   def send(context)
33      #obtengo los parámetros necesarios de la base de datos
34      context.adm.send(context)
```

```
35   end
36 end
```

La siguiente instrucción instanciará la clase *Sender* utilizando la estrategia *PlaindSend*:

```
@sender = Sender.new(PlainSend.new)
```

Para la implementación del patrón *Singleton* en la capa de administración de conexiones solo es necesario importar la clase *Singleton* de la siguiente manera:

```
require 'singleton'
```

El módulo Singleton implementa todo lo necesario para que el patrón funcione de manera adecuada. Crea la variable de clase y la inicializa con la instancia única, crea el método instance y convierte el método *new* en privado.

La variable `@@connections` contiene todas las conexiones que han sido asociadas, como es una variable de clase, su valor se mantendrá cada vez que se instancie la clase de administración de conexiones.

11.7.4 Pruebas

Las pruebas de esta iteración se hicieron bajo el mismo ambiente descrito en la iteración 4 .

Pruebas Funcionales

En la Tabla 11.12 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Patrón Strategy	Comprobar la correcta implementación del patrón Strategy y las distintas estrategias a utilizar.	Enviar mensajes utilizando las diferentes estrategias implementadas.
2	Patrón Singleton	Comprobar la correcta implementación del patrón Singleton.	Obtener una única instancia de la clase AdmConnection.

Tabla 11.12: Pruebas funcionales - Iteración 6

Para realizar las pruebas del patrón Strategy y el patrón Singleton se utilizó el shell interactivo de ruby (irb, Interactive Ruby Shell).

- **Prueba 1 - Patrón Strategy**

La finalidad de esta prueba es poder utilizar las distintas estrategias implementadas durante esta iteración, y poder constatar que se cumpla el diseño del patrón *Strategy*, que permite a los usuarios cambiar de una estrategia a otra en tiempo de ejecución.

```
irb(main):0001:0> require 'send'
=> true
irb(main):0002:0> a = Sender.new(Plainsend.new)
=> #<Sender:0x2e1b520, @sendtype = #<Plainsend:0x2e14068>>
irb(main):0003:0> a.setconfig(['04123853283'],nil,0,'0')
=> "0"
irb(main):0004:0> a.send('hello world')
Sent message 'hello world' to 04123853283
=> nil
irb(main):0005:0> a.sendtype = Configsend.new
=> #<Configsend:0x2e12ee8>
irb(main):0006:0> a.send('hello world')
Sent message 'hello world' to 04123853283
Sent message 'hello world' to 04129233403
=> nil
```

- **Resultado**:

En primer lugar es instanciada la clase *Sender* utilizando la estrategia *PlainSend*. Para esta estrategia los parámetros deben ser especificados explícitamente, razón por la cual se utilizó el método *setconfig*.

La estrategia puede ser modificada en tiempo de ejecución, ésto se puede observar al ejecutar la instrucción *a.sendtype = Configsend.new*, donde se escogió como nueva estrategia ConfigSend. En esta estrategia los parámetros necesarios para enviar un mensaje son leídos de un archivo de configuración. Para realizar esta prueba se contó con dos destinatarios.

- **Prueba 2 - Patrón Singleton**

A través del patrón Singleton se garantiza que una clase sólo tenga una instancia, por esto la finalidad de esta prueba es comprobar que se tenga una sola instancia de la clase *AdmConnection*.

```
irb(main):0001:0> require 'adm_connection'
=> true
irb(main):0002:0> a = AdmConnection.instance
=> #<AdmConnection:0x2baafac>
irb(main):0003:0> b = AdmConnection.instance
=> #<AdmConnection:0x2baafac>
irb(main):0004:0> a == b
=> true
```

- **Resultado**:

En esta prueba se instanció dos veces la clase *AdmConnection*, observandose que las instancias retornadas efectivamente son iguales.

11.8 Iteración 7

Del 15-Jun-2009 al 19-Jun-2009

Se desarrolla el módulo de manejo de errores y excepciones para dar robustez al sistema.

11.8.1 Planificación

Para esta iteración se establece la meta *Elaboración del módulo de manejo de errores y excepciones.* En la Tabla 11.13 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	15 - 06 - 2009	Crear una clase propia para el manejo de errores.
2	15 - 06 - 2009	Usar los códigos y mensajes de errores provistos por Gnokii en las clases creadas en Ruby.
3	15 - 06 - 2009	Manejar errores y excepciones generales que puedan ocurrir durante la ejecución de la aplicación.
4	15 - 06 - 2009	Manejar errores y excepciones de la aplicación específicamente en cuanto al envío de mensajes de texto.

Tabla 11.13: Historia de usuario - Iteración 7

11.8.2 Diseño

Al momento de interactuar con los dispositivos celulares pueden ocurrir diferentes tipos de errores, siendo algunas de las causas por ejemplo, que el dispositivo se encuentre fuera del área de cobertura, no posea saldo suficiente, se encuentre apagado, entre muchas otras. Algunos de estos errores no pueden ser corregidos, pero pueden ser manejados a través del uso de excepciones.

Ruby ofrece una clase para el manejo de excepciones, es por esto que cuando un error ocurre se crea automáticamente un objeto de la clase *Exception.* Por defecto, los programas Ruby terminan cuando una excepción ocurre, pero es posible escribir código que maneje estas excepciones tal como se hizo en esta iteración.

Adicionalmente Ruby posee algunas clases predefinidas que heredan de la clase Exception, las cuales ayudan a manejar errores particulares que ocurren en el programa.

Por otro lado, para llevar el control de los errores ocurridos se utiliza un log proporcionado por la clase Logger, la cual maneja distintos niveles para tratar los errores. Estos niveles son:

- FATAL: Error que no puede ser manejado y por esto finaliza el programa.
- ERROR: Error que puede ser manejado.
- WARN: Advertencia.

- INFO: Información útil acerca de las operaciones del sistema.
- DEBUG: Información de bajo nivel, que es útil para los desarrolladores.

En el caso particular del envío de mensajes de texto los errores deben ser manejados de manera que la aplicación se haga robusta. Por ejemplo, si algún mensaje no ha podido ser enviado, se debería intentar enviar dicho mensaje a través de otra conexión disponible y no descartarlo.

11.8.3 Codificación

Se define un módulo (*ErrorHandler*) para tratar los errores. Este contiene una clase *Error* que extiende de la clase *StandardError*. A continuación se muestra la clase Error:

```
1  class Error < StandardError
2
3     attr_reader :id
4     attr_reader :message
5
6     def initialize(id)
7        @id = id
8        @message = printError(id)
9     end
10
11 end
```

En esta clase se definen dos atributos, un id y un message, el id identifica el tipo de error y el message da información acerca del mismo. Adicionalmente se implementan distintas clases de error (extienden de la clase Error) para agrupar y caracterizar los errores. Estas subclases se muestran a continuación:

```
1  class GeneralError < Error; end
2
3  class ConfigError < Error; end
4
5  class StatemachineError < Error; end
6
7  class CallError < Error; end
8
9  class OtherError < Error; end
10
11 class FormatError < Error; end
12
13 class LocationError < Error; end
```

Al producirse un error, el método *exception* es invocado pasando como argumento el id del error, este id permite determinar que excepción es lanzada a través del método *raise*, pudiendo así, tomar las medidas adecuadas en base a la categoría del error.

```
1  def exception (error)
2     case error
3        when 1..9 then raise ErrorHandler::GeneralError.new(error)
```

```
4         when 10..15 then raise ErrorHandler::StatemachineError.new(error)
5         when 16..18 then raise ErrorHandler::LocationError.new(error)
6         when 19..21 then raise ErrorHandler::FormatError.new(error)
7         when 22..25 then raise ErrorHandler::CallError.new(error)
8         when 26..29 then raise ErrorHandler::OtherError.new(error)
9         when 30..35 then raise ErrorHandler::ConfigError.new(error)
10     end
11 end
```

Para manejar una excepción es indispensable que el bloque de código que la generó se encuentre dentro de una sentencia begin –end seguido de una o más cláusulas *rescue*. Si se escribe una cláusula *rescue* sin lista de parámetros, el parámetro tomado por defecto será *StandardError*. Cada cláusula *rescue* puede especificar múltiples excepciones que rescatar. A continuación se muestra un ejemplo de lo mencionado anteriormente.

```
1 begin
2     ...
3 rescue ErrorHandler::ConfigError
4     ...
5 rescue ErrorHandler::FormatError
6     ...
7 rescue ErrorHandler::Error
8     ...
9 end
```

Para la implementación del log de mensajes se utiliza la clase *Logger*:

```
1 require 'logger'
2
3 @log = Logger.new ('sms.log')
```

Para colocar un mensaje en el log se escribe lo siguiente:

```
<nombre del log>.<Nivel> <Mensaje a colocar en el log>
```

Por ejemplo: `@log.error` "Problem writing the configuration file". En este caso particular un mensaje de error es guardado en el log.

Se implementa la función *retryable* como parte del manejo de excepciones. Dicha función es mostrada a continuación:

```
1  def retryable(options = {}, &block)
2
3     opts = { :tries => 1, :on => Exception }.merge(options)
4
5     retry_exception, retries = opts[:on], opts[:tries]
6
7     begin
8        return yield
9     rescue retry_exception
10       retry if (retries -= 1) > 0
11    end
12    yield
```

```
13 end
```

Esta función permite ejecutar un bloque de código pasado como argumento tantas veces como indique la variable tries, en caso de ocurrir la excepción especificada en la variable `retry_exception`. A continuación se muestra un ejemplo de su uso en la clase AdmConnection.

```
1 retryable(:tries => 2, :on => ErrorHandler::Error) do
2    @@connections[n].execute(to_hash(Thread.current[:v].to_s))
3 end
```

Si al intentar enviar el mensaje dos veces el envío falla nuevamente, el mensaje se coloca en una cola de emergencia, que es atendida cuando terminan de enviarse todos los mensajes de la cola original.

Los mensajes contenidos en la cola de emergencia, tratan de enviarse por cada una de las conexiones disponibles hasta que los mismos puedan ser enviados satisfactoriamente. Si algún mensaje no pudo ser enviado, el mismo se descarta y se coloca un mensaje en el log.

Para implementar lo anteriormente explicado se utiliza la función `send_emergency`:

```
1  def send_emergency(n,max)
2     Thread.current[:wfs]=0
3     loop do
4        begin
5           @sync.mutexe.synchronize{
6           if (@sync.eq.size == 0 and @consumed < max)
7              Thread.current[:wfs] =  1
8              @sync.emptye.wait(@sync.mutexe)
9              Thread.current[:wfs] = 0
10          elsif (@consumed == max)
11             Thread.exit
12          end
13          unless @sync.eq.empty?
14             Thread.current[:v]=@sync.eq.pop
15             retryable(:tries => 2, :on => ErrorHandler::Error) do
16                @@connections[n].execute(to_hash(Thread.current[:v].to_s))
17             end
18             @consumed+=1
19             (Thread.list.each{|t| @sync.emptye.signal if (t!=Thread.current
                  and t!=Thread.main and t[:wfs]==1)})   if @consumed == max
20          end
21          }
22       rescue Exception => e
23          @sync.mutexe.synchronize{
24             @log.error "Can't send the message for connection in port #{
                  @array[n]}. Exception:: #{e.message}" unless @log.nil?
25             @sync.eq << Thread.current[:v]
26             @sync.emptye.signal if @sync.eq.size==1
27          }
28          Thread.exit
29       end
30    end
31 end
```

11.8.4 Pruebas

Las pruebas de esta iteración se realizaron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.14 se encuentran las distintas pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Excepciones no manejadas	Establecer una relación entre distintos escenarios irregulares y códigos de error o excepciones arrojadas.	Terminación anormal del programa debido a alguna excepción particular, generada a partir de distintos escenarios irregulares planteados.
2	Excepciones manejadas	Verificar que el manejo de excepciones incorporado evite la terminación anormal del programa y de información suficiente acerca de la causa de error.	Registro en el log de eventos de la excepción generada sin provocar la terminación anormal del programa.

Tabla 11.14: Pruebas funcionales - Iteración 7

Para todas las pruebas fue colocada una impresión luego de la invocación al método que genera la excepción, tal y como se muestra a continuación

```
sms.sender.dst=['04129985118']
sms.send("Hola Mundo!!!")
puts "Programa terminado de manera normal"
```

- **Prueba 1 - Excepciones no manejadas**

Para realizar las pruebas se generaron distintos escenarios, a través de los cuales se produjeron varios tipos de excepciones.

1. Al intentar enviar un mensaje de texto en un lugar donde el dispositivo celular estuviese fuera del área de cobertura, ocurre la siguiente excepción:

```
'exception': No Carrier error during data call setup? (ErrorHandler::
   CallError)
```

2. Si el archivo de configuración no es encontrado en la ruta establecida el intentar enviar un mensaje de texto genera la siguiente excepción:

```
>ruby test.rb
Couldn't read C:/Tesis/_gnokiirc config file.
'exception': Config file cannot be read. (ErrorHandler::ConfigError)
```

3. Si el dispositivo celular es desconectado antes que el mensaje de texto pueda ser enviado, ocurre la siguiente excepción:

```
'exception': Command timed out. (ErrorHandler::StatemachineError)
```

- **Resultado**:

En esta primera prueba la impresión *Hola Mundo!!!* colocada nunca es abservada debido a que el programa termina de manera anormal.

- **Prueba 2 - Excepciones manejadas**

En los casos anteriores el programa termina de forma incorrecta debido a que ocurren excepciones que no son controladas, para esto, al programa de prueba se le añadió un manejo sencillo de excepciones, donde se utiliza un log para llevar un control de los errores ocurridos. Un ejemplo es mostrado a continuación:

```
1  def consumer(n,max)
2     begin
3        ...
4        rescue ErrorHandler::Error => ex
5          @log.error "Can't send the message for connection in port #{@array[n
               ]}. Exception:: #{ex.message}" unless @log.nil?
6               ...
7        rescue Exception => ex
8          @log.error "Can't send the message for connection in port #{@array[n
               ]}. Exception:: #{ex.message}" unless @log.nil?
9        end
10     end
11 end
```

- **Resultado**:

Utilizando el mismo programa que en la prueba 1, se producen los siguientes resultados en el log creado:

```
# Logfile created on Wed Jun 17 01:32:29 -0400 2009 by /
E, [2009-06-17T01:32:29.000000 #3396] ERROR -- : Can't send the message for
    connection in port COM1. Exception:: No Carrier error during data call
    setup?
E, [2009-06-17T01:34:02.000000 #2532] ERROR -- : Can't send the message for
    connection in port COM1. Exception:: Config file cannot be read.
E, [2009-06-17T01:34:41.406000 #2328] ERROR -- : Can't send the message for
    connection in port COM1. Exception:: Command timed out.
```

Adicionalmente se observa la impresión del *Hola Mundo!!!* colocado. Lo que permite concluir que el programa terminó de manera normal manejando la excepción obtenida de manera adecuada.

11.9 Iteración 8

Del 22-Jun-2009 al 26-Jun-2009

En esta iteración se implementan nuevas funciones en la librería de código abierto Gnokii para dar soporte al comando AT+CMGL, utilizado para obtener mensajes de texto del dispositivo celular.

11.9.1 Planificación

Para esta iteración se establece la meta *Incorporación del comando AT+CMGL en la librería de código abierto Gnokii*. En la Tabla 11.15 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	22 - 06 - 2009	Implementar las funciones necesarias para dar soporte al comando AT+CMGL siguiendo la estructura utilizada en Gnokii.

Tabla 11.15: Historia de usuario - Iteración 8

11.9.2 Diseño

El comando AT+CMGL permite obtener los mensajes almacenados en un dispositivo celular, razón por la cual, se incorporan las funcionalidades necesarias en la librería de código abierto Gnokii para dar soporte a dicho comando, pudiendo así, desarrollar posteriormente el módulo de recepción de mensajes de texto.

Básicamente la secuencia de funciones que deben incorporarse en Gnokii para dar soporte a un nuevo comando son las siguientes:

- Función de alto nivel invocada por los usuarios.
- Función interna que permita especificar el comando AT a escribir en el puerto serial al que se encuentra conectado el dispositivo celular.
- Función *Replay* que permita procesar la respuesta dada por el dispositivo luego de procesar el comando. Los mensajes leídos se almacenan en una estructura interna codificados y con una cabecera.
- Función *parse* que permita recorrer cada uno de los mensajes almacenados, pasarlos por una función de decodificación y almacenarlos en una nueva estructura accesible por los usuarios.

11.9.3 Codificación

En primer lugar se incorporan a la estructura de datos utilizada por Gnokii dos nuevas variables que permiten almacenar tanto los mensajes codificados, como los mensajes decodificados legibles por los usuarios.

```
1 typedef struct {
2         ...
3         gn_sms_raw *raw_sms_list[GN_SMS_MAX_MESSAGES];
4         gn_sms *sms_list[GN_SMS_MAX_MESSAGES];
```

```
        ...
}gn_data;
```

A continuación se implementa la función de alto nivel para obtener los mensajes del dispositivo celular. Éstos son almacenados en las estructuras antes mencionadas.

```
GNOKII_API gn_error gn_sms_get_list(gn_data *data, struct gn_statemachine *
    state)
{
        gn_error error;
        gn_sms_raw rawsms;
        ...
        memset(&rawsms, 0, sizeof(gn_sms_raw));
        rawsms.memory_type = data->sms->memory_type;
        data->raw_sms = &rawsms;
        error = gn_sms_request_get_list(data, state);
        ERROR();
        return gn_sms_parse_list(data);
}
```

En dicha función se establece el tipo de memoria a utilizar del dispositivo celular (memoria del SIM, memoria del teléfono, etc.). Seguidamente se hace el request para invocar a la función interna, encargada de especificar el comando AT a escribir en el puerto serial, y por último se invoca a la función encargada de decodificar los mensajes almacenados en `raw_sms_list` y almacenarlos en `sms_list`.

```
gn_error  gn_sms_request_get_list(gn_data *data, struct gn_statemachine *
    state)
{
   if (!data->raw_sms) return GN_ERR_INTERNALERROR;
   return gn_sm_functions(GN_OP_GetListSMS, data, state);
}
```

En la función `gn_sms_request_get_list` se invoca la función general de gnokii (`gn_sm_functions`) que, dependiendo de la constante especificada como parámetro, determina cuál es la función correcta para realizar la labor requerida.

Para que `gn_sm_functions` funcione correctamente es necesario realizar una asociación entre la constante especificada, la función a invocar para dicha constante y la función que se invoca una vez obtenida la respuesta para el comando AT, la cual se encarga de procesar dicha respuesta (todas estas funciones fueron implementadas para dar soporte al nuevo comando).

Con el siguiente vector se realiza la asociación antes especificada, siendo la última entrada la colocada para asociar las funciones que dan soporte al comando AT+CMGL.

```
static at_function_init_type at_function_init[] = {
  { GN_OP_Init,                NULL,                    Reply },
  { GN_OP_Terminate,           Terminate,               Reply },
  ...
  { GN_OP_GetListSMS,        AT_GetListSMS,          ReplyGetListSMS },
};
```

Se implementa la función `AT_GetListSMS`, siendo el fragmento de código más importante el siguiente:

```
static gn_error AT_GetListSMS(gn_data *data, struct gn_statemachine *state)
{
   unsigned char req[32];
   gn_error err;
   at_set_charset(data, state, AT_CHAR_GSM);
   err = AT_SetSMSMemoryType(data->raw_sms->memory_type,  state);

   if (err)
      return err;
   ...
   snprintf(req, sizeof(req), "AT+CMGL=%d\r", 4);
   if (sm_message_send(strlen(req), GN_OP_GetListSMS, req, state))
      return GN_ERR_NOTREADY;
   return sm_block_no_retry(GN_OP_GetListSMS, data, state);
}
```

En la función se especifica el tipo de memoria a utilizar y se manda al dispositivo celular el comando AT+CMGL con valor 4 a través de la función `sm_message_send`(una serie de funciones internas son ejecutadas para lograr la escritura del comando en el puerto serial al que se encuentra conectado el dispositivo celular).

Una vez que se obtiene una respuesta del dispositivo celular para el comando especificado, la función ReplyGetListSMS es invocada procesando la respuesta y almacenando los mensajes codificados en la estructura `raw_sms_list`, utilizando la función `gn_sms_pdu2raw` implementada en Gnokii. Un fragmento de la función ReplyGetListSMS se muestra a continuación.

```
static gn_error ReplyGetListSMS(int messagetype, unsigned char *buffer, int
    length, gn_data *data, struct gn_statemachine *state)
{
   at_line_buffer buf;
   gn_error ret = GN_ERR_NONE;
   unsigned int sms_len, pdu_flags, ind_raw=0,number;
   unsigned char *tmp, *aux;
   gn_error error;
   buf.length = length;
   aux = strtok( buffer, "\r\n" );    // Primera llamada => Primer token
   buf.line1=aux;
   aux = strtok( NULL, "\r\n" );
   while( strcmp(aux,"OK")!=0 && strcmp(aux,"ERROR")!=0 && aux != NULL){
      buf.line2=aux;
      aux = strtok( NULL, "\r\n" );
      buf.line3=aux;
      ...
      data->sms_list[ind_raw]= (gn_sms *)malloc(sizeof(gn_sms));
      memset(data->sms_list[ind_raw], 0, sizeof(gn_sms));
      memset(data->raw_sms_list[ind_raw], 0, sizeof(gn_sms_raw));
      data->raw_sms_list[ind_raw]->memory_type = data->sms->memory_type;
      // buf.line2 es procesado para obtener todos los atributos del mensaje
          leido
      sms_len = strlen(buf.line3) / 2;
      tmp = calloc(sms_len, 1);
      if (!tmp) {
         dprintf("Not enough memory for buffer.\n");
```

```
26            return GN_ERR_INTERNALERROR;
27        }
28        dprintf("%s\n", buf.line3);
29        hex2bin(tmp, buf.line3, sms_len);
30        ret = gn_sms_pdu2raw(data->raw_sms_list[ind_raw], tmp, sms_len,
              pdu_flags);
31        if (tmp)
32            free(tmp);
33        aux = strtok( NULL, "\r\n" );
34        ind_raw++;
35    }
36    return ret;
37 }
```

Una vez completada la ejecución de la función ReplayGetListSms los mensajes de texto leídos del dispositivo celular quedan almacenados en la estructura `raw_sms_list`.

El último paso es la decodificación de los mensajes almacenados en `raw_sms_list`. Para esto se implementa la función `gn_sms_parse_list`, cuyo código se muestra a continuación:

```
1  gn_error gn_sms_parse_list(gn_data *data)
2  {
3      int ind_raw = 0;
4      gn_error error = GN_ERR_NONE;
5      while(data->raw_sms_list[ind_raw]){
6          data->sms_list[ind_raw]->status = data->raw_sms_list[ind_raw]->status;
7          error = sms_pdu_decode(data->raw_sms_list[ind_raw], data->sms_list[
               ind_raw]);
8          ind_raw++;
9      }
10     return error;
11 }
```

La decodificación de cada mensaje es llevada a cabo por una función provista por Gnokii llamada `sms_pdu_decode`.

11.9.4 Pruebas

Las pruebas de esta iteración se realizaron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.16 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

- **Prueba 1 - Comando AT+CMGL**

Con estas pruebas se busca comprobar que a través del comando AT+CMGL, se puedan listar los mensajes de un dispositivo celular específico, siempre y cuando, este dispositivo soporte el comando.

Se implementó un programa que permitiera probar la nueva función incorporada a la librería *Gnokii*. Dicho programa se muestra a continuación:

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Comando AT+CMGL	Comprobar el correcto funcionamiento del comando AT+CMGL en la librería Gnokii.	Terminación normal del programa. EL programa debe listar los mensajes del dispositivo celular, utilizando el comando implementado.

Tabla 11.16: Pruebas funcionales - Iteración 8

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <gnokii.h>
struct gn_statemachine *state;
int main(int argc, char *argv[]) {
   gn_error     error;
   gn_sms               *message;
   gn_data              data;
   gn_sms_status        smsstatus = {0, 0, 0, 0};
   char                 *memory_type_string;
   int i=0;

   businit();

   memory_type_string = "ME";

   gn_data_clear(&data);

   data.sms_status = &smsstatus;

   message=(gn_sms*)malloc(sizeof(gn_sms));
   memset(message, 0, sizeof(gn_sms));
   message->memory_type = gn_str2memory_type(memory_type_string);
   data.sms = message;
   gn_memory_status phonememorystatus = {GN_MT_ME, 0, 0};
   data.memory_status = &phonememorystatus;

   error = gn_sms_get_list(&data, state);

   while(data.sms_list[i]){
      printf("::Mensaje obtenido. Enviado por: %s El mensaje dice: %s \n",
         data.sms_list[i]->remote.number,data.sms_list[i]->user_data[0].u.
         text);
      i++;
   }

   printf("%s\n", gn_error_print(error));
   busterminate();
   exit(error);
}
```

Se observa cómo es recorrida la estructura `sms_list` en busca de los mensajes obtenidos luego de la llamada a la función `gn_sms_get_list` incorporada a la librería *Gnokii*.

- **Resultado**:

La salida generada luego de la ejecución del programa fue la siguiente (El dispositivo celular conectado tenía almacenado dos mensajes al momento de las pruebas):

```
::Mensaje obtenido. Enviado por: +584129233403 el mensaje dice: Hola mundo
No error
::Mensaje obtenido. Enviado por: +584129233403 el mensaje dice: Hola mundo2
No error
```

Con estos resultados se puede constatar que se estan obteniendo los mensajes que se encuentran en el dispositivo celular.

11.10 Iteración 9

Del 29-Jun-2009 al 3-Jul-2009

En esta iteración se implementa el módulo de recepción de mensajes de texto utilizando la librería de código abierto Gnokii.

11.10.1 Planificación

Para esta iteración se establece como meta *Crear módulo de recepción SMS en lenguaje C*. En la Tabla 11.17 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	29 - 06 - 2009	Dar soporte a la recepción de mensajes de texto utilizando dispositivos celulares de distintos fabricantes.
2	29 - 06 - 2009	Borrar los mensajes de texto del dispositivo celular una vez leídos.

Tabla 11.17: Historia de usuario - Iteración 9

11.10.2 Diseño

El principal problema que se presenta en el desarrollo del objetivo planteado, es la diferencia que existe con respecto al manejo de la memoria donde residen los mensajes de texto en el dispositivo celular. En algunos casos los tipos de memoria varían de un dispositivo a otro, en otros aunque los tipos son iguales, la memoria donde los mensajes nuevos son almacenados difiere y en algunos se dificulta el funcionamiento de algún comando AT particular utilizado para recepción de mensajes.

Para ello se han creado dos funciones distintas que permitan ejecutar dos comandos AT diferentes en el dispositivo celular, ambos comandos destinados a la recepción de mensajes. El primero (AT+CMGR) obtendrá solo un mensaje a la vez (debe indicarse la localización del mensaje en la memoria del teléfono) y el segundo (AT+CMGL) devolverá una lista con todos los mensajes almacenados en el teléfono (no es necesario indicar la localización de los mensajes en la memoria).

Una vez obtenidos los mensajes de texto, independientemente del comando AT utilizado, los mensajes serán borrados del dispositivo celular para evitar que la memoria donde residen los mensajes se llene.

11.10.3 Codificación

En primer lugar fue modificado el archivo de cabecera *smsr.h* para incorporar una estructura de datos que represente un mensaje de texto con sus distintos atributos. Dicha estructura se muestra a continuación:

```
1 typedef struct {
2    int error;
3    int index;
4    char *date;
5    char *status;
6    char *source_number;
7    char *text;
```

```
8     char *type_sms;
9  } gn_message;
```

La estructura `gn_connection` se ha modificado, agregándose una nueva variable. Dicha variable hace referencia a cada una de las estructuras `gn_data` asociadas a las conexiones existentes. La estructura `gn_connection` se muestra a continuación:

```
1  typedef struct {
2     int activeconnect;
3     struct gn_statemachine *connections[CONNECTIONS_MAX_LENGTH];
4     gn_data *datag[CONNECTIONS_MAX_LENGTH];
5  } gn_connection;
```

En el archivo *sms.c* se han implementado las siguientes funciones:

```
1  int get_sms(int idconn){
2     return get_smsi(&conn,idconn);
3  }
```

La finalidad de esta función es obtener el número total de mensajes que se han leído del dispositivo celular asociado a la conexión especificada. Para ello es invocada la función interna `get_smsi` pasando el identificador de la conexión que hace la solicitud y una referencia a la estructura `gn_connection`.

```
1  gn_message get_msj(int number, int idconn){
2     return get_msji(number, &conn,idconn);
3  }
```

La finalidad de esta función es obtener el mensaje de número "number" correspondiente al número de mensajes totales obtenidos en una llamada previa a la función `get_sms`. La función devolverá una estructura de tipo `gn_message`.

A continuación se muestran fragmentos de código de la función `get_smsi` que permiten observar cómo se realiza la diferenciación entre los tipos de dispositivos y tipos de memoria.

```
1  int get_smsi(struct gn_statemachine *state,gn_connection *conn,int id_conn)
       {
2     gn_data      *data;
3     gn_message      sms;
4     int          total,i,cont = 0;
5     data=conn->datag[id_conn];
6     . . .
7     strcpy(manufacter,gn_lib_get_phone_manufacturer(state));
8     if (!at_manufacturer_compare(state->sm_data.manufacturer,"samsung")){
9        memory_type_string = "SM";
10       ...
11            sms = get_sms_msj(i,total,memory_type_string,data,state);
12            ...
13       list_sms[cont] = sms;
14    }
15    else {
16       if (!at_manufacturer_compare(state->sm_data.manufacturer,"motorola")){
17          memory_type_string = "MT";
18       }
```

```
19        else if (!at_manufacturer_compare(state->sm_data.manufacturer,"Sony
              Ericsson")){
20           memory_type_string = "ME";
21        }
22        else{
23           memory_type_string = "SM";
24        }
25        cont = get_sms_list(memory_type_string,data,state,id_conn);
26     }
27     return cont;
28  }
```

Como puede observarse básicamente dos funciones son utilizadas:

```
1  int get_sms_list(char *memory_type_string, gn_data *data, struct
       gn_statemachine *state,int id_conn){
2     ...
3     message->memory_type = gn_str2memory_type(memory_type_string);
4     error = gn_sms_get_list(data, state);
5     ...
6  }
```

```
1  gn_message get_msji(int number, struct gn_statemachine *state,gn_connection
       *conn,int id_conn){
2     gn_message   sms;
3     gn_error     error;
4     int          locations[8];
5
6     if (!at_manufacturer_compare(state->sm_data.manufacturer,"samsung")){
7        return list_sms[number];
8     }
9     else{
10       // es rellenada la variable sms con la información del mensaje
             solicitado
11       gn_sms_delete(conn->datag[id_conn], state);
12       for(i=0;i<k;i++){
13          conn->datag[id_conn]->sms->number=locations[i];
14          gn_sms_delete(conn->datag[id_conn], state);
15       }
16       return sms;
17    }
18 }
```

11.10.4 Pruebas

Las pruebas de esta iteración se realizaron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.18 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Recepción SMS	Verificar el correcto funcionamiento del módulo de recepción SMS.	Terminación normal del programa. El programa debe listar los mensajes de los dispositivos celulares utilizando el módulo de recepción implementado.

Tabla 11.18: Pruebas funcionales - Iteración 9

- **Prueba 1 - Recepción SMS**

Se implementó un programa que permita comprobar el funcionamiento adecuado del módulo de recepción realizado. Dicho programa se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <gnokii.h>
#include "smsr.h"

int main(int argc, char *argv[]) {
  gn_error      error=0;
  gn_message    sms;
  int conn,i,total;

  conn = businit(NULL,NULL);

  total=get_sms(conn*-1)*-1;

  for(i=0;i<total;i++){
         sms=get_msj(i, conn*-1);
         printf("::Mensaje obtenido. Enviado por: %s El mensaje dice: %s \n"
             ,sms.source_number,sms.text);
  }

  exit(error);
}
```

- **Resultado**:

La salida generada luego de la ejecución del programa fue la siguiente (El dispositivo celular conectado tenía almacenado dos mensajes al momento de las pruebas):

```
::Mensaje obtenido. Enviado por: +584129233403 el mensaje dice: Hola mundo
No error
::Mensaje obtenido. Enviado por: +584129233403 el mensaje dice: Hola mundo2
No error
```

Los mensajes del dispositivo celular conectado fueron listados correctamente, además los mensajes leidos fueron borrados del dispositivo.

11.11 Iteración 10

Del 6-Jul-2009 al 10-Jul-2009

En esta iteración se implementa la capa de recepción de mensajes de texto en Ruby del middleware SMS, utilizando la extensión creada en iteraciones anteriores.

11.11.1 Planificación

Para esta iteración se establece la meta *Creación de la capa de recepción del middleware SMS*. En la Tabla 11.19 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	06 - 07 - 2009	Implementar la capa de recepción de mensajes de texto en Ruby haciendo uso de la extensión generada en iteraciones anteriores.
2	06 - 07 - 2009	Permitir la recepción de mensajes de texto desde distintos dispositivos simultáneamente.
3	06 - 07 - 2009	Definir e implementar distintos tipos de recepción de mensajes.

Tabla 11.19: Historia de usuario - Iteración 10

11.11.2 Diseño

Ya que pueden disponerse de las principales funcionalidades de recepción desde Ruby, la idea ahora es realizar la implementación completa de la capa manejando distintos tipos de recepción, y haciendo posible, que distintos celulares se encuentren en modo recepción al mismo tiempo.

Existe la posibilidad que alguna acción o conjunto de instrucciones sean ejecutadas con la llegada de cada mensaje de texto. Basta con pasar un bloque de código a la función de recepción y dicho bloque es ejecutado tantas veces como mensajes de texto se reciban en el intervalo de tiempo establecido para recepción.

Es importante resaltar que mientras un dispositivo celular se encuentre en modo de recepción puede haber otros dispositivos enviando mensajes, sin embargo, un teléfono no puede recibir y enviar mensajes simultáneamente.

Para identificar que dispositivos celulares están autorizados para enviar y recibir mensajes de texto, se utiliza un archivo de configuración donde es colocada la Identidad Internacional de Equipo Móvil (IMEI, International Mobile Equipment Identity) de cada dispositivo celular autorizado para enviar y recibir.

En la Figura 11.10 se muestra un diagrama de secuencia, donde se puede observar la interacción de las distintas clases al recibir un mensaje.

11.11.3 Codificación

Para la recepción de mensajes se utilizan dos funciones del módulo de recepción realizado en lenguaje C, `get_sms` y `get_msj`. Con la primera función se obtiene el número de mensajes nuevos que posee el dispositivo celular en un instante determinado o devuelve un error en caso de fallo. Con la segunda

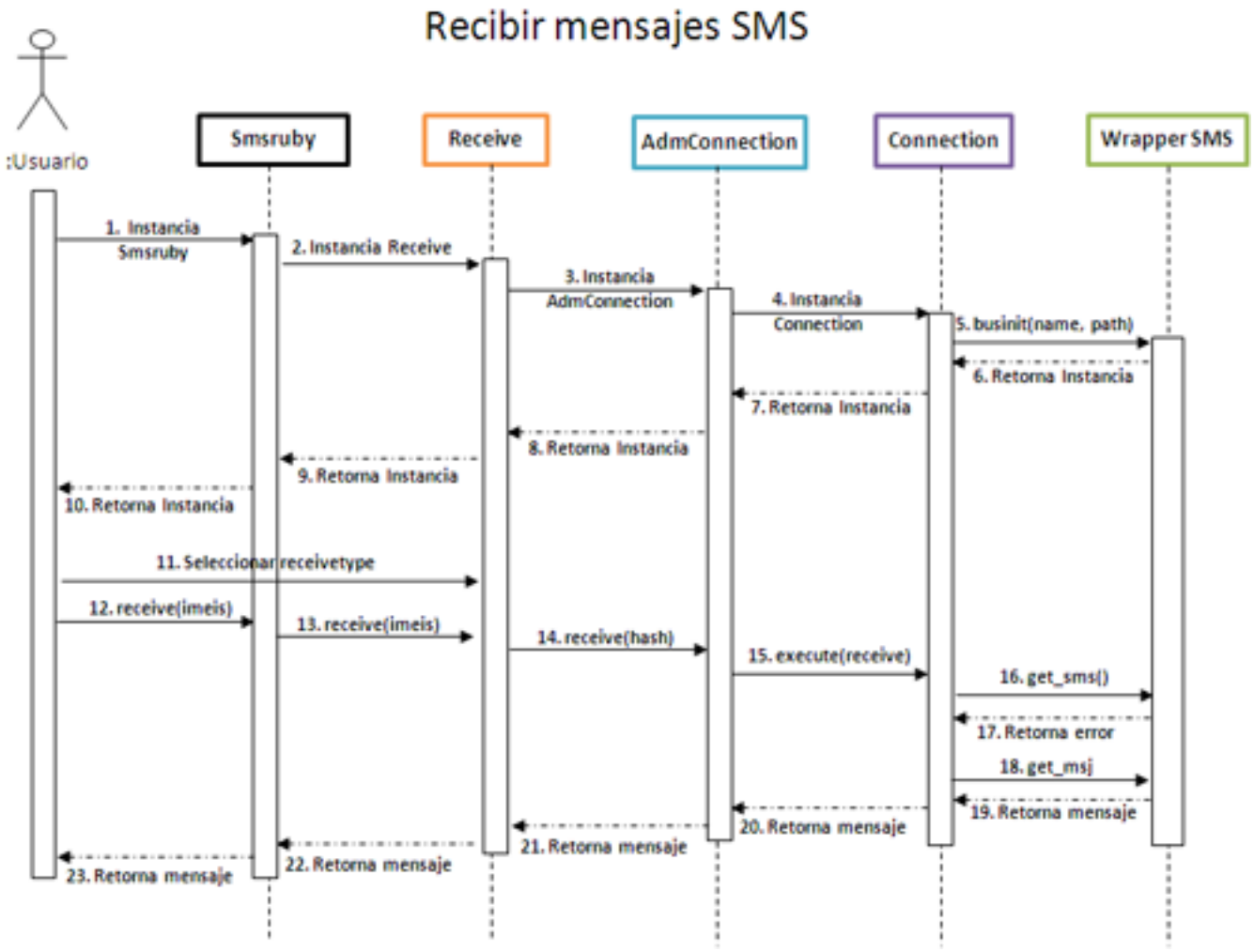

Figura 11.10: Diagrama de Secuencia - Recepción de Mensajes SMS

mencionada se obtiene alguno de los mensajes nuevos que posee el dispositivo celular. El siguiente segmento de código muestra la utilización de las funciones mencionadas:

```
error = get_sms(@id_connection)
error <= 0 ? number=error.to_i*-1 : exception(error)
number.times { |i|
   m = get_msj(i,@id_connection) # Se pide cada mensaje
   if(m.type_sms.eql?("Inbox Message"))
      msj[nmsj] = m  if (m.error == 0)
      nmsj+=1 if m.error ==0
   end
}
```

Para el módulo de recepción de SMS se define un atributo receivetype. Este atributo puede tener valor de 0 y 1. En caso de tener valor 0, el teléfono se coloca en modo escucha y pide todos los mensajes nuevos cada cierto intervalo de tiempo. Existe la posibilidad de especificar el tiempo total

que el dispositivo celular se mantiene en este modo a través del atributo time. El tiempo especificado se toma en segundos. El siguiente fragmento de código implementa lo antes expuesto:

```
if receivetype==0
   verify(time,10){
      list = conn.execute(hash)  #Se traen los mensajes nuevos del teléfono
      ...
   }
end
```

Si el valor del atributo receivetype es 1, los mensajes nuevos del dispositivo celular son solicitados solo una vez.

Un dispositivo celular particular puede estar configurado para recibir y enviar, sin embargo, no puede hacer las dos cosas al mismo tiempo. Por esta razón deben realizarse distintas validaciones tal y como muestra el siguiente bloque de código:

```
if @adm.getconn.inject(false){|res,act| (act[1].phone_imei==i and act[1].
   status=='available' and (act[1].typec=='r' or act[1].typec=='sr')) || res
   }
   t=Thread.new{receive_internal(i){|x| yield x if block_given? }}
   t[:type]='r'
else
   @adm.log.error "Can't receive message." unless @adm.log.nil?
end
```

El bloque anterior es ejecutado por cada conexión configurada para recibir.

11.11.4 Pruebas

Las pruebas de esta iteración se realizaron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.20 se encuentran las distintas pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

La identificación de los dispositivos autorizados para enviar o recibir mensajes de texto fue realizada a través de un archivo de configuración. En la realización de las pruebas fue utilizado el siguiente archivo:

```
## config_sms.yml
   receive:
      imeis:
      ## ---- Lista de IMEI's
         - 353736027166810   #sony ericsson
         - 355078000896800   #motorola

   send:
      imeis:
      ## ---- Lista de IMEI's
         - 356662001124632   #nokia
         - 352008018253747   #samsung
```

Nro	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Capa de recepción, no múltiples dispositivos	Comprobar el correcto funcionamiento de la capa de recepción utilizando un dispositivo configurado para recibir y utilizando un dispositivo no configurado para recibir.	Terminación normal del programa. En el primer caso el programa debe terminar luego del tiempo especificado para recibir, habiendo procesado todos los mensajes recibidos. En el segundo caso el programa debe terminar mostrando una advertencia, ya que no se especifica un dispositivo configurado para recibir.
2	Capa de recepción, múltiples dispositivos	Comprobar el correcto funcionamiento de la capa de recepción utilizando más de un dispositivo configurado para recibir.	Terminación normal del programa. El programa debe terminar luego del tiempo especificado para recibir, adicionalmente debe haber procesado todos los mensajes recibidos por cada uno de los dispositivos celulares.
3	Tipo de recepción	Comprobar el correcto funcionamiento de la capa de recepción utilizando un tipo de recepción distinto al utilizado en pruebas anteriores (explicado en codificación)	Terminación normal del programa. El programa debe terminar luego de haber obtenido los mensajes nuevos almacenados en el(los) dispositivo(s) celular(es).

Tabla 11.20: Pruebas funcionales - Iteración 10

Utilizando el siguiente código fue probada la capa de recepción SMS:

```
1 sms = Smsruby.new
2 sms.receive([Lista de IMEI de teléfonos que se colocaran en modo de
     recepción]){|x|
3         # Bloque de código a ejecutar cuando llegue un mensaje
4 }
5 sms.close
```

La función close es utilizada para permitir la finalización de todas las conexiones de manera satisfactoria. Las pruebas realizadas fueron las siguientes:

- **Prueba 1 - Capa de recepción, sin múltiples dispositivos**

- Recepción utilizando un dispositivo configurado para recibir.

```
1         sms = Smsruby.new
2         sms.receive(['353736027166810']){|x|
```

```
3                   puts  'Mensaje Recibido: '+ x
4         }
5         sms.close
```

- **Resultado**:

```
::PROGRAM STARTED::
:: Iniciando conexión con id 0 phone_imei 353736027166810 Sony Ericsson::
:: Se abrió una conexión en el puerto /dev/ttyACM0 Tipo de la conexión es:
   Recepción ::
:: Se va a Recibir por la conexión con id 0 con imei 353736027166810::

Task started. Tue Jul 14 00:57:12 -0430 2009
Mensaje Recibido: Hola mundo 1 desde Sony Ericsson
Mensaje Recibido: Hola mundo 2 desde Sony Ericsson
Task done. Tue Jul 14 00:57:12 -0430 2009

Task started. Tue Jul 14 00:57:22 -0430 2009
Mensaje Recibido: Hola mundo 3 desde Sony Ericsson
Task donde. Tue Jul 14 00:57:22 -0430 2009
Task started. Tue Jul 14 00:57:32 -0430 2009
Mensaje Recibido: Hola mundo 4 desde Sony Ericsson
Task donde. Tue Jul 14 00:57:32 -0430 2009
```

El conjunto de impresiones colocadas muestran como es iniciada una conexiòn configurada para recibir (con imei 353736027166810 asociado), y como a travès de esta se reciben 4 mensajes de texto en un intervalo de 30 segundos transcurridos. El programa termina satisfactoriamente.

- Recepción utilizando un dispositivo no configurado para recibir.

```
1 sms = Smsruby.new
2 sms.receive(['356662001124632']){|x|
3         puts  'Mensaje Recibido: '+ x
4 }
5         sms.close
```

- **Resultado**:

```
::PROGRAM STARTED::
:: Iniciando conexión con id 0 phone_imei 356662001124632 Nokia::
:: Se abrió una conexión en el puerto /dev/ttyACM0 Tipo de la conexión es:
   Envío ::

No puede recibirse mensajes a través del dispositivo cuyo imei es:
   356662001124632
```

El programa termina con una advertencia, indicando que no pueden recibirse mensajes por una conexiòn que no està configurada para recibir.

- **Prueba 2 - Capa de recepción, múltiples dispositivos**

```
sms = Smsruby.new
sms.receive{|x|
        puts  'Mensaje Recibido'
}
sms.close
```

En este caso al no colocarse argumentos, se reciben mensajes por todos los dispositivos configurados para recibir que se encuentran conectados.

- **Resultado**:

```
::PROGRAM STARTED::
:: Iniciando conexión con id 0 phone_imei 353736027166810   Sony Ericsson::
:: Se abrió una conexión en el puerto /dev/ttyACM0 Tipo de la conexión es:
   Recepción ::
:: Iniciando conexión con id 1 phone_imei 355078000896800  Motorola::
:: Se abrió una conexión en el puerto /dev/ttyACM1 y el tipo de la conexión
   es: Recepción ::

:: Se va a recibir por la conexión con id 0 con imei 353616012272916 ::

Task started. Tue Jul 14 01:09:50 -0430 2009

:: Se va a recibir por la conexión con id 1 con imei 352008018253740 ::

Task started. Tue Jul 14 01:09:50 -0430 2009
Mensaje Recibido: Hola mundo 1 para dispositivo Sony Ericsson
Mensaje Recibido: Hola mundo 2 para dispositivo Sony Ericsson

Mensaje Recibido: Hola mundo 1 para dispositivo Motorola
Mensaje Recibido: Hola mundo 2 para dispositivo Motorola

Task done. Tue Jul 14 01:10:00 -0430 2009
Task done. Tue Jul 14 01:10:00 -0430 2009

Task started. Tue Jul 14 01:10:00 -0430 2009
Task started. Tue Jul 14 01:10:00 -0430 2009

Mensaje Recibido: Hola mundo 3 para dispositivo Sony Ericsson

Mensaje Recibido: Hola mundo 3 para dispositivo Motorola

Task done. Tue Jul 14 01:10:00 -0430 2009
Task done. Tue Jul 14 01:10:09 -0430 2009

Task started. Tue Jul 14 01:10:10 -0430 2009

Mensaje Recibido: Hola mundo 3 para dispositivo Sony Ericsson
```

```
Task done. Tue Jul 14 01:10:19 -0430 2009

Task started. Tue Jul 14 01:10:19 -0430 2009

Mensaje Recibido: Hola mundo 3 para dispositivo Motorola

Task donde. Tue Jul 14 01:10:19 -0430 2009
```

El conjunto de impresiones colocadas muestran como son iniciadas dos conexiònes configuradas para recibir (con imei 353736027166810 y 355078000896800 asociados), y como a través de éstas se reciben 4 mensajes de texto. El programa termina satisfactoriamente.

- **Prueba 3 - Tipo de recepción**

```
1 sms = Smsruby.new
2 sms.receiver.receivetype = 1
3 sms.receive(['353736027166810','355078000896800']){|x|
4         puts  'Mensaje Recibido'
5 }
6 sms.close
```

El tipo de receive se modica a través del atributo receivetype. Para esta prueba se utilizaron 2 conexiones configuradas para recibir.

- **Resultado**:

```
::PROGRAM STARTED::
:: Iniciando conexión con id 0 phone_imei 353736027166810   Sony Ericsson::
:: Se abrió una conexión en el puerto /dev/ttyACM0 Tipo de la conexión es:
   Recepción ::
:: Iniciando conexión con id 1 phone_imei 355078000896800  Motorola::
:: Se abrió una conexión en el puerto /dev/ttyACM1 y el tipo de la conexión
   es: Recepción ::

:: Se va a recibir por la conexión con id 0 con imei 353616012272916 ::
:: Se va a recibir por la conexión con id 1 con imei 352008018253740 ::

Task started. Tue Jul 14 01:14:53 -0430 2009
Task started. Tue Jul 14 01:14:53 -0430 2009

Mensaje Recibido: Hola mundo 1 para dispositivo Sony Ericsson
Mensaje Recibido: Hola mundo 2 para dispositivo Sony Ericsson
Mensaje Recibido: Hola mundo 1 para dispositivo Motorola

Task donde. Tue Jul 14 01:15:04 -0430 2009

Mensaje Recibido: Hola mundo 2 para dispositivo Motorola
Task donde. Tue Jul 14 01:15:04 -0430 2009
```

El conjunto de impresiones colocadas muestran como son iniciadas dos conexiònes configuradas para recibir (con imei 353736027166810 y 355078000896800 asociados), y como a través de éstas se reciben 4 mensajes de texto (dos por cada dispositivo celular). El programa termina satisfactoriamente.

11.12 Iteración 11

Del 13-Jul-2009 al 17-Jul-2009

En esta iteración el middleware SMS es empaquetado como un gem de Ruby, para posteriormente ser colocado en los repositorios de RubyForge y GitHub.

11.12.1 Planificación

Para esta iteración se establece la meta *Creación de una gem de Ruby que contenga la extensión realizada y el middleware SMS*. En la Tabla 11.21 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	13 - 07 - 2009	Empaquetar el código bajo la estructura de un gem de Ruby.
2	13 - 07 - 2009	Importar el gem a los repositorios de RubyForge y Github.

Tabla 11.21: Historia de usuario - Iteración 11

11.12.2 Diseño

Para que el gem pueda funcionar correctamente debe contener todos los archivos necesarios, ordenados en una estructura que tenga sentido. Existen algunas convenciones generales como por ejemplo, el código fuente debe ir en una carpeta llamada lib, las extensiones utilizadas deben estar en una carpeta llamada ext y siempre se debe incluir un archivo README.

Adicionalemente es necesario crear un archivo que contenga información clave acerca del gem, como datos acerca de quien la realizó, la versión, la fecha, página web del gem, lista de archivos que van dentro del gem, entre otros. Este archivo es la especificación del gem o el gemspec.

Una vez creado el gem, se procede a colocarlo en los repositorios de RubyForge y Github bajo una licencia de software libre, pudiendo ser descargada por todo aquel que desee utilizarla o mejorarla.

11.12.3 Codificación

La Figura 11.11 muestra la estructura final del gem.

Una vez ordenados los archivos siguiendo las convenciones antes mencionadas, debe realizarse el archivo gemspec:

```
1  SPEC = Gem::Specification.new do | s |
2     s.name = "smsRuby"
3     s.version = "1.0.0"
4     s.homepage = "http://smsRuby.rubyforge.org"
5     s.author = "Alejandro García, Yennifer Chacón"
6     s.email = "hewital@gmail.com, ychacon@gmail.com"
7     s.platform = Gem::Platform::CURRENT
8     s.extensions = ["ext/extconf.rb"]
9     s.summary = "An easy way to send and receive SMS message."
10    s.files = Dir.glob("{doc,ext,lib}/**/*")
11    s.require_paths = ["lib"]
12    s.has_rdoc = true
```

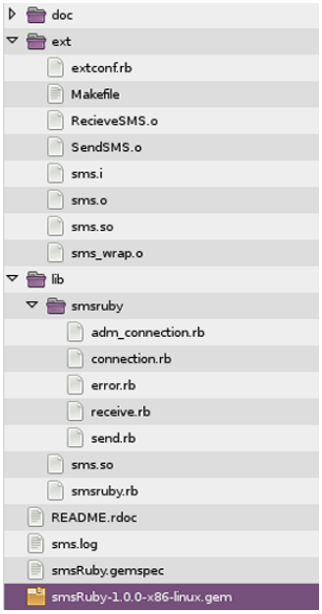

Figura 11.11: Estructura del gem

```
13    s.extra_rdoc_files = ["README.rdoc"]
14    s.add_dependency("sqlite3-ruby",">=1.2.4")
15  end
```

Una vez especificadas las reglas en el archivo gemspec que permitan indicar a RubyGems como empaquetar el código, se utiliza el siguiente comando para construir el gem:

```
1  gem build smsRuby.gemspec
2  Successfully built RubyGem
3  Name:smsRuby
4  Version : 1.0.0
5  File: smsRuby-1.0.0-x86-linux.gem
```

Luego, el gem puede ser instalado con el siguiente comando:

```
1  gem install smsRuby-1.0.0-x86-linux.gem
2  Successfully installed smsRuby-1.0.0-x86-linux
3  gem installed
4  Installing ri documentation for smsRuby-1.0.0-x86-linux . . .
5  Installing RDoc documentation for smsRuby-1.0.0-x86-linux . . .
```

11.12.4 Pruebas

Las pruebas de esta iteración se realizaron bajo el mismo ambiente descrito en la iteración 4.

Pruebas Funcionales

En la Tabla 11.22 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades realizadas en esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Creación del gem SMS	Verificar la correcta generación del gem, que incorpore la extensión realizada y el middleware SMS.	Hacer uso del gem creado, utilizando cada una de las funciones que este contiene.

Tabla 11.22: Pruebas funcionales - Iteración 11

- **Prueba 1 - Creación del gem SMS**

Para realizar las pruebas se utilizó irb, con lo que se desea probar que el paquete SmsRuby instalado, funciona de manera adecuada.

- **Resultado**:

```
/$ irb
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require 'smsruby'
=> true
irb(main):003:0> sms = Smsruby.new
=> #<Smsruby:0xb796aa84 @receiver=#<Receive:0xb796a9e4 @time=0, @receivetype
   =0>, @sender=#<Sender:0xb796aa48>>
irb(main):004:0>
```

De esta manera se puede constatar que el paquete instalado funciona correctamente, bajo la estructura de un gem de Ruby, y además este puede ser instanciado.

11.13 Iteracion 12

Del 20-Jul-2009 al 7-Ago-2009

En esta iteración se desarrollan las aplicaciones de prueba que utilizan el gem creado en iteraciones anteriores.

11.13.1 Planificación

Para esta iteración se establece la meta *Crear aplicación de prueba que haga uso del gem dsarrollado y realizar las pruebas finales del mismo.* En la Tabla 11.23 se encuentran las distintas historias de usuario correspondientes a esta iteración.

Número	Fecha	Descripción
1	20 - 07 - 2009	Crear la aplicación de prueba - web.
2	27 - 07 - 2009	Crear la aplicación de prueba - Standalone.
3	3 - 08 - 2009	Realizar las pruebas finales del gem.

Tabla 11.23: Historia de usuario - Iteración 12

11.13.2 Diseño

Para probar las funcionalidades del gem se crearon dos aplicaciones que lo utilizan, una aplicación web y una aplicación standalone, mostrando así, que el gem puede ser utilizado en ambos escenarios. La primera aplicación fue realizada haciendo uso del framework Ruby on Rails. En ésta se muestran las funcionalidades básicas de envío y recepción del gem.

La segunda aplicación fue realizada haciendo uso de FXRuby, una librería para el desarrollo de aplicaciones con interfaz gráfica. En este punto, se busca mostrar todas las funcionalidades del gem de una forma gráfica y más detallada, para que puedan ser apreciadas las distintas opciones que éste ofrece.

Para verificar el correcto funcionamiento del gem, así como su rendimiento, se realizaron un conjunto de pruebas que buscan corregir posibles errores.

11.13.3 Codificación

Con la aplicación web se muestran las funcionalidades básicas del gem, utilizando las funciones de envío y recepción en su forma mas sencilla. Se muestra una interfaz donde sólo es permitido el envío de un mensaje a un número especificado de manera estática en el código. Es posible también la lectura de los mensajes que se encuentren en el dispositivo celular conectado. La Figura 11.12 muestra la pantalla inicial que presenta la aplicación al iniciarse.

Al pulsar el botón de envío se ejecuta el siguiente código:

```
1  ...
2  @sms.sender.dst = ['04123853283']
3  @sms.send('Probando smsRuby desde RoR')
4  ...
```

Figura 11.12: Pantalla inicial en aplicación Web

Donde @sms es una instancia de la clase Smsruby y permite tanto el envío como la recepción de mensajes, ya que contiene un objeto Sender y uno Receive. Para enviar un mensaje es necesario indicar el número de teléfono del destinatario, así como el mensaje que será enviado.

Para leer los mensajes que se encuentran en el dispositivo celular, se utiliza el botón de recibir. Al pulsarlo el siguiente código es ejecutado:

```
1  ...
2  @sms.receiver.receivetype = 1
3  @sms.receive(['357691005340208']){ |mess,dest|
4    @mensajes.merge!({ mess => dest})
5  }
6  @sms.wait_receive
7  ...
```

Como sólo se quiere listar los mensajes que se encuentran en el dispositivo celular al momento de ejecutar el código, al tipo de recepción se le coloca e valor 1. En caso de que se desee recibir mensajes de forma continua el tipo de recepción debe ser colocado en 0. El método receive obtiene un bloque de código, donde se indica que instrucciones ejecutar cada vez que se recibe un mensaje. En este caso cada mensaje se agrega al hash @mensajes.

Con esta aplicación se muestra que el gem construido puede ser utilizado de manera sencilla y en ambientes web.

Para mostrar todas las utilidades que brinda el gem de una forma mas detallada, se creó una aplicación standalone utilizando FXRuby (La Figura 11.13 muestra la interfaz de la aplicación realizada). Desde esta aplicacion se pueden enviar mensajes SMS indicando cada uno de los parámetros necesarios

de manera manual, con un archivo de configuración o a través de una base de datos. La recepción de los mensajes también puede ser configurada, indicando si desean obtenerse los mensajes sólo una vez o hacerlo de una manera continua. Adicionalmente, con la aplicación pueden administrarse los dispositivos conectados, indicando si son dispositivos configurados para envío, recepción o ambos; de la misma forma, se puede modificar el archivo de configuración necesario para poder utilizar el gem smsRuby.

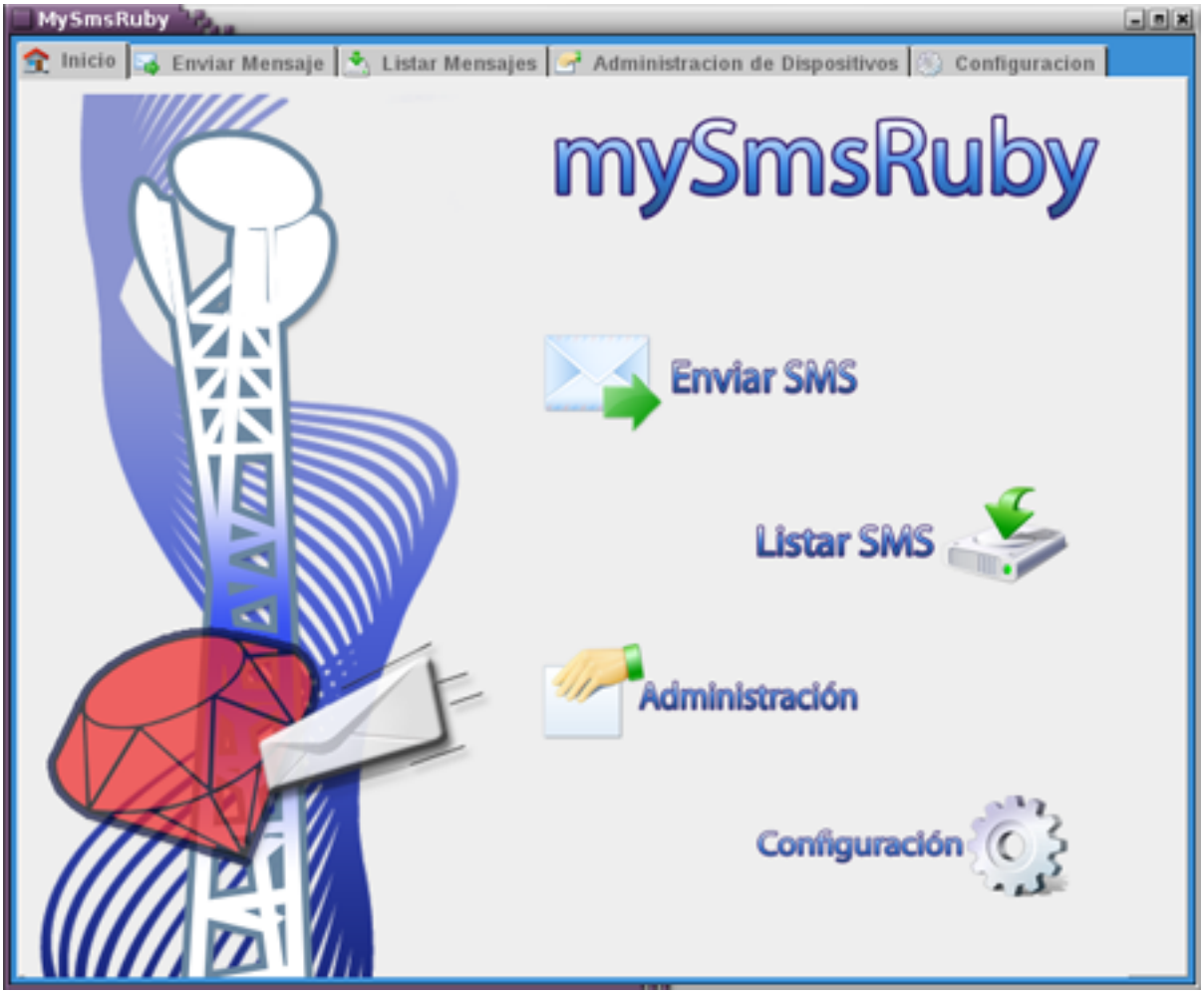

Figura 11.13: Aplicación mySmsRuby

El envío de mensajes puede realizarse de distintas maneras. La primera, es asignando directamente el valor de cada uno de los parámetros necesarios en la clase de envío (Plainsend), la segunda forma a través de un archivo de configuración (Configsend) donde se encuentren los parámetros de envío, y la última forma es obteniendo los parámetros desde una base de datos (BDsend).

Con el siguiente código se permite el envío de mensajes utilizando el modo correspondiente:

```
1 enviar = FXButton.new(buttons, "Enviar",:target => self, :opts =>
      BUTTON_NORMAL|LAYOUT_RIGHT)
2
3 enviar.connect(SEL_COMMAND) do |sender, sel, data|
```

```
4     sendtype=nil
5     ar=[]
6     case envio_opt.value
7       when 0
8         ar.push(dest.text)
9         msg = Message.new(ar, @text.text,gp_datatarget.value, smsc.text,nil)
10      when 1
11        sendtype=Configsend.new
12        msg = Message.new(nil, @text.text, nil, nil,nil)
13      when 2
14        sendtype=BDsend.new
15        msg = Message.new(nil, @text.text, nil, nil,nil)
16      end
17      @adm.send(sendtype,msg)
18  end
```

La Figura 11.14 muestra la interfaz utilizada para el envío de mensajes.

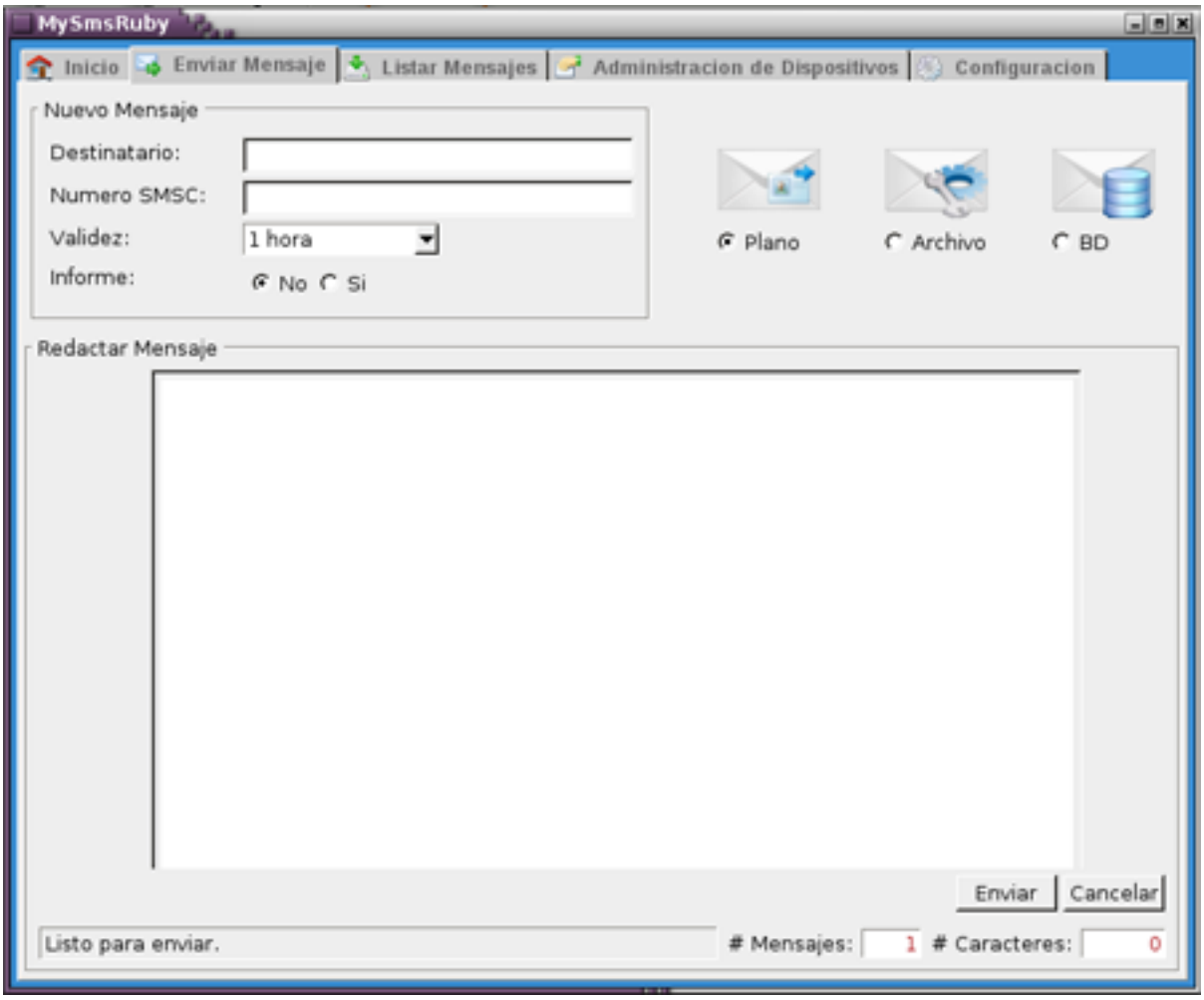

Figura 11.14: Aplicación mySmsRuby - Envío de SMS

La recepción de mensajes también puede ser realizada de distintas maneras. una de ellas es listando los mensajes del dispositivo celular sólo una vez al momento de ejecutar la petición y la otra forma listando los mensajes del dispositivo cada cierto tiempo.

Para escoger el tipo de recepción deseado se debe asignar el valor de 0 o 1 al atributo receivetype del objeto Receiver, siendo 0 el valor necesario para la recepción continua y 1 para listar los mensajes sólo una vez.

La función recibir recibe un bloque de código el cual se ejecuta cada vez que un mensaje es recibido. En el caso de la aplicación, este bloque contiene instrucciones para la construcción de la interfaz gráfica, donde se van listando los mensajes recibidos. En el siguiente código se muestra la llamada a la función recibir del gem.

```
@adm.receive(imeis){ |mess,dest|
  @contmsj=@contmsj+=1
  @nummen.text=@contmsj.to_s
  @mensajes.push(mess)
  r = @table.numRows
  r = r-1
  if mess.status == 'Read'
    icon = @read
  else
    icon = @unread
  end
  a = (@config['phones'].select { |x,y| x.to_s.eql?(dest)})
  @table.setItemIcon(r, 0, icon)
  @table.setItemJustify(r, 0, FXTableItem::CENTER_X)
  @table.setItemText(r, 1,mess.date)
  @table.setItemJustify(r,1,FXTableItem::LEFT)
  @table.setItemText(r, 2,mess.source_number)
  @table.setItemJustify(r,2,FXTableItem::LEFT)
  @table.setItemText(r, 3,a[0][1].to_s)
  @table.setItemJustify(r,2,FXTableItem::LEFT)
  @table.setItemText(r, 4,mess.text)
  @table.setItemJustify(r,3,FXTableItem::LEFT)
  @table.insertRows(@table.numRows, 1, false)
}
```

La aplicación muestra una tabla donde se van agregando los mensajes cuando son recibidos por el dispositivo celular. En el código anterior, se agrega una nueva fila a la tabla por cada mensaje nuevo que es recibido, en dicha fila se mostrará información del mensaje. En la Figura 11.15 se muestra la interfaz utilizada para la recepción de mensajes.

Para utilzar el gem, es necesario incluir un archivo de configuración, donde son especificados valores para distintas opciones. Es posible especificar la función que va a tener algún dispositivo al ser conectado al computador (envío, recepción o envío/recepción), también es posible parametrizar las opciones para el envío de los mensajes SMS cuando desee utilizarse o bien un archivo de configuración o una base de datos.

Utilizando la interfaz se puede modificar este archivo de configuración sin necesidad de conocer la estructura exacta del mismo. Las Figuras 11.16 y 11.17 muestran la interfaz utilizada para modificar el archivo de configuración.

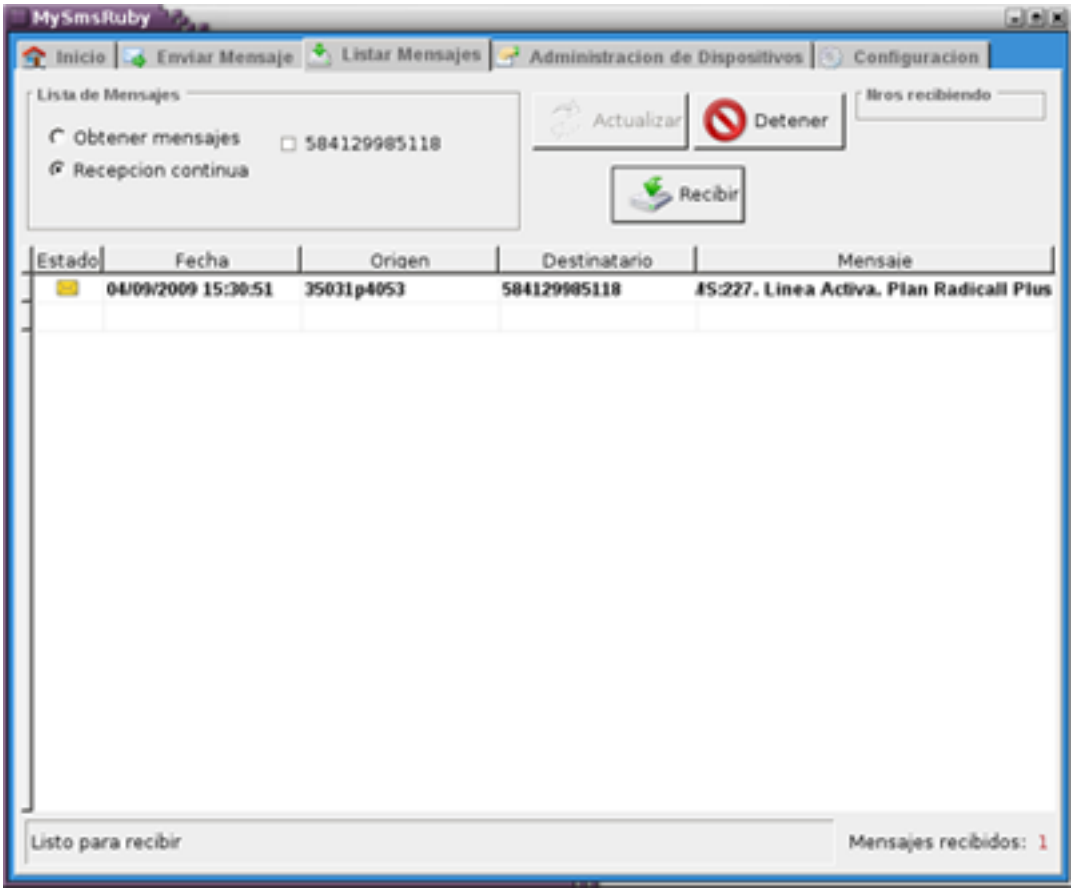

Figura 11.15: Aplicación mySmsRuby - Recepción de SMS

11.13.4 Pruebas

En la Tabla 11.24 se encuentran las pruebas funcionales realizadas para verificar el correcto funcionamiento de las actividades correspondientes a esta iteración.

Nro.	Nombre de la Prueba	Objetivos	Resultados Esperados
1	Pruebas Generales	Comprobar el correcto funcionamiento del gem.	Lograr el envío y recepción de mensajes SMS utilizando las distintas opciones ofrecidas por el gem desde las aplicaciones construidas.

Tabla 11.24: Pruebas funcionales - Iteración 12

- **Prueba 1 - Pruebas Generales**

Para probar el gem de una forma general y comprobar el correcto funcionamiento de cada una de las funcionalidades que éste ofrece, se realizaron distintas pruebas a través de las aplicaciones creadas.

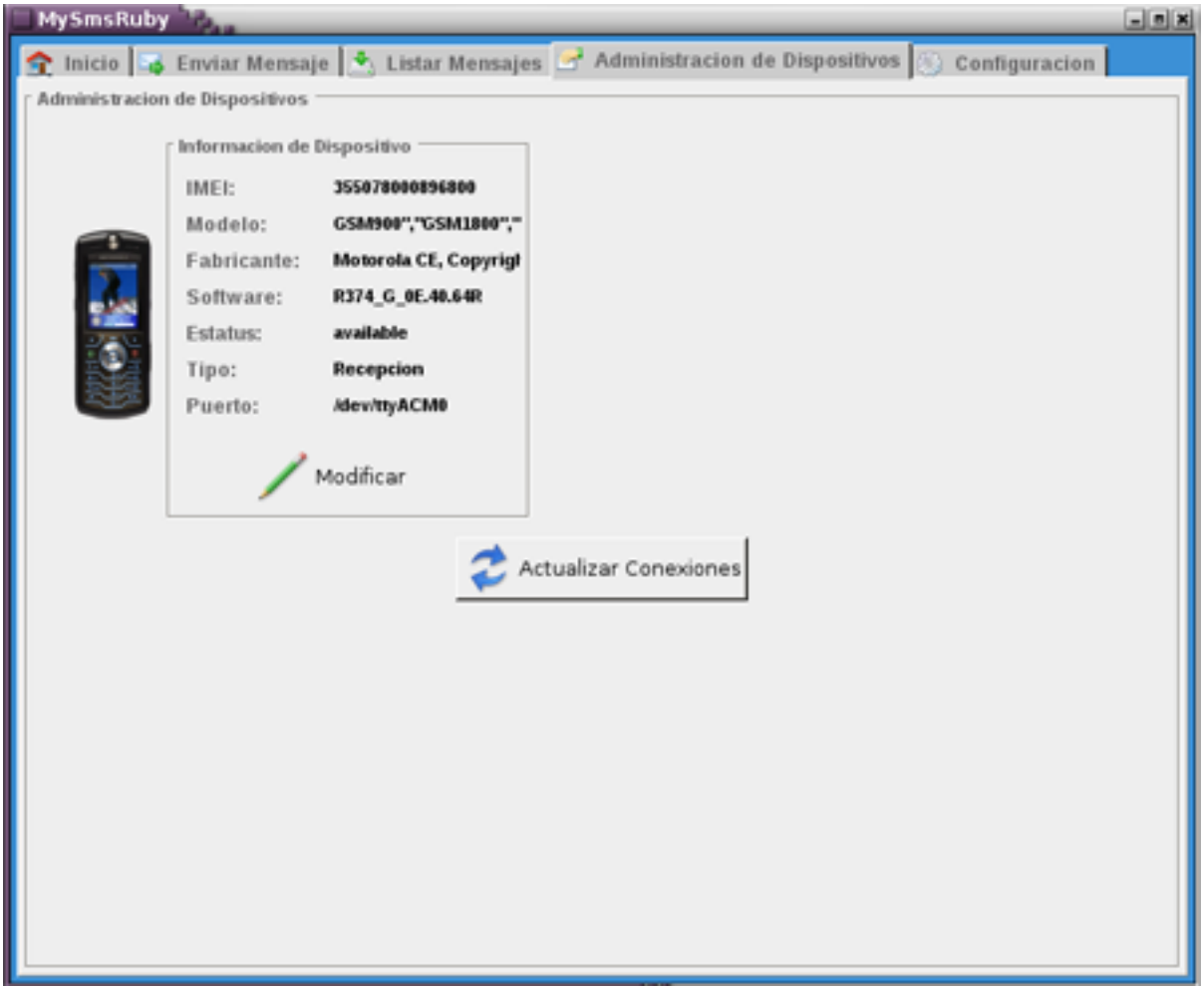

Figura 11.16: Aplicación mySmsRuby - Administrador de dispositivos

- **Prueba 1.1** Envío de 25 mensajes SMS a través de un dispositivo celular.

- **Resultado**

Al enviar 25 mensajes desde un solo dispositivo celular se registró un tiempo de 5:52 minutos, todos los mensajes fueron recibidos satisfactoriamente. En este caso, algunos mensajes no pudieron ser enviados al primer intento de envío y entraron a la cola de emergencia, de donde posteriormente pudieron ser enviados, logrando así, que todos los mensajes se recibieran satisfactoriamente.

- **Prueba 1.2** Envío de 25 mensajes SMS a través de dos dispositivos celulares.

- **Resultado**

Al enviar 25 mensajes desde dos dispositivos celulares se registró un tiempo de 2:07 minutos, todos los mensajes fueron recibidos satisfactoriamente.

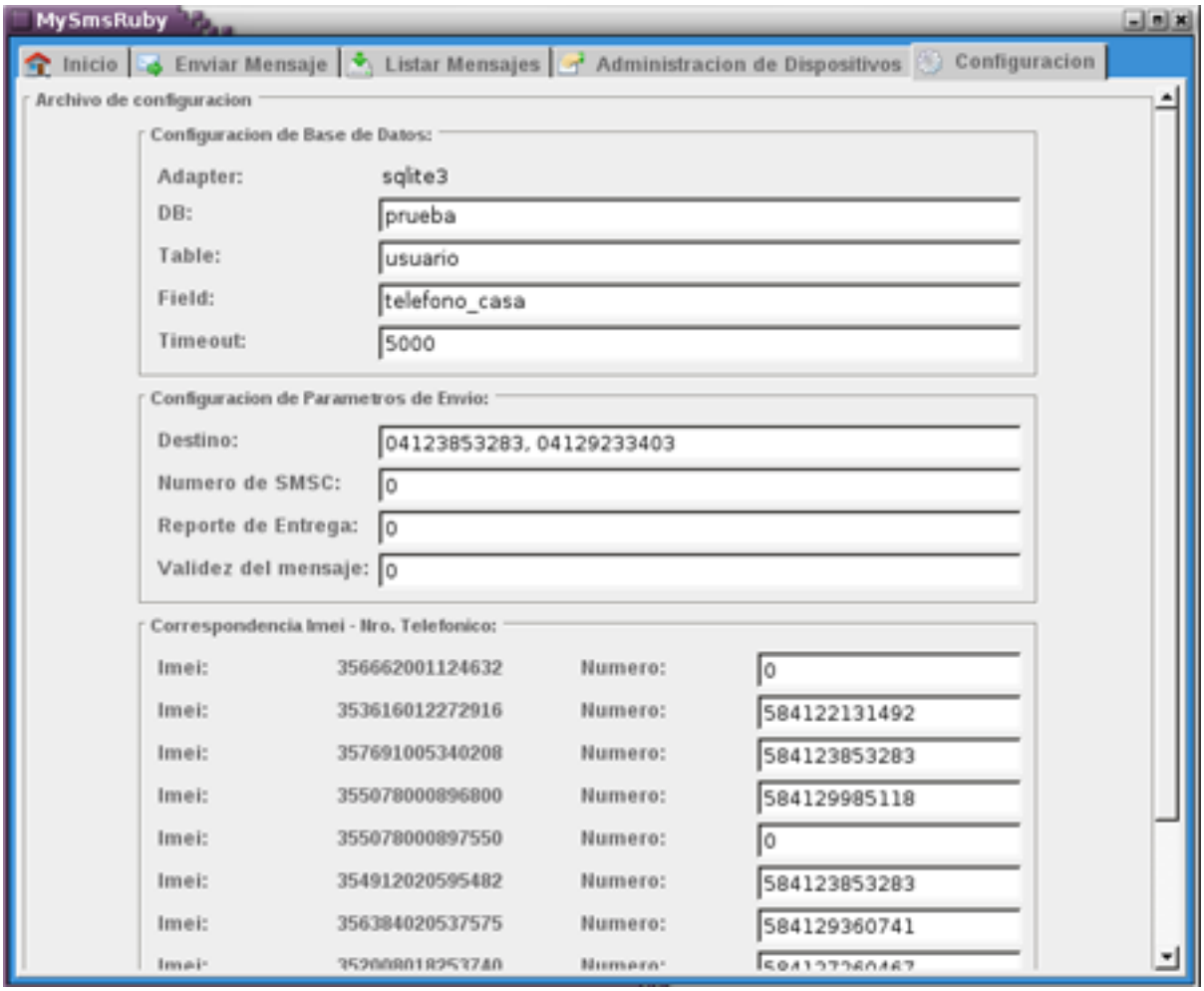

Figura 11.17: Aplicación mySmsRuby - Configuración

- **Prueba 1.3** Envío de 25 mensajes SMS a través de cinco dispositivos celulares.

- **Resultado**

Al enviar 25 mensajes desde cinco dispositivos celulares se registró un tiempo de 1:20 minutos, todos los mensajes fueron recibidos satisfactoriamente.

- **Prueba 1.4** Envío de 100 mensajes SMS a través de cinco dispositivos celulares.

- **Resultado**

Al enviar 100 mensajes desde un solo dispositivo celular se registró un tiempo de 6:20 minutos, todos los mensajes fueron recibidos satisfactoriamente.

Parte IV

Conclusiones

Capítulo 12

Conclusiones

El presente trabajo se basó en la investigación y desarrollo, de forma que pudieran disponerse de las herramientas y soluciones necesarias que permitieran cumplir con los objetivos propuestos. El resultado de dicho desarrollo fue SmsRuby, un producto destinado a ofrecer un middleware que permita el desarrollo sencillo de aplicaciones que desean incorporar la tecnología SMS.

El desarrollo del producto se realizó utilizando el lenguaje de programación Ruby y el lenguaje de programación C/C++. El primero se utilizó para la implementación del middleware SMS, incorporando cada una de sus capas, mientras que el segundo se utilizó para realizar las extensiones de Ruby, que permitan la comunicación entre el computador y los dispositivos celulares a través de los comandos AT. El producto final fue empaquetado, construyendo así, un gem de Ruby que puede ser administrado a través de RubyGems.

Fue utlizado el lenguaje Ruby debido a que el diseño de éste se enfoca en el programador, ofreciendo un conjunto de funciones que facilitan su tarea. Para implementar la comunicación entre el computador y los dispositivos celulares fue necesario realizar la integración de los lenguajes de programación Ruby y C. La forma más sencilla encontrada para realizar dicha integración, fue a través de la herramienta SWIG, la cual permite construir extensiones de manera rápida y sencilla.

A través de los comandos AT se hizo efectiva la comunicación entre el middleware SMS y los distintos dispositivos celulares. Sin embargo, no todos los comandos AT utilizados e implementados por la tecnología GSM, funcionaron de manera adecuada en la totalidad de dispositivos. Para solventar el problema se utilizaron diversos comandos para lograr la compatibilidad con todos los dispositivos probados. Básicamente el problema fue encontrado en los dispositivos y no en los comandos; en efecto, a pesar de que dichos comandos son estándares y reconocidos por los teléfonos, la forma en que éstos manejan su memoria interna no siempre es igual, lo cual causa problemas en la ejecución de algunos comandos.

La implementación para el envío de mensajes fue realizada de manera satisfactoria utilizando los comandos AT extendidos e incorporando una capa de envío en el middleware SMS. Sin embargo, se presentó un inconveniente al momento de enviar mensajes de texto largos (más de 160 caracteres); en ese caso, algunos dispositivos presentaron conflictos tratando de enviar el mensaje, por lo que en caso de fallo, dicho mensaje fue dividido para ser enviado por partes, cada una con su respectiva identificación.

Por otro lado la implementación del módulo de recepción fue más complicada, debido a que muchos dispositivos manejan de forma diferente la memoria donde son almacenados los mensajes. Por esta

razón, fue necesario utilizar diferentes comandos para asegurar la compatibilidad con los distintos dispositivos celulares. Con la utilización de dichos comandos y la implementación de la capa de recepción en el middleware SMS, fue posible colocar un dispositivo celular en modo escucha, esperando la llegada de nuevos mensajes, o simplemente obtener todos los mensajes almacenados.

Tanto para el envío como la recepción de mensajes de texto se incorporaron funcionalidades para el manejo de múltiples conexiones, lo que permitió utilizar dispositivos dedicados sólo a la tarea de envío o a la tarea de recepción. Adicionalmente aplicar una distribución de carga en el envió de mensajes.

Con la inserción de un middleware SMS como la capa de más alto nivel (en donde residen las aplicaciones), se logró un grado de encapsulamiento y abstracción de los servicios de más bajo nivel. Este middleware introduce un conjunto de interfaces uniformes de programación de aplicaciones, que son usadas para invocar nuevos servicios (lo que ayuda a aislar las aplicaciones de cambios en la plataforma, sistemas operativos, etc.) que permiten la interacción con las funcionalidades ofrecidas por la tecnología SMS.

De esta forma se facilita considerablemente el desarrollo de aplicaciones basadas en la tecnología SMS, simplificando la programación y permitiendo a los programadores crear o modificar las aplicaciones de manera más rápida.

Siguiendo el proceso de desarrollo XP se logró reducir el tiempo de implementación utilizando sus actividades y principios, permitiendo dividir iteraciones por alcance y ajustarlas a las necesidades presentadas, dando la flexibilidad necesaria para cubrir e implementar cada uno de los requerimientos propuestos.

Al utilizar la metodología AM se simplificó el proceso de modelado y diseño de la documentación, ya que no se utilizaron rígidos modelos. Por el contrario, se permitió a los desarrolladores, a través de un conjunto de principios y valores, crear modelos simples y sencillos adaptables a cualquier tipo de cambio para lograr un trabajo de calidad.

Un aspecto importante que vale la pena resaltar, es la posibilidad de extender las funcionalidades del gem, lo que permitirá agregar mejoras en diversos aspectos, en particular, la posibilidad de dar soporte a dispositivos de última generación y la incorporación de nuevos formatos de mensaje.

También es importante destacar que el desarrollo del sistema estuvo orientado más que todo a las plataformas Windows y Linux. Sin embargo, el proceso de adaptación para otras plataformas como MAC es relativamente sencillo. Por ejemplo para el caso de MAC, solamente será necesario generar un objeto bundle (por ejemplo sms.bundle) e incorporar dicho objeto a los módulos realizados en Ruby.

Finalmente, se puede concluir que los objetivos planteados para el desarrollo de este trabajo fueron alcanzados, debido a que se logró desarrollar un gem de Ruby, que encapsula un middleware SMS capaz de ofrecer servicios de envío y recepción de mensajes de texto.

12.1 Mejoras a Futuro

Es importante destacar que el sistema elaborado es un producto escalable, es por esto que se pueden agregar nuevas funcionalidades, entre las cuales se encuentran:

- Implementar el envío y recepción de mensajes utilizando otros tipos de conexión entre el dispositivo celular y el computador, como por ejemplo el uso de tecnología Bluetooth.
- Dar soporte al envío y recepción de mensajes multimedia (MMS).
- Elaboración de un plugin para Rails utilizando el gem construido, para el desarrollo de aplicaciones web que integren la tecnología SMS.

Referencias

[agilemodeling.com, 2008] agilemodeling.com (2008). Agile modeling (am) home page effective practices for modeling and documentation. http://www.agilemodeling.com. [21]

[Baird, 2007] Baird, K. (2007). *Ruby by Example: Concepts and Code*. No Starch Press. [55]

[Bakken, 2000] Bakken, D. E. (2000). Middleware. http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf. [26, 27]

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained*. Addison-Wesley Professional. [15]

[Berube, 2007] Berube, D. (2007). *Practical Ruby Gems*. Apress. [54]

[Bodic, 2003] Bodic, G. L. (2003). *Mobile Messaging Technologies and Services*. Wiley. [29]

[Canós and Letelier, 2002] Canós, J. and Letelier, P. (2002). Métodologías Ágiles en el desarrollo de software. [13]

[Carlson and Richardson, 2006] Carlson, L. and Richardson, L. (2006). *Ruby Cookbook*. O'Reilly. [63, 64]

[Fulton, 2006] Fulton, H. (2006). *The Ruby Way: Solutions and Techniques in Ruby Programming*. Addison Wesley Professional. [54]

[Home, 2008] Home, D. (2008). Sms tutorial: Introduction to at commands. http://www.developershome.com/sms/atCommandsIntro.asp. [41]

[Jesus Carretero Perez and Costoya, 2001] Jesus Carretero Perez, Felix Garcia Caballeira, P. d. M. A. and Costoya, F. P. (2001). *Sistemas operativos, Una vision aplicada*. McGraw Hill, 1 edition. [25]

[Olsen, 2008] Olsen, R. (2008). *Design Patterns in Ruby*. Addison-Wesley. [102]

[Ruby-Lang, 2008] Ruby-Lang (2008). Ruby programming language. http://www.ruby-lang.org/en/. [53]

[Stalling, 2005] Stalling, W. (2005). *Sistemas operativos, aspectos internos y principios de diseno*. Prentice Hall, 5 edition. [25]

[SWIG, 2008] SWIG (2008). Swig: Simplified wrapper and interface generator. http://www.swig.org/. [64]

[Thomas, 2006] Thomas, D. (2006). *Programming Ruby, The Pragmatic Programmers Guide*. The Pragmatic Bookshelf, 2 edition. [55, 61, 62]

[xprogramming.org, 2008] xprogramming.org (2008). Xprogramming.com - an agile software development resource. http://www.xprogramming.com. [15]